An Efficient Algorithm for Mining Frequent Sequences by a New Strategy without Support Counting

Ding-Ying Chiu Yi-Hung Wu Arbee L.P. Chen[*]
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
dr908312@cs.nthu.edu.tw yihwu@mx.nthu.edu.tw alpchen@cs.nthu.edu.tw

Abstract

Mining sequential patterns in large databases is an important research topic. The main challenge of mining sequential patterns is the high processing cost due to the large amount of data. In this paper, we propose a new strategy called DIrect Sequence Comparison (abbreviated as DISC), which can find frequent sequences without having to compute the support counts of non-frequent sequences. The main difference between the DISC strategy and the previous works is the way to prune non-frequent sequences. The previous works are based on the anti-monotone property, which prune the non-frequent sequences according to the frequent sequences with shorter lengths. On the contrary, the DISC strategy prunes the non-frequent sequences according to the other sequences with the same length. Moreover, we summarize three strategies used in the previous works and design an efficient algorithm called DISC-all to take advantages of all the four strategies. The experimental results show that the DISC-all algorithm outperforms the PrefixSpan algorithm on mining frequent sequences in large databases. In addition, we analyze these strategies to design the dynamic version of our algorithm, which achieves a much better performance.

1. Introduction

Mining sequential patterns from a large database is important and interesting to the fundamental research in the data mining community [1][6][8][11][12]. It is also useful for a variety of applications such as marketing data analysis and stock trend prediction. There are three major topics in this research field. One is to develop efficient algorithms for mining sequential patterns [9][13][18]. Another is to add restrictions on the sequential patterns to be mined, such as the ones in a noisy environment [17] and the ones that satisfy some constraints [5][10]. The other is to apply the techniques of mining sequential patterns to special data types, such as the Web [16], music [7], and biological data [3][15].

Based on the problem definition in [1], a large 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* [1].

Following the definitions in [1], the *length* of a sequence is the total number of item occurrences in it. A *k-sequence* stands for a sequence with length k. Let S_A and S_B respectively denote two sequences $<A_1A_2…A_n>$ and $<B_1 B_2…B_m>$, where A_i's and B_j's are itemsets and $m \ge n$. If there exist integers $i_1<i_2< … <i_n$ such that $A_1 \subseteq B_{i_1}$, $A_2 \subseteq B_{i_2}$, …, and $A_n \subseteq B_{i_n}$, it is said that S_B *contains* S_A and S_A is a *subsequence* of S_B. Furthermore, if a customer sequence contains a sequence S_A, we call that the customer sequence *supports* S_A. The *support count* of a sequence is the number of customer sequences that support it. If the support count of a sequence is larger than a user-specified *minimum support count*, we call it a *frequent sequence*. Given a database of customer sequences, the goal of this paper is to efficiently find all the frequent sequences.

1.1. Related works

The main challenge toward the problem of mining sequential patterns is the high processing cost due to a large amount of data. Many algorithms have been proposed to speed up the mining process. The representative ones are GSP [13], SPADE [18], SPAM [2], and PrefixSpan [9]. Srikant and Agrawal [13] adopt a bottom-up approach in the GSP algorithm, which generates frequent 1-sequences first, then frequent 2-sequences, and so on. This approach generates candidate k-sequences from frequent (k-1)-sequences in iteration based on the *anti-monotone* property that all the subsequences of a frequent sequence must be frequent. In any iteration, the candidate k-sequences are determined to be frequent based on their

[*] Contact author

support counts. The GSP algorithm costs a lot to decompose the customer sequences for the computation of support counts.

Another obvious problem of the GSP algorithm is the large number of candidate sequences. To solve this problem, Zaki [18] introduces the *lattice* concept in the SPADE algorithm to divide the candidate sequences into groups by items such that each group can be completely stored in the main memory. In addition, this algorithm uses the *ID-List* technique to reduce the costs for computing support counts. An ID-list of a sequence keeps a list of pairs, which indicate the positions that it appears in the database. In a pair, the first value stands for a customer sequence and the second refers to a transaction in it, which contains the last itemset of the sequence. For the example database in Table 1, the ID-list of sequence <(a, g)(b)> is <(1,2), (1,6), (4,3), (4,4)>, where the pair (1,2) means that this sequence appears in the first customer sequence and ends in the second transaction. Note that a sequence may appear more than once in the same customer sequence, and therefore more than one pair will be recorded.

Table 1: The example database

CID	Customer Sequences
1	(a, e, g)(b)(h)(f)(c)(b, f)
2	(b)(d, f)(e)
3	(b, f, g)
4	(f)(a, g)(b, f, h)(b, f)

The SPADE algorithm also adopts a bottom-up approach to generate frequent sequences with different lengths. By iteration, this approach computes the support count of a candidate k-sequence generated by merging the ID-lists of any two frequent (k-1)-sequences with the same (k-2)-prefix. Consider the same database in Table 1. To compute the support count of sequence <(a, g)(h)(f)>, the SPADE algorithm merges the two ID-lists of sequences <(a, g)(h)> and <(a, g)(f)>, which are <(1,3), (4,3)> and <(1,4), (1,6), (4,3), (4,4)> respectively. As a result, the ID-list of sequence <(a, g)(h)(f)> is <(1, 4), (1, 6), (4, 4)>, indicating that this sequence appears in the first and the fourth customer sequences and therefore has a support count of 2. The SPADE algorithm costs a lot to repeatedly merge the ID-lists of frequent sequences for a large number of candidate sequences. To reduce this cost of merging, Ayres et al. [2] adopt the lattice concept in the SPAM algorithm but represent each ID-list as a *vertical bitmap*. The SPAM algorithm is efficient under the assumption that all the bitmaps can be completely stored in the main memory.

On the other hand, Pei et al. [9] employ the *projection* scheme in the PrefixSpan algorithm to project the customer sequences into overlapping groups called *projected databases* such that all the customer sequences in each group have the same prefix which corresponds to a frequent sequence. For the example database in Table 1, assuming that the minimum support count is two, the PrefixSpan algorithm first scans the database to find the frequent 1-sequences, i.e. <(a)>, <(b)>, <(e)>, <(f)>, <(g)>, and <(h)>. After that, this algorithm generates the projected database for each frequent 1-sequence. For instance, Table 2 shows the projected database of <(a)>. For this projected database, the PrefixSpan algorithm continues the discovery of frequent 1-sequences to form the frequent 2-sequences with prefix <(a)>. In this way, the PrefixSpan algorithm recursively generates the projected database for each frequent k-sequence to find frequent (k+1)-sequences. Obviously, the PrefixSpan algorithm costs a lot to recursively generate a large number of projected databases.

Table 2: The projected database of <a>

CID	Customer Sequences
1	(_, e, g)(b)(h)(f)(c)(b, f)
4	(_, g)(b, f, h)(b, f)

We summarize the three strategies that are used in the related works as follows.

1. **Candidate sequence pruning**: This strategy prunes away the candidate sequences that cannot be frequent as early as possible. All the GSP, SPADE, SPAM, and PrefixSpan algorithms adopt this strategy based on the anti-monotone property. This strategy contributes to the reduction of processing costs and storage overheads for support counting.
2. **Database partitioning**: This strategy partitions the database into groups such that each group can fit into the main memory. The PrefixSpan algorithm adopts this strategy by projecting the database according to the prefixes of customer sequences, while the SPADE and SPAM algorithms implicitly partition the database based on the candidate sequences. This strategy eliminates the unnecessary decompositions of customer sequences while adding the extra costs for partitioning the database.
3. **Customer sequence reducing**: This strategy reduces the customer sequences as much as possible. The PrefixSpan algorithm adopts this strategy in its projection scheme. For example, the fourth customer sequence <(f)(a, g)(b, f, h)(b, f)> in Table 1 is reduced to <(_, g)(b, f, h)(b, f)> in Table 2. This strategy contributes to the reduction of processing costs for decomposing the customer sequences.

1.2. Overview of our approach

As opposed to the above strategies, in this paper, we propose the fourth strategy, named *DIrect Sequence Comparison* (abbreviated as *DISC*) to reduce the costs for support counting and the decomposition of customer sequences. The goal of this strategy is to recognize the frequent sequences for a specific length k without having to compute the support counts of the non-frequent sequences. Furthermore, we propose an algorithm called

DISC-all that combines all the four strategies to efficiently find frequent sequences in large databases.

In our approach, we define the order of two sequences having the same length. Given two sequences, we examine their items from left to right and compare the leftmost distinct items by the alphabetical order. For example, <(a)(b)(h)> is smaller than <(a)(c)(f)> because in the 2nd transactions, b is smaller than c. It cannot distinguish the cases where the items contained in both sequences are the same while their distributions in the sequences are different, e.g., <(a, b)(c)> and <(a)(b, c)>. Therefore, before finding the leftmost distinct items, we examine the common prefixes of two sequences from left to right and identify the leftmost items located in different transactions. A sequence is smaller if its leftmost item found is located in an earlier transaction, e.g., <(a, b)(c)> is smaller than <(a)(b, c)>.

The DISC strategy then iteratively checks whether a k-sequence is frequent from the minimum k-sequence. For this reason, we find the *k-minimum subsequence* in each customer sequence and sort customer sequences by the order of their associated k-minimum subsequence. After customer sequences are sorted, we get a k-sorted database. For example, Table 3 is the 3-sorted database of Table 1. In a k-sorted database, we focus on two positions, i.e., the first position and the δ-th position where δ is the minimum support count. The k-minimum subsequence at the first position is denoted by α_1 and the k-minimum subsequence at the δ-th position is denoted by α_δ.

The main idea of the DISC strategy is to compare α_1 with α_δ, to decide whether α_1 is a frequent k-sequence or not. If α_1 is equal to α_δ, then all the k-minimum sequences in the k-sorted database between α_1 and α_δ must all be equal to α_1. Therefore, α_1 must be frequent. In this case, the next potential frequent k-sequence must be greater than α_δ by the order we defined. Therefore, for each customer sequence whose associated k-minimum subsequence is equal to α_1, we find the minimum k-sequence greater than α_δ (called the *conditional k-minimum sequence*) and update the position of each customer sequences in the k-sorted database. The DISC strategy then repeats its process to find the next frequent k-sequence. If α_1 is not equal to α_δ, there is not enough customer sequence to support α_1, and therefore α_1 is not frequent. In this case, the next potential frequent k-sequence must be greater than or equal to α_δ. Therefore, for each customer sequence whose associated k-minimum subsequence is smaller than α_δ, we find the conditional k-minimum sequence which is greater than or equal to α_δ and update the position of each customer sequences in the k-sorted database. The DISC strategy then repeats its process to find the next frequent k-sequence.

The DISC strategy uses a k-sorted database to find all the frequent k-sequences and skips most non-frequent k-sequences by checking only the conditional k-minimum subsequences. In this way, all the frequent k-sequences can be found without computing the support counts of non-frequent ones.

Table 3: The 3-sorted database of Table 1

CID	3-minimum Subsequences	Customer Sequences
1	(a)(b)(b)	(a, e, g)(b)(h)(f)(c)(b, f)
4	(a)(b)(b)	(f)(a, g)(b, f, h)(b, f)
2	(b)(d)(e)	(b)(d, f)(e)
3	(b, f, g)	(b, f, g)

In the following, two examples are used to show the main advantages of the DISC strategy.

Example 1.1. From Table 3, it can be seen that the customer sequences with the same k-minimum subsequences are located in the continuous positions of the k-sorted database. Moreover, the minimum of all the k-minimum subsequences must be located in the first N positions if its support count is exactly N. For instance, the sequence <(a)(b)(b)> is the minimum and its support count is equal to 2. Therefore, we can determine whether α_1 is frequent by simply comparing α_1 and α_δ.

Example 1.2. Considering Table 3, if δ is 3, <(a)(b)(b)> (α_1) is not frequent. From this, we also know that the 3-sequences smaller than <(b)(d)(e)> (α_δ), e.g. <(a)(b)(c)> and <(a)(b, f)> cannot be frequent. Therefore, for CID 1 and 4, we generate its conditional 3-minimum sequences, which should be larger than or equal to <(b)(d)(e)>. As a result, we have another 3-sorted database with new α_1 and α_δ as shown in Table 4. In this way, all the non-frequent 3-sequences smaller than <(b)(d)(e)> are skipped.

Table 4: Table 3 after re-sorting CID 1 and 4

CID	3-minimum Subsequences	Customer Sequences
2	(b)(d)(e)	(b)(d, f)(e)
4	(b, f)(b)	(f)(a, g)(b, f, h)(b, f)
3	(b, f, g)	(b, f, g)
1	(b)(f)(b)	(a, e, g)(b)(h)(f)(c)(b, f)

As a result, the DISC strategy has the following advantages:

1. Only the support counts of frequent sequences are required to be computed. That is, no candidate sequence is generated.
2. As many of the non-frequent sequences are skipped, the costs for decomposing customer sequences are implicitly reduced.
3. The frequent k-sequences can be directly discovered without following the bottom-up approach.

In this paper, we propose an algorithm called *DISC-all* that combines all the four strategies to efficiently find frequent sequences in large databases. Compared with the previous work, our algorithm has the same advantages that come from the three strategies and more from the DISC strategy. The existing algorithms and their strategies are summarized in Table 5.

The rest of this paper is organized as follows. The basic definitions and lemmas used for the DISC strategy are presented in Section 2. After that, we describe the details of the DISC-all algorithm in Section 3. In Section 4, the performance of the DISC-all algorithm is evaluated via a series of experiments and the efficiency issue is further discussed according to the observations on the experiment results. Finally, we make conclusions on this work with future works in Section 5.

Table 5: The existing algorithms and strategies

Algorithm Strategy	Candidate Sequence Pruning	Database Partitioning	Customer Sequence Reducing	DISC
GSP	√			
SPADE	√	√		
SPAM	√	√		
PrefixSpan	√	√	√	
DISC-all	√	√	√	√

2. Basic definitions and lemmas

As described above, for sorting the customer sequences, we have to provide a way to compare two sequences. Given two sequences, we first renumber the transactions in each sequence from left to right and associate each item with the corresponding number (called the *transaction number*). For instance, in <(a)(b)(c, d)(e)>, the transaction number of the five items are 1, 2, 3, 3, and 4, respectively. In this way, a sequence can be represented in the form of $<A_1A_2\ldots A_n>$ where each A_i is associated with two values, i.e. an item and a transaction number (denoted as $A_i.item$ and $A_i.no$). Based on the alphabetic order, we define a specific position at two sequences that can distinguish them as follows:

Definition 2.1 Differential point

Given two sequences $A=<A_1A_2\ldots A_n>$ and $B=<B_1B_2\ldots B_m>$, the j-th position in both sequences is the *differential point* if both the following conditions hold:

(a) $\forall i<j$, *(*$A_i.item=B_i.item$*) and (*$A_i.no=B_i.no$*)*

(b) *(*$A_j.item\neq B_j.item$*) and (*$A_j.no\neq B_j.no$*)*

Condition (a) stands for the common prefixes of the two sequences, while condition (b) refers to the first position in both sequences that are different. Without loss of generality, when a sequence is the prefix of another sequence, we can add a special item that is smaller than any other item to the end of the shorter sequence as the differential point. In this way, given two sequences, at most one differential point can be found to determine their order as follows:

Definition 2.2 Comparative order

Given two sequences $A=<A_1A_2\ldots A_n>$ and $B=<B_1B_2\ldots B_m>$, $A=B$ if no differential point can be found. Otherwise, let the differential point be j and $A<$B if one of the following conditions holds:

(a) $A_j.item<B_j.item$

(b) *(*$A_j.item=B_j.item$*) and (*$A_j.no<B_j.no$*)*

Finally, $A>B$ if both the above conditions do not hold.

Example 2.1. Given two sequences A=<(a, c, d)(d, b)> and B=<(a, d, e)(a)>, the differential point is the second position because A_2.item is smaller than B_2.item. Given another sequence C=<(a, c)(d, a)>, the differential point of A and C is the third position because A_3.no is smaller than B_3.no. By Definition 2.2(a), we have A<B. Moreover, we have A<C according to Definition 2.2(b).

Based on the comparative order, we define the *k-minimum subsequence* of a customer sequence and the *k-minimum order* that determine the order of sequences based on their k-minimum subsequences as follows:

Definition 2.3 K-minimum subsequence

A sequence μ_k is the *k-minimum subsequence* of a sequence A if both the following conditions hold:

(a) μ_k *is a k-sequence and a subsequence of A*

(b) $\forall$ *k-sequence* μ, *which is a subsequence of A,* $\mu_k \leq \mu$

Definition 2.4 K-minimum order

Let the signs $=_k$, $<_k$, and $>_k$ be the comparative operators for the denotation of k-minimum order. Given two sequences A and B whose k-minimum subsequences are μ_k and ν_k respectively, we define the *k-minimum order* of A and B as follows:

(a) $A=_kB$ *if* $\mu_k=\nu_k$

(b) $A<_kB$ *if* $\mu_k<\nu_k$

(c) $A>_kB$ *if* $\mu_k>\nu_k$

Example 2.2. Considering sequence A in Example 2.1, by Definition 2.3, we have 1-minimum sequence <(a)>, 2-minimum sequence <(a)(b)>, 3-minimum sequence <(a, c)(b)>, 4-minimum sequence <(a, c, d)(b)>, and 5-minimum sequence <(a, c, d)(d, b)>. Moreover, the 3-minimum sequences of B and C are <(a, d)(a)> and <(a, c)(a)>, respectively. By Definition 2.4, we have the 3-minimum order $C<_3A<_3B$ and the 2-minimum order $C=_2B<_2A$.

As Section 1.2 depicts, the customer sequences can be sorted into the k-sorted database by the k-minimum order. The k-minimum subsequence at the first position of the k-sorted database is called the *candidate k-sequence* and denoted by α_1. Given a minimum support count δ, the k-minimum subsequence at the δ-th position of the k-sorted database is called the *candidate k-sequence* and denoted by α_δ. As a result, we have the following lemmas to show the correctness of the DISC strategy.

Lemma 2.1 Frequent k-sequences

α_1 *is frequent if* $\alpha_1=\alpha_\delta$.

Proof: Because the database is sorted according to the k-minimum subsequences, α_1 must repeatedly appear at the first δ positions when α_1 equals α_δ. In other words, the first δ customer sequences in the k-sorted database take α_1 as their k-minimum subsequences. In this case, the support of α_1 must be at least δ and therefore α_1 is frequent.

Lemma 2.2 Non-frequent k-sequences

$\forall$k-sequence α, α is non-frequent if $\alpha_1 \leq \alpha < \alpha_\delta$.

Proof: Because the database is sorted according to the k-minimum subsequences, it is not possible for α to appear below the δ-th position when α is smaller than α_δ. Moreover, all the k-minimum subsequences that appear below the δ-th position are larger than α_δ. In other words, only the customer sequences above the δ-th position may contain α. In this case, the support of α must be at most δ-1 and therefore α cannot be frequent.

Note that Lemma 2.1 and Lemma 2.2 were applied in Example 1.1 and Example 1.2, respectively.

After the comparison between α_1 and α_δ is done, we have to generate the conditional k-minimum subsequence of each customer sequence whose k-minimum subsequence is equal to α_1. As the two cases discussed in Section 1.2, we define the *conditional k-minimum subsequence* as follows:

Definition 2.5 Conditional k-minimum subsequence

A sequence μ_k is the *conditional k-minimum subsequence* of A if both the following conditions hold:

(a) *μ_k is a k-sequence and a subsequence of A*

(b) *$\forall\mu$, which is a k-sequence and a subsequence of A, if $\alpha_1 = \alpha_\delta$, $\alpha_\delta < \mu_k \leq \mu$; otherwise, $\alpha_\delta \leq \mu_k \leq \mu$*

Note that the generation of conditional k-minimum subsequences has been illustrated in Section 1.2.

3. The DISC-all algorithm

In this section, we present the *DISC-all* algorithm that combines four mining strategies to efficiently find all the frequent sequences. The DISC-all algorithm is mainly based on the database partitioning and DISC strategies. At first, we scan the database to divide the customer sequences into partitions by their minimum 1-sequences such that the partitions can be ordered according to the minimum 1-sequences they have. These partitions are called the *first-level partitions*. During the partitioning, we can find all the frequent 1-sequences. For each first-level partition, we regard its minimum 1-sequence as the prefix in the PrefixSpan algorithm to find the frequent 2-sequences in it. For each customer sequence in the first-level partition, we remove non-frequent 1-sequences and non-frequent 2-sequences to generate a shorter customer sequence. Note that the minimum 1-sequence of this partition must not be removed. After that, we divide the reduced customer sequences into partitions by its 2-minimum sequence. These partitions are called the *second-level partitions*. For each second-level partition, we can also find the frequent 3-sequences in it. The above scheme is named *multi-level partitioning*, where the number of levels should be adaptive and depends on the tradeoff between overheads and profits brought from partitioning. In this paper, we adopt the two-level partitioning scheme for the ease of presentation.

For each second-level partition, the DISC-all algorithm iteratively generates the k-sorted databases where k is larger than 3. Given a k-sorted database, the frequent k-sequences can be generated by the DISC strategy. After that, each customer sequence of the second-level partition is reassigned to another second-level partition by the next 2-minimum sequence. When all the second-level partitions under a first-level partition have been processed, each customer sequence of the first-level partition is also reassigned to another first-level partition by the next minimum 1-sequence. Note that the other two strategies are also incorporated into the DISC-all algorithm. For example, the removal of non-frequent 2-sequences before generating the second-level partitions is based on both the strategies of candidate sequence pruning and customer sequence reducing. For described above, the DISC-all algorithm uses a *multi-level partitioning* scheme to find the frequent sequences with lengths smaller than 4 and adopt the DISC strategy to find the other frequent sequences. As the framework shown in Figure 1 indicates, the generation of partitions is executed in the breadth-first order, i.e. all the first-level partitions and then all the second-level partitions. However, finding frequent sequences from these partitions is in the depth-first order, i.e. the first-level partition 1, the second-level partition 1, the second-level partition 2, and so on. In the following sections, we will introduce the proposed techniques used in these two components, i.e. multi-level partitioning and direct sequence comparison, respectively.

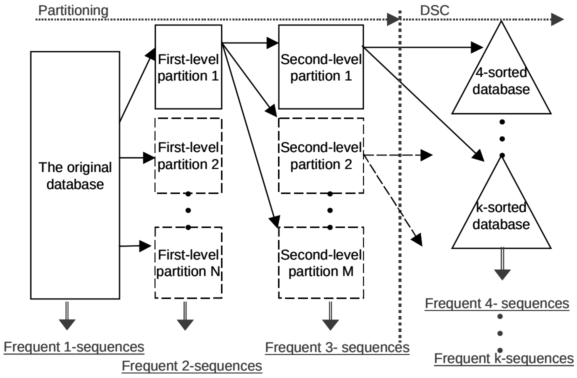

Figure 1: The framework of the DISC-all algorithm

3.1. Multi-level partitioning

The input of the DISC-all algorithm includes a database of customer sequences and δ, while the output is the set of all the frequent sequences. The DISC-all algorithm based on two-level partitioning is shown in Figure 2. The first step finds all the frequent 1-sequences and generates the first-level partitions by scanning the original database once. To find all the frequent 1-sequences, we simply use an array to accumulate the count of each 1-sequence during the database scan. In the meantime, for each customer sequence, we also find the minimum 1-sequence and keep the leftmost position that it appears in the customer sequence. This position is called the *minimum point*. Finally, we classify each customer sequence into a partition according to its minimum 1-sequence. We call the partition with the minimum 1-sequence λ the *<(λ)>-partition*.

```
Input: A sequence database DB, δ
Output: A set of all the frequent sequences
1. Scan DB once to do:
  (a) Find all the frequent 1-sequences
  (b) Generate first-level partitions by minimum 1-sequences
2. For each first-level partition FP to do:
  2.1 If the minimum 1-sequence is frequent,
    2.1.1 Find all the frequent 2-sequences in FP
    2.1.2 Scan FP once again to do:
      (a) Removing non-frequent 1-sequences/2-sequences
      (b) Generate the second-level partitions under FP
    2.1.3 For each second-level partition SP to do:
      2.1.3.1 Find all the frequent 3-sequences in SP
      2.1.3.2 Let k=4, Repeat
        (a) Generated the k-sorted database of SP
        (b) Find all the frequent k-sequences
        (c) Let k=k+1
        Until (size of SP < δ) or (no frequent (k-1)-sequence)
      2.1.3.3 Reassign customer sequences from SP to others
  2.2 Reassign customer sequences from FP to others
```

Figure 2: The DISC-all algorithm

The second step of the DISC-all algorithm processes each of the first-level partitions in the alphabetic order of the minimum 1-sequences. Given a <(λ)>-partition, if λ is frequent, Step 2.1 will discover all the frequent sequences that contain λ as the first item. Therefore, only the items to the right of the minimum point have to be processed. The subsequences of a customer sequence may contribute to the support counts of different frequent sequences in more than one first-level partition. Therefore, in Step 2.2, for each customer sequence in the first-level partition that has been processed, we further find the next minimum 1-sequence in it and reclassify it into the other first-level partitions.

Example 3.1. Consider the example database in Table 6 and let δ be 3. The first-level partition of each customer sequence is shown in the third column of Table 6 and all the 1-sequences except <(d)> are frequent. For instance, the first seven customer sequences belong to <(a)>-partition because their minimum 1-sequences are a. As a result, initially there are four partitions with disjoint sets of customer sequences.

In the second step, <(a)>-partition will be processed first to find all the frequent sequences that contain a as the first item, e.g. <(a, e)> and <(a)(g, h)>. After that, we find the next minimum 1-sequence in each of the seven customer sequences and then reclassify them into the other first-level partitions, respectively. For instance, CID 1 and 2 are respectively reassigned into <(c)>-partition and <(b)>-partition as shown in the rightmost column of Table 6. Note that the reassignments of customer sequences may lead to the creation of a new partition (e.g. <(c)>-partition) and the removal of a customer sequence when its minimum point is at its end (e.g. CID 5).

Table 6: Database and first-level partitions

CID	Customer Sequences	Initial Partitions	After processing <(a)>-partition
1	(a, d)(d)(a, g, h)(c)	<(a)>-partition	<(c)>-partition
2	(b)(a)(f)(a, c, e, g)	<(a)>-partition	<(b)>-partition
3	(a, f, g)(a, e, g, h)(c, g, h)	<(a)>-partition	<(c)>-partition
4	(f)(a, c, f)(a, c, e, g, h)	<(a)>-partition	<(c)>-partition
5	(a, g)	<(a)>-partition	Removed
6	(a, f)(a, e, g, h)	<(a)>-partition	<(e)>-partition
7	(a, b, g)(a, e, g)(g, h)	<(a)>-partition	<(b)>-partition
8	(b, f)(b, e)(e, f, h)	<(b)>-partition	<(b)>-partition
9	(d, f)(d, f, g, h)	<(d)>-partition	<(d)>-partition
10	(b, f, g)(c, e, h)	<(b)>-partition	<(b)>-partition
11	(e, g)(f)(e, f)	<(e)>-partition	<(e)>-partition

In Step 2.1, given a <(λ)>-partition where λ is frequent, we scan the partition once to discover all the frequent 2-seuqneces via a mechanism called the *counting array*. In the <(λ)>-partition, item λ is regarded as the common prefix of all the frequent 2-sequences to be found from it. Therefore, the counting array only reserves two entries for each item x to respectively keep the support counts of the 2-sequences in the forms of <(λ)(x)> and <(λx)>. Moreover, each entry is associated with two values, i.e. the support count and the last CID when the support count is updated. The CID information can avoid counting the repetitions of a 2-sequence in the same customer sequence. In this way, all the support counts of 2-sequences can be correctly computed in only one scan.

After that, we reduce the length of each customer sequence by removing all the non-frequent 1-sequences and non-frequent 2-sequences. We keep the set of reduced customer sequences as another copy. Given a customer sequence in the <(λ)>-partition, the following two conditions are used to determine whether an item to the right of the minimum point can be removed or not:

1. The transaction having x contains λ.
2. The minimum point is to the left of the transaction having x.

When the condition 1 does not hold, item x can be removed if <(λ)(x)> is not frequent. When the condition 1 holds but condition 2 does not hold, item x can be removed if <(λx)> is not frequent. If both the conditions hold, item x can be removed only if both <(λ)(x)> and <(λx)> are not frequent. All the occurrences of λ cannot be removed because they may be involved in the support counting for the frequent sequences with larger lengths. In the meantime, the reduced customer sequences are classified into the second-level partitions according to their 2-minimum sequences.

Example 3.2. Take the <(a)>-partition in Table 6 as an example. Because <(a)> is frequent, we use the counting array to accumulate the support counts of 2-sequences during one scan. Figure 3 shows the results of the counting array, where (x) and (_x) refer to the two forms <(a)(x)> and <(ax)>, respectively. Only <(a)(b)>, <(a)(d)>, <(a)(f)>, <(ab)>, <(ac)>, and <(ad)> are not frequent. Finally, the customer sequences can be reduced by removing the non-frequent sequences except item a. As shown in Table 7, item c in CID 2 is not removed because <(a)(c)> is frequent. Moreover, the customer sequences with lengths smaller than 3, e.g. CID 5, is also removed from the reduced partition.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
Support count	6	0	4	1	5	1	6	5
Last CID	7	0	4	1	7	2	7	7
	(_a)	(_b)	(_c)	(_d)	(_e)	(_f)	(_g)	(_h)
Support count	0	1	2	1	5	3	7	5
Last CID	0	7	4	1	7	6	7	7

Figure 3: The count array of <(a)>-partition

In Step 2.1.3, with the counting array, we also find the frequent 3-sequences in each second-level partition in one scan. After that, we apply the DISC strategy to find the frequent k-sequences iteratively. Finally, each customer sequence is reassigned to another second-level partition by the next 2-minimum sequence. In the next section, we will present the core of the DISC-all algorithm, i.e. the techniques for direct sequence comparison.

Table 7: <(a)>-partition with reduced sequences

CID	Customer Sequences
1	(a)(a, g, h)(c)
2	(b)(a)(a, c, e, g)
3	(a, f, g)(a, e, g, h)(c, g, h)
4	(f)(a, f)(a, c, e, g, h)
6	(a, f)(a, e, g, h)
7	(a, g)(a, e, g)(g, h)

3.2. Direct sequence comparison

To find all the frequent sequences in a second-level partition, we adopt the bottom-up approach to start at the discovery of frequent 4-sequences and repeat it until no more frequent sequences can be found as indicated by Step 2.1.3.2 in Figure 2. The *frequent k-sequence discovery* procedure consists of two stages as shown in Figure 4. The first stage finds the k-minimum subsequence of each customer sequence to construct the k-sorted database. The second stage repeats three steps, i.e. direct sequence comparison, generation of conditional k-minimum subsequences, and the re-sorting of k-sorted database. Note that the customer sequences without conditional k-minimum subsequences will be removed from the k-sorted database. Therefore, all the frequent k-sequences in this partition are found when the size of the k-sorted database is smaller than δ.

```
Input: <$\lambda_1\lambda_2$>-partition, $\delta$, k
Output: All the frequent k-sequences with prefix <$\lambda_1\lambda_2$>
1. Scan the <$\lambda_1\lambda_2$>-partition once to do:
  (a) Generate the k-minimum subsequence for each
      customer sequence
  (b) Construct the k-sorted database SD
2. While (the size of SD $\geq \delta$) do:
  2.1 Check candidate k-sequence by condition k-sequence
  2.2 Generate the conditional k-minimum subsequences
  2.3 Resort SD by the conditional k-minimum
subsequences
```

Figure 4: Frequent k-sequence discovery

We propose an algorithm named *Apriori-KMS* that can generate the k-minimum subsequences in Step 1(a) and a similar one called *Apriori-CKMS* that can generate the conditional k-minimum subsequences in Step 2.2. Step 2.1 is based on the two lemmas described in Section 2, indicating that the candidate k-sequence is frequent if it is the same as the condition k-sequence. In this way, we can determine whether the candidate k-sequence is frequent by direct sequence comparison. The sorting methods required in Step 1(b) and 2.3 is based on a mechanism called the *locative AVL-Tree*, which can provide efficient sorting and retrieval. In addition, we also design the *bi-level* version of the DISC-all algorithm, which discovers all the frequent k-sequences and frequent (k+1)-sequences in only one scan of the k-sorted database.

For the ease of presentation, we call the prefix of a sequence with length k the *k-prefix*. For instance, the 3-prefix of <(a)(a, g, h)(c)> is <(a)(a, g)>. According to the anti-monotone property, the k-minimum subsequence cannot be frequent if its (k-1)-prefix is not frequent. Therefore, we utilize the frequent (k-1)-sequences for generating the k-minimum subsequences to skip the k-sequence whose (k-1)-prefix is not frequent. When a k-sorted database is processed, all the frequent (k-1)-sequences are linked together in the ascending order and called the *(k-1)-sorted list*. The Apriori-KMS algorithm is shown in Figure 5.

At first, the Apriori-KMS algorithm selects the frequent (k-1)-sequences from the (k-1)-sorted list one by one, where the smallest one is chosen first. Let the chosen frequent (k-1)-sequence be F. In Step 4, the customer sequence S is scanned to find the leftmost match of F and record the position on S that matches the last item of F, called the *matching point*. When the matching point is not at the end of S, the minimum of the items to the right of the matching point is found and added to the end of F to form the k-minimum subsequence of S. If no match is found, the next frequent (k-1)-sequence is selected for another iteration. After the k-minimum subsequence is found, each customer sequence is associated with a pointer named *apriori pointer* that refers to a node of the (k-1)-sorted list, whose frequent (k-1)-sequence is the (k-1)-prefix of the k-minimum subsequence. This piece of information will be used in the Apriori-CKMS algorithm.

```
Input: A customer sequence S
Output: The k-minimum subsequence of S
1. P=the first node of the (k-1)-sorted list
2. While ($P \neq NULL$) Do {
3.    F=the frequent (k-1)-sequence of P
4.    Find the leftmost match of F on S
5.    M=the matching position of $F_{k-1}$ on S
6.    If ($F \subset S$) and ($M \neq$ End of S) {
7.          Z=minimum of the items to the right of $S_M$
8.          Return the concatenated sequence <FZ>}
9.    Else  P=the next node in the (k-1)-sorted list}
10. Return with no result
```

Figure 5: The Apriori-KMS Algorithm

Example 3.3. Let δ be 3. Take the <(a)(a)>-partition and its 3-sorted list in Table 8 as an example. At the beginning, we pick up <(a)(a, e)> to scan CID 1 but no match is found. After that, we scan CID 1 again and find a match of <(a)(a, g)>. The matching point is 3 and therefore item c is selected to generate the 4-minimum sequence <(a)(a, g)(c)>. A complete 4-sorted database is shown in Table 9, where CID 1 is associated with the apriori pointer referring to the frequent 3-sequence <(a)(a, g)>.

Table 8: <(a)(a)>-partition and its 3-sorted list

CID	Customer Sequences
1	(a)(a, g, h)(c)
2	(b)(a)(a, c, e, g)
3	(a, f, g)(a, e, g, h)(c, g, h)
4	(f)(a, f)(a, c, e, g, h)
6	(a, f)(a, e, g, h)
7	(a, g)(a, e, g)(g, h)

The 3-sorted List	
No	Frequent 3-sequences
1	(a)(a, e)
2	(a)(a, g)
3	(a)(a, h)

Table 9: 4-sorted database of <(a)(a)>-partition

CID	4-minimum Subsequences	Customer Sequences	Apriori Pointer
3	(a)(a, e)(c)	(a, f, g)(a, e, g, h)(c, g, h)	1
2	(a)(a, e, g)	(b)(a)(a, c, e, g)	1
4	(a)(a, e, g)	(f)(a, f)(a, c, e, g, h)	1
6	(a)(a, e, g)	(a, f)(a, e, g, h)	1
7	(a)(a, e, g)	(a, g)(a, e, g)(g, h)	1
1	(a)(a, g)(c)	(a)(a, g, h)(c)	2

Given a customer sequence S and the condition k-sequence α_δ, the goal of the Apriori-CKMS algorithm is to efficiently find the conditional k-minimum subsequence from S. By Definition 2.5, if the candidate k-minimum subsequence is frequent, the conditional k-minimum subsequence has to be greater than α_δ. Otherwise, the conditional k-minimum subsequence should be greater than or equal to α_δ. Therefore, the Apriori-CKMS algorithm needs a parameter Ω to indicate whether the conditional k-minimum subsequence can be equal to α_δ or not. Based on the apriori pointer, the Apriori-CKMS algorithm as shown in Figure 6 skips the k-sequence whose (k-1)-prefix is not frequent.

```
Input:  customer sequence S and its apriori pointer P,
        condition k-sequence $\alpha_\delta$, operation indicator Ω
Output: The conditional k-minimum subsequence of S
         under the constraints $\alpha_\delta$ and Ω
1. X=(k-1)-prefix of $\alpha_\delta$, Y=the last item of $\alpha_\delta$,
2. If (P=NULL)   Return with no result
3. F=the frequent (k-1)-sequence of P
4. While (F<X) Do {
5.    P=the next node in the (k-1)-sorted list
6.    If (P=NULL)      Return with no result
7.    F=the frequent (k-1)-sequence of P}
8. While ($P \neq NULL$) {
9.    F=the frequent (k-1)-sequence pointed to by P
10.   Find the leftmost match of F on S
11.   M=the matching position of $F_{k-1}$ on S
12.   If ($F \subset S$) and ($M \neq$ End of S) {
13.     If ($F \neq X$) Z=minimum of the items to the right of $S_M$
14.     Else Z=minimum of $S_i$, $\forall S_i$, ($M < i$) and ($S_i$ Ω Y)
15.     If (Z is found)   Return <FZ>}
16.   P=the next node in the (k-1)-sorted list}
17. Return with no result
```

Figure 6: The Apriori-CKMS algorithm

The difference between this algorithm and the Apriori-KMS algorithm is that the conditional k-minimum subsequence α must satisfy the constraint "α Ω α_δ", where Ω is either '>' or '≥'. To meet the requirement, from Steps 4~7, we select the smallest frequent (k-1)-sequence from the ones in the (k-1)-sorted list, which are larger than or equal to the (k-1)-prefix of α_δ. Note that the apriori pointer associated with each customer sequence can speed up this selection process. Let the chosen frequent (k-1)-sequence be F. After that, we follow the same steps of the Apriori-KMS algorithm to find the matching point. When the matching point is not at the end of S, the following cases are considered:

1. When F does not equal the (k-1)-prefix of α_δ, we follow the same steps of the Apriori-KMS algorithm to compose the conditional k-minimum subsequence.
2. Otherwise, the constraint is required to be checked as we search the minimum of the items to the right of the matching point. If such a minimum does not exist, the next frequent (k-1)-sequence is selected for the following iteration.

Example 3.4. From Table 9, by Lemma 2.2, the <(a)(a, e)(c)> is not frequent. By Definition 2.5, the constraint used in the Apriori-CKMS algorithm includes the condition 4-sequence <(a)(a, e, g)> and an operation indicator '≥'. Moreover, the apriori pointer of CID 3 refers to the frequent 3-sequence <(a)(a, e)>, which equals the 3-prefix of condition 4-sequence. Therefore, Steps 4~7 can be skipped. In Step 10, the matching point is 5 and the minimum of the items that satisfy the constraint is item g in the second transaction. In this way, the conditional k-minimum subsequence <(a)(a, e, g)> is obtained. The 4-sorted database after re-sorting CID 3 is shown in Table 10.

Table 10: Table 9 after re-sorting CID 3

CID	4-minimum Subsequences	Customer Sequences	Apriori Pointer
2	(a)(a, e, g)	(b)(a)(a, c, e, g)	1
4	(a)(a, e, g)	(f)(a, f)(a, c, e, g, h)	1
6	(a)(a, e, g)	(a, f)(a, e, g, h)	1
7	(a)(a, e, g)	(a, g)(a, e, g)(g, h)	1
3	(a)(a, e, g)	(a, f, g)(a, e, g, h)(c, g, h)	1
1	(a)(a, g)(c)	(a)(a, g, h)(c)	2

For the efficient construction and retrieval of the k-sorted database, we propose *the locative AVL-tree* such that the condition k-sequence can be quickly located. The locative AVL tree focuses on two issues, i.e. how to find the condition k-sequence by the access key and how to maintain the access key. For the first issue, we propose an algorithm that can quickly find the node corresponding to δ, indicating the location of condition k-sequence. The details can be found in [4]. To deal with the second issue, we modify the adjustments for the AVL-tree balance, which can be found in [14].

In a locative AVL-tree, when the candidate k-sequence is found frequent, all the customer sequences contained in the corresponding node can be regarded as a *virtual partition*. From this virtual partition, each frequent (k+1)-sequence whose k-prefix is the candidate k-sequence can be derived. Therefore, we employ the counting-array similar to the one used in Section 3.1 to compute the support count of each (k+1)-sequence whose k-prefix is the candidate k-sequence. Based on the anti-monotone property, all the frequent (k+1)-sequences can be discovered from these virtual partitions. Therefore, we can find the frequent sequences with lengths k and k+1 via only one call to the frequent k-sequence discovery procedure. This technique is named *bi-level* and used as the version of our algorithm for experiments.

Example 3.5. Following Example 3.3, by Lemma 2.1, <(a)(a, e, g)> is a frequent 4-sequence. Moreover, CID 2, 3 and 4 constitute a virtual partition that can be used to discover the frequent (k+1)-sequence whose k-prefix is <(a)(a, e, g)>. During running the Apriori-CKMS algorithm to find the conditional 4-minimum sequences, a counting-array is also used to accumulate the support count of each 5-sequence whose 4-prefix is <(a)(a, e, g)>. Figure 7 shows the results after three customer sequences have been processed, indicating that <(a)(a, e, g, h)> is the only one frequent 5-sequence with the 4-prefix <(a)(a, e, g)>.

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
Support count	0	0	1	0	0	0	1	1
Last CID	0	0	3	0	0	0	3	3
	(_a)	(_b)	(_c)	(_d)	(_e)	(_f)	(_g)	(_h)
Support count	0	0	0	0	0	0	0	3
Last CID	0	0	0	0	0	0	0	3

Figure 7: Counting array for the bi-level version

4. Performance Evaluation

The experiments are made upon the Intel Pentium 4 CPU 2.8GHz with 512 MB main memory and Microsoft Windows XP Professional. The databases used in our experiments are synthesized via the IBM data generator [1] with the version dated July 22, 1997. In parameter setting, we adopt most of the default values for the command options provided by the data generator, except the followings:

Table 11: Parameter setting of self-tuned options

Command Option	Description	Value
Ncust	Number of customers	50K~500K
Slen	Average number of transactions per customer	10
Tlen	Average number of items per transaction	2.5
nitems	Number of different items	1K
seq.patlen	Average length of maximal pattern	4

4.1. Performance evaluation of DISC-all

We compare the DISC-all algorithm with the PrefixSpan algorithm [9] in efficiency. In addition to the basic version of the PrefixSpan algorithm, we also consider the version based on *pseudo-projection* named *Pseudo* in the comparisons. The Pseudo algorithm employs a mechanism to link together all the customer sequences in a projection database. This mechanism can reduce the costs on projecting databases when the projected database can fit into the main memory. Note that we adopt the bi-level version of our algorithm in all the experiments.

Based on the parameter setting in Table 11, we generate a series of databases whose sizes range from 50K to 500K. The parameters except the database size are the same as the setting used in the IBM data generator. Figure 8 shows the experimental results under different numbers of customer sequences. The DISC-all algorithm outperforms the others for all these databases even when the minimum support threshold is set to 0.0025. Moreover, the improvement of the DISC-all algorithm enlarges as the database size increases. The reason why the PrefixSpan algorithm does not adapt to large databases is because the number of projections increases as the growth of database size. By contrast, the DISC-all algorithm can skip more non-frequent sequences during direct sequence comparison because δ increases as the growth of database size.

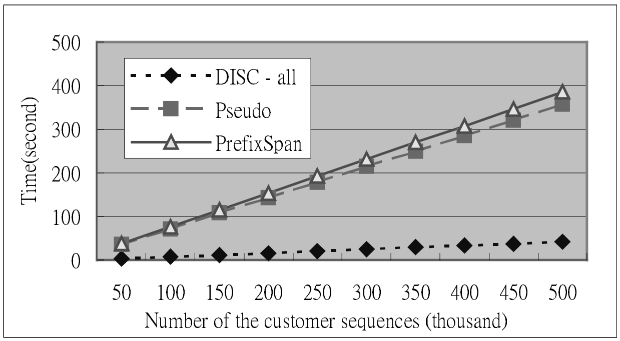

Figure 8: Comparisons on database sizes

In the second experiment, based on the parameter setting in [8] where the slen, tlen and seq.patlen are all set to 8, we generate a database with 10K customer sequences. Figure 9 shows the experimental results under different settings of δ's, where the minimum support threshold is the proportion of δ to the database size.

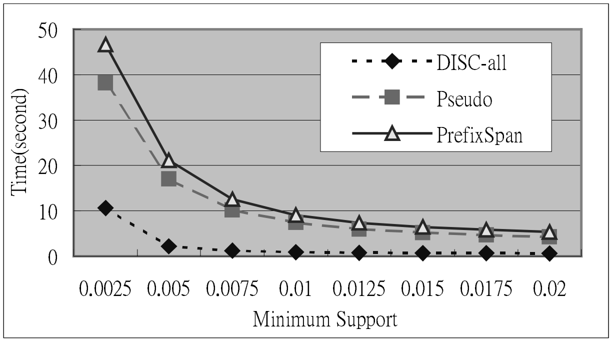

Figure 9: Comparisons on different δ's

Obviously, the DISC-all algorithm is the best of the three because it always spends the least amount of processing time when the minimum support threshold ranging from 0.02 to 0.0025. When the minimum support threshold is set to 0.0025, we observe that there are more than 100K frequent sequences and the length of the maximal frequent sequences is at least 14.

4.2. Discussions on multi-level partitioning

The multi-level partitioning scheme is good at reducing the number of customer sequences such that the unnecessary decompositions of customer sequences can be eliminated. Given a partition Q, let N_Q be the number of its child partitions. To evaluate the effects of partitioning, for each partition, we estimate the average ratio of the size of its child partition to its partition size, which is called the *non-reduction rate* and denoted by *NRR,* as follow:

$$\mathrm{NRR}_Q = \frac{1}{N_Q} \sum_{\text{p is a child partition of Q}} \frac{\mathrm{Size}_p}{\mathrm{Size}_Q}, \cdots (2)$$

Given a <λ>-partition where λ is k-sequence, the simplest way to compute the NRR of this partition is to consider the support count of each frequent (k+1)-sequence discovered in this partition as the size of its child partition. We define the *average NRR* as the average of the NRR's for all the partitions in the same level under the multi-level partitioning scheme. Table 12 shows the average NRR of each level under different minimum support thresholds and the database size 10K. Obviously, each of the first-level partitions is much smaller than the original database according to its average NRR. However, the partitions at different levels may have various effects on the NRR. For example, when the minimum support threshold is 0.005, the average NRR of the second-level partitions is 0.64, which is much higher than the one of the first-level partitions (0.11) but much smaller than the one of the third-level partitions (0.9). Based on the multi-level partitioning scheme, the partition size is never smaller than δ. As the partitioning goes to a deeper level, the partition size is getting smaller and close to δ. Therefore, the NRR of a partition tends to become larger at the deeper level.

Table 12: Average NRR under different δ's

Average NRR δ	Original	1	2	3	4	5	6	7	8
0.02	0.0027	0.18	-	-	-	-	-	-	-
0.0175	0.0026	0.18	-	-	-	-	-	-	-
0.015	0.0025	0.16	-	-	-	-	-	-	-
0.0125	0.0024	0.15	-	-	-	-	-	-	-
0.01	0.0022	0.14	0.92	-	-	-	-	-	-
0.0075	0.002	0.12	0.9	0.98	0.98	-	-	-	-
0.005	0.0019	0.11	0.64	0.9	0.94	0.97	0.99	-	-
0.0025	0.0018	0.08	0.43	0.83	0.85	0.85	0.86	0.87	0.90

In Table 13, for the database size 10K, the DISC-all algorithm can achieve the most significant improvement when the minimum support threshold is 0.0075. By contrast with Table 12, we find that both the NRR of the original database and the average NRR of the first-level partitions are small (0.002 and 0.12). Moreover, all the average NRR's of the partitions at the other levels are large (0.9, 0.98, and 0.98). From the fact that the DISC-all algorithm statically replaces the database partitioning strategy with the DISC strategy at level 2, we conclude that the database partitioning strategy prefers the partition with a low NRR. Considering the extreme case, if the NRR of a partition is 1, all the child partitions have the same sizes as it has. Therefore, the overhead is against the benefit under the database partitioning strategy.

From the above observation, the divide between the database partitioning strategy and the DISC strategy is important to the performance of the DISC-all algorithm. Therefore, we further develop a dynamic version of the DISC-all algorithm, called the *Dynamic DISC-all* algorithm, which can adapt the divide between the database partitioning strategy and DISC strategy to the growth of NRR. Initially, this algorithm repeatedly adopted the database partitioning strategy. When the NRR of a partition becomes larger than a predefined threshold, the DISC strategy is used to find all the remaining frequent sequences in this partition. The Dynamic DISC-all algorithm is attached in the Appendix.

Table 13: The ratio of Pseudo to DISC-all

δ	Pseudo	DISC-all	Pseudo/DISC-all
0.0025	38.234	10.656	3.588026
0.005	17.015	2.203	7.723559
0.0075	**10.235**	**1.234**	**8.294165**
0.01	7.375	0.906	8.140177
0.0125	5.969	0.766	7.792428
0.015	5.234	0.703	7.445235
0.0175	4.672	0.671	6.962742
0.02	4.282	0.64	6.690625

4.3. Performance evaluation of Dynamic DISC-all

In this experiment, we adopt most of the default values provided by the data generator, except 50K customer sequences and 1000 items. Let the average number of transactions per customer sequence in the entire database be denoted as θ. Moreover, we generate a series of databases whose θ's range from 10 to 40. The minimum support threshold is set to 0.005. Table 14 shows the average NRR's of each level during processing these databases.

From Table 14, we observe that the average NRR of a partition in each level tends to decrease as the growth of θ. The reason is as follows. The growth of θ may lead to the enlargement of both the partition and its child partition, which will result in the change of the NRR of this partition. When the increase of the partition is much larger than most of the increases of its child partitions, the NRR of this partition will be decreased.

Table 14: Average NRR under different θ's

Average NRR θ	Original	1	2	3	4	5	6
10	0.0072	0.1	0.83	0.83	-	-	-
15	0.0096	0.09	0.66	0.81	0.87	0.99	-
20	0.0114	0.09	0.56	0.81	0.83	0.98	-
25	0.0129	0.1	0.26	0.75	0.85	0.85	0.82
30	0.014	0.11	0.2	0.74	0.82	0.88	0.80
35	0.0151	0.11	0.2	0.71	0.78	0.84	0.91
40	0.016	0.12	0.2	0.52	0.76	0.78	0.77

Figure 10 shows the process time of different approaches for these databases. Obviously, the Dynamic DISC-all algorithm outperforms all the others under different average numbers of transactions per customer sequence. Moreover, the DISC-all algorithm also outperforms the other two approaches at all cases except for θ 40. The reason why the DISC-all algorithm becomes worse than the Pseudo algorithm when θ is 40, can be observed as follows. As described in Sec. 4.2, the multi-level database partitioning strategy prefers the partition with a low NRR. However, in Table 14, for θ 40, the average NRR at level 2 and 3 are 0.2 and 0.52 respectively. By contrast with the dynamic version, the DISC-all algorithm statically replaces the database partitioning strategy with the DISC strategy at level 2. Therefore, it does not take full advantage of the database partitioning at the levels deeper than 2 and the performance is worsened when θ is 40.

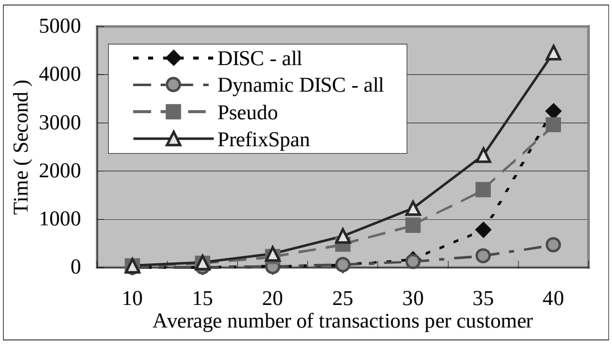

Figure 10: Comparisons on different θ's

As a summary, the database partitioning strategy is good for the partition with a small NRR, while its overhead is against the benefit when the NRR of a partition is large. In the latter case, the DISC strategy, which is not influenced by the NRR, can avoid unnecessary decompositions of customer sequences. Therefore, the Dynamic DISC-all algorithm performs much better than the DISC-all algorithm when the NRR is varied.

5. Conclusion

In this paper, we propose the DISC strategy that reduces candidate sequences without using the anti-monotone property. Moreover, we design the DISC-all algorithm that combines it with the other strategies used in the pervious work to find frequent sequences in large databases. Furthermore, we develop the Dynamic DISC-all algorithm that dynamically combines the multi-level partitioning scheme with the procedure of frequent sequence discovery. Finally, we make experiments and compare our algorithm with the PrefixSpan algorithm to reveal the usefulness of the DISC strategy and the characteristic of the database partitioning strategy. The following summarizes the main contributions of this paper:

1. We propose a new strategy for mining sequential patterns and prove its usefulness.
2. We design efficient algorithms to meet the requirements of the proposed strategy.
3. We classify the related works and summarize their strategies.
4. We design an algorithm that takes advantages of all the

strategies.

5. We analyze the partition strategy to design the dynamic version of our algorithm and achieve a much better performance.

The DISC strategy is not limited by the anti-monotone property and therefore it can be applied to many of the real world applications. The so-called weighting applications are very common and important in the real world. For example, when finding the traversal patterns in the WWW, different pages may have a variety of importance, e.g. page weights. Moreover, in DNA sequence analysis, some genes may be more important than the others in a particular disease. For both the scenarios, a pattern depends on not only the number of its occurrences but also its weight, defined by a specific application. It is challenging and interesting to apply the DISC strategy to such kinds of weighting applications.

6. Acknowledgement

This work was partially supported by the MOE Program for Promoting Academic Excellence of Universities under the grant number 89-E-FA04-1-4, and the NSC under the contract number 92-2213-E-007-094.

References

[1] R. Agrawal and R. Srikant, “Mining Sequential Patterns,” *Proceedings of IEEE Conference on Data Engineering*, pp. 3-14, 1995.

[2] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu “Sequential Pattern Mining using A Bitmap Representation,” *Proceedings of ACM SIGKDD Conference*, pp. 429-435, 2002.

[3] J. K. Bonfield and R. Staden, “ZTR: A New Format for DNA Sequence Trace Data,” *Bioinformatics*, 18(1): 3-10, 2002.

[4] D. Y. Chiu, Y. H. Wu, A. L. P. Chen, “An Efficient Algorithm for Mining Frequent Sequences by the DISC Strategy,” *Technical Report*, 2003.

[5] M. N. Garofalakis, R. Rastogi, and K. Shim, “Mining Sequential Patterns with Regular Expression Constraints,” *IEEE Transactions on Knowledge and Data Engineering*, 14(3): 530-552, 2002.

[6] J. W. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. C. Hsu, “FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining,” *Proceedings of ACM Conference on Knowledge Discovery and Data Mining*, pp. 355-359, 2000.

[7] J. L. Hsu, C. C. Liu, and A. L. P. Chen “Discovering Nontrivial Repeating Patterns in Music Data,” *IEEE Transactions on Multimedia*, 3(3): 311-325, 2001.

[8] N. Lesh, M. J. Zaki, and M. Ogihara, “Mining Features for Sequence Classification,” *Proceedings of ACM Conference on Knowledge Discovery and Data Mining*, pp. 342-346, 1999.

[9] J. Pei, J. W. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,” *Proceedings of IEEE Conference on Data Engineering*, pp. 215-224, 2001.

[10] J. Pei, J. W. Han, and W. Wang, “Mining Sequential Patterns with Constraints in Large Databases,” *Proceedings of ACM Conference on Information and Knowledge Management*, 2002.

[11] H. Pinto, J. W. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal, “Multi-Dimensional Sequential Pattern Mining,” *Proceedings of ACM Conference on Information and Knowledge Management*, pp. 81-88, 2001.

[12] P. Y. Rolland, “FlExPat: Flexible Extraction of Sequential Patterns,” *Proceedings of IEEE Conference on Data Mining*, pp. 481-488, 2001.

[13] R. Srikant and R. Agrawal, “Mining Sequential Patterns: Generalizations and Performance Improvements,” *Proceedings of International Conference on Extending Database Technology*, pp. 3-17, 1996.

[14] M. A. Weiss, *Data Structures and Algorithm Analysis in C – 2nd ed.*, Addison-Wesley, pp. 110-122, 1997.

[15] J. J. Wesselink, B. Iglesia, S. A. James, J. L. Dicks, I. N. Roberts, and V. J. Rayward-Smith, “Determining a Unique Defining DNA Sequence for Yeast Species Using Hashing Techniques,” *Bioinformatics*, 18(7): 1004-1010, 2002.

[16] Y. H. Wu and A. L. P. Chen, “Prediction of Web Page Accesses by Proxy Server Log,” *World Wide Web: Internet and Web Information Systems*, 5(1): 67-88, 2002.

[17] J. Yang, W. Wang, P. S. Yu, and J. W. Han, “Mining Long Sequential Patterns in a Noisy Environment,” *Proceedings of ACM SIGMOD Conference*, 2002.

[18] M. J. Zaki, “SPADE: An Efficient Algorithm for Mining Frequent Sequences,” *Machine Learning*, 42(1): 31-60, 2001.

Appendix

The Dynamic DISC-all Algorithm

```
Input: A <$\lambda$>-partition X, where the length of $\lambda$ is k and
       the maximum NRR threshold $\gamma$
Output: All the frequent sequences in the <$\lambda$>-partition
1. Scan X once to find all the frequent (k+1)-sequences
   with prefix <$\lambda$>
2. Let $NRR_X$=NRR of X
3. If ($NRR_X < \gamma$)
   (a) Generate the set of partitions at the next level SP
   (b) For each partition in SP, call Dynamic DISC-all
4. Else
   Let k=k+2, Repeat
   (a) Generated the k-sorted database of X
   (b) Call Frequent k-sequence discovery in Figure 4
   (c) Let k=k+1
   Until (size of X < $\delta$) or (no frequent (k-1)-sequence)
```

Note: for the original database, λ=NULL and k=0.