Effective Sequential Pattern Mining Algorithms for Dense Database

Zhenglu YANG†, Yitong WANG†, and Masaru KITSUREGAWA†

† Institute of Industrial Science, The Univeristy of Tokyo
Komaba 4–6–1, Meguro-Ku, Tokyo, 153–8505 Japan

Abstract Sequential pattern mining is very important because it is the basis of many applications. Although there has been a great deal of effort on sequential pattern mining in recent years, its performance is still far from satisfactory because of two main challenges: large search spaces and the ineffectiveness in handling dense data sets. To offer a solution to the above challenges, we have proposed a series of novel algorithms, called the LAst Position INduction (LAPIN) sequential pattern mining, which is based on the simple idea that the last position of an item, α, is the key to judging whether or not a frequent k-length sequential pattern can be extended to be a frequent (k+1)-length pattern by appending the item α to it. LAPIN can largely reduce the search space during the mining process, and is very effective in mining dense data sets. Our experimental data and performance studies show that LAPIN outperforms PrefixSpan [5] by up to an order of magnitude on long pattern dense data sets.

Key words algorithm, sequence mining, performance evaluation

1. Introduction

Sequential pattern mining, which extracts frequent subsequences from a sequence data-base, has attracted a great deal of interest during the recent surge in data mining research because it is the basis of many applications, such as customer behavior analysis, stock trend prediction, and DNA sequence analysis. The sequential mining problem was first introduced in [8]; two sequential patterns examples are: "80% of the people who buy a television also buy a video camera within a day", and "Every time Microsoft stock drops by 5%, then IBM stock will also drop by at least 4% within three days". The above patterns can be used to determine the efficient use of shelf space for customer convenience, or to properly plan the next step during an economic crisis. Sequential pattern mining is also very important for analyzing biological data [3], in which long patterns frequently appear.

Sequence discovery can be thought of as essentially an association discovery over a temporal database. While association rules [7] discern only intra-event patterns (itemsets), sequential pattern mining discerns inter-event patterns (sequences).

Much work has been carried out on mining frequent patterns, as for example, in [7] [11] [6] [5] [4] [2]. However, all of these works suffer from the problems of having a large search space and the ineffectiveness in handling dense data sets. In this work, we propose a new strategy to reduce the space necessary to be searched. Instead of searching the entire projected database for each item, as PrefixSpan [5] does, we only search a small portion of the database by recording the last position of each item in each sequence. Because support counting is usually the most costly step in sequential pattern mining, the LAst Position INduction (LAPIN) technique can improve the performance greatly by avoiding cost scanning and comparisons using a pre-constructed table in bit vector format.

1.1 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of items. A subset of I is called an $itemset$ or an $element$. A $sequence$, s, is denoted as $\langle t_1, t_2, \ldots, t_l\rangle$, where t_j is an itemset, i.e., $(t_j \subseteqq I)$ for 1 ≦ j ≦ l. The $itemset$, t_j, is denoted as $(x_1 x_2 \ldots x_m)$, where x_k is an item, i.e., $x_k \in I$ for 1 ≦ k ≦ m. For brevity, the brackets are omitted if an $itemset$ has only one item. That is, $itemset$ (x) is written as x. The number of items in a sequence is called the $length$ of the sequence. A sequence with length l is called an l-$sequence$. A sequence, $s_a = \langle a_1, a_2, \ldots, a_n\rangle$, is contained in another sequence, $s_b = \langle b_1, b_2, \ldots, b_m\rangle$, if there exists integers $1 \leqq i_1 < i_2 < \ldots < i_n \leqq m$, such that $a_1 \subseteqq b_{i_1}$, $a_2 \subseteqq b_{i_2}, \ldots, a_n \subseteqq b_{i_n}$. We denote s_a a $subsequence$ of s_b, and s_b a $supersequence$ of s_a. Given a sequence $s = \langle s_1, s_2, \ldots, s_l\rangle$, and an item α, $s \diamond \alpha$ denotes that s concatenates with α, which has two possible forms, such as $Itemset\ Extension$ (IE), $s \diamond \alpha = \langle s_1, s_2, \ldots, s_l \cup \{\alpha\}\rangle$, or $Sequence\ Extension$ (SE), $s \diamond \alpha = \langle s_1, s_2, \ldots, s_l, \{\alpha\}\rangle$. If $s' = p \diamond s$, then p is a $prefix$ of s' and s is a $suffix$ of s'.

A $sequence\ database$, S, is a set of tuples $\langle sid, s\rangle$, where sid is a sequence_id and s is a sequence. A tuple $\langle sid, s\rangle$ is said to contain a sequence β, if β is a $subsequence$ of s. The support of a sequence, β, in a sequence database, S, is the number of tuples in the database containing β, denoted as $support(\beta)$. Given a user specified positive integer, ε, a sequence, β, is called a frequent sequential pattern if $support(\beta) \geqq \varepsilon$. In this work, the objective was to find the complete set of sequential patterns of database S in an efficient manner.

Table 1 Sequence Database

SID	Sequence
10	$ac(bc)d(abc)ad$
20	$b(cd)ac(bd)$
30	$d(bc)(ac)(cd)$

Example 1. Let our running database be the sequence database S shown in Table 1 with min_support = 2. We will use this sample database throughout the paper. We can see that the set of items in the database is {a,b,c,d}. The length of the second sequence is equal to 7. A 2-sequence $\langle ac\rangle$ is contained in the sequence 10, 20, and 30, respectively, and its support is equal to 3. Therefore, $\langle ac\rangle$ is a frequent pattern.

1.2 Related Work

Sequential pattern mining algorithms can be grouped into two categories. One category is Apriori-like algorithm, such as GSP [11], SPADE [6], and SPAM [4], the other category is projection-

Table 2 SE Item Last Position List

SID	Last Position of SE Item
10	$b_{last}=5\ c_{last}=5\ a_{last}=6\ d_{last}=7$
20	$a_{last}=3\ c_{last}=4\ b_{last}=5\ d_{last}=5$
30	$b_{last}=2\ a_{last}=3\ c_{last}=4\ d_{last}=4$

Table 3 IE Item Last Position List

SID	Last Position of IE Item
10	$(ab)_{last}=5\ (ac)_{last}=5\ (bc)_{last}=5$
20	$(cd)_{last}=2\ (bd)_{last}=5$
30	$(bc)_{last}=2\ (ac)_{last}=3\ (cd)_{last}=4$

based pattern growth, such as PrefixSpan [5].

Srikant and Agrawal proposed the GSP algorithm [11], which iteratively generates candidate k-sequences from frequent (k-1)-sequences based on the anti-monotone property that all the subsequences of a frequent sequence must be frequent. Zaki proposed SPADE [6] to elucidate frequent sequences using efficient lattice [1] search techniques and simple join operations. SPADE divides the candidate sequences into groups by items, and transforms the original sequence database into a vertical ID-List database format, in which each id is associated with its corresponding items and a time stamp. SPADE counts the support of a candidate k-sequence generated by merging the ID-Lists of any two frequent (k-1)-sequences with the same (k-2)-prefix in each iteration. Ayres et al. [4] proposed the SPAM algorithm, which uses SPADE's lattice concept, but represents each ID-List as a vertical bitmap. SPADE and SPAM use a lot of time on merging and bitmap ANDing operations.

On the other hand, Pei et al. proposed a projection-based algorithm, PrefixSpan [5], which projects sequences into different groups called $projected\ databases$. All the sequences in each group have the same prefix. The PrefixSpan algorithm first scans the database to find the frequent 1-sequences. Then, the sequence database is projected into different groups according to these frequent items, where each group is the projection of the sequence database with respect to the corresponding 1-sequence. For these projected databases, the PrefixSpan algorithm continues to find the frequent 1-sequences to form the frequent 2-sequences with the same corresponding prefix. Recursively, the PrefixSpan algorithm generates a projected database for each frequent k-sequence to find the frequent (k+1)-sequences. To obtain the sequential pattern, PrefixSpan constructs a S-Matrix in each recursive step. PrefixSpan uses a lot of time because it needs to scan the entire projected database, which can be very large.

1.3 Overview of Our Algorithm

As Ayres et al. did in [4], our mining process includes two steps: a $sequence\text{-}extension\ step$ ($S\text{-}Step$) and a $itemset\text{-}extension\ step$ ($I\text{-}Step$) in a standard depth-first manner.

Discovering (k+1)-length frequent patterns. For any time series database, the last position of an item is the key used to judge whether or not the item can be appended to a given prefix (k-length) sequence (assumed to be s). For example, in a sequence, if the last position of item α is smaller than, or equal to, the position of the last item in s, then item α cannot be appended to s as a (k+1)-length sequence extension in the same sequence.

Example 2. When scanning the database in Table 1 for the first time, we obtain Table 2, which is a list of the last positions of the 1-length frequent sequences in ascending order. At the same time, we can obtain Table 3, which is a list of the last positions of the frequent 2-length IE sequences in ascending order. Suppose that we have a prefix frequent sequence $\langle a\rangle$, and its positions in Table 1 are 10:1, 20:3, 30:3, where sid:eid represents the sequence ID and the element ID. Then, we check Table 2 to obtain the first indices whose positions are larger than $\langle a\rangle$'s, resulting in 10:1, 20:2, 30:3, i.e., (10:b_{last} = 5, 20:c_{last} = 4, and 30:c_{last} = 4). We start from these indices to the end of each sequence, and increment the support of each passed item, resulting in $\langle a\rangle : 1, \langle b\rangle : 2, \langle c\rangle : 3$, and $\langle d\rangle : 3$, from which, we can determine that $\langle ab\rangle$, $\langle ac\rangle$ and $\langle ad\rangle$ are the frequent patterns. In our implementation, we constructed a mapping table for a specific position to the corresponding index of the item-last-position list, thus avoiding searching in each iteration. The I-Step methodology is similar to the S-Step methodology, with the only difference being that, when constructing the mapping table, I-Step maps the specific position to the index whose position is equal to or larger than the position in Table 3. To determine the itemset extension pattern of the prefix sequence $\langle a\rangle$, we obtain its mapped indices in Table 3, which are 10:1, 20:2, and 30:2. Then, we start from these indices to the end of each sequence, and increment the support of each passed item, resulting in $\langle (ab)\rangle : 1$, and $\langle (ac)\rangle : 2$. We can also obtain the support of the 3-length sequences $\langle a(bc)\rangle : 1, \langle a(bd)\rangle : 1$, and $\langle a(cd)\rangle : 1$, which is similar to the bi-level strategy of PrefixSpan, but we avoid scanning the entire projected database.

Table 4 Last Position of DB (S-Step)

SID	Sequence
10	$**(**)*(*bc)ad$
20	$*(**)ac(bd)$
30	$*(b*)(a*)(cd)$

From the above example, we can show that the main difference between LAPIN and previous works is the scope of the search space. PrefixSpan scans the entire projected database to find the frequent pattern. SPADE temporally joins the entire ID-List of the candidates to obtain the frequent pattern of next layer. LAPIN can obtain the same result by scanning only part of the search space of PrefixSpan and SPADE, which indeed, are the last positions of the items. Table 4 shows the search space of LAPIN based on Table 1 ($S\text{-}Step$). We can avoid scanning the $*$ part in the projected database or in the ID-List. Let $\bar{D}$ be the average number of customers (i.e., sequences) in the projected DB, $\bar{L}$ be the average sequence length in the projected DB, $\bar{N}$ be the average total number of the distinct items in the projected DB, and m be the distinct item recurrence rate or density in the projected DB. Then m=$\bar{L}/\bar{N}$ ($m \geqq 1$), and the relationship between the runtime of PrefixSpan (T_{ps}) and the runtime of LAPIN (T_{lapin}) in the support counting part is

$$T_{ps}/T_{lapin} = (\bar{D}\times\bar{L})/(\bar{D}\times\bar{N}) = (\bar{D}\times\bar{L})/(\bar{D}\times\bar{L}/m) = m \quad (1).$$

Because support counting is usually the most costly step in the entire mining process, Formula (1) illustrates the main reason why our LAPIN algorithm is faster than PrefixSpan for dense data sets, whose m (density) can be very high. For example, suppose we have a special data set, which has only one single long sequence with one distinct item a and the sequence length is 100. The total time used to scan the projected databases in PrefixSpan is $100 + 99 + 98 + 97 + \ldots + 1$=5050. However, LAPIN only needs $100 + 1 + 1 + \ldots + 1$=199 scanning time. Hence, we have m=5050/199≈25. From this example, we know that scanning most of the duplicate items in the projected DB is useless and time consuming.

The remainder of this paper is organized as follows. In Section 2, we introduce a series of LAPIN algorithms in detail. Our experimental results and performance analysis are reported in Section 3. We conclude the paper and provide suggestions for future work in Section 4.

2. LAPIN Sequential Pattern Mining

2.1 Definitions, Lemmas, and Theorem

[Definition 1] (Prefix border position set) Given two sequences,

Table 5 SE Position List of DB

SID	$Item\ Positions$
10	$a:1 \rightarrow 5 \rightarrow 6 \rightarrow null$ $b:3 \rightarrow 5 \rightarrow null$ $c:2 \rightarrow 3 \rightarrow 5 \rightarrow null$ $d:4 \rightarrow 7 \rightarrow null$
20	$a:3 \rightarrow null$ $b:1 \rightarrow 5 \rightarrow null$ $c:2 \rightarrow 4 \rightarrow null$ $d:2 \rightarrow 5 \rightarrow null$
30	$a:3 \rightarrow null$ $b:2 \rightarrow null$ $c:2 \rightarrow 3 \rightarrow 4 \rightarrow null$ $d:1 \rightarrow 4 \rightarrow null$

A=$\langle A_1 A_2 \dots A_m \rangle$ and B=$\langle B_1 B_2 \dots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \dots C_l \rangle$ for $l \leqq m$ and $l \leqq n$, and that C is a common prefix for A and B. We record both positions of the last item C_l in A and B, respectively, e.g., C_l=A_i and C_l=B_j. The position set, (i, j), is called the *prefix border position set* of the common prefix C, denoted as $\mathcal{S}_c$. Furthermore, we denote $\mathcal{S}_{c,i}$ as the prefix border position of the sequence, i.

For example, if A=$\langle abc \rangle$ and B=$\langle acde \rangle$, then we can deduce that one common prefix of these two sequences is $\langle ac \rangle$, whose prefix border position set is (3,2), which is the last item c's positions in A and B.

[Definition 2] (Local candidate item list) Given two sequences, A=$\langle A_1 A_2 \dots A_m \rangle$ and B=$\langle B_1 B_2 \dots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \dots C_l \rangle$ for $l \leqq m$ and $l \leqq n$, and that C is a common prefix for A and B. Let $D = (D_1 D_2 \dots D_k)$ be a list of items, such as those appended to C, and $C' = C \diamond D_j$ $(1 \leqq j \leqq k)$ is the common sequence for A and B. The list D is called the $local\ candidate\ item\ list$ of the prefix C'.

For example, if A=$\langle abce \rangle$ and B=$\langle abcde \rangle$, we can deduce that one common prefix of these two sequences is $\langle ab \rangle$, and $\langle abc \rangle$, $\langle abe \rangle$ are the common sequences for A and B. Therefore, the item list (c,e) is called the $local\ candidate\ item\ list$ of the prefixes $\langle abc \rangle$ and $\langle abe \rangle$.

[Definition 3] (SE Item-last-position list) Given two sequences, A=$\langle A_1 A_2 \dots A_m \rangle$ and B=$\langle B_1 B_2 \dots B_n \rangle$, the list of the last positions of the different frequent 1-length items in ascending order (or if the same, based on alphabetic order) for these two sequences is called the *SE item-last-position list*, denoted as L_s. Furthermore, we denote $L_{s,n}$ as the $item\text{-}last\text{-}position\ list$ of the sequence, n. Each node of $L_{s,n}$ is associated with two values, i.e., an item and an element number (denoted as $D_{s,n}.item$ and $D_{s,n}.num$ for $D_{s,n} \in L_{s,n}$)

[Definition 4] (IE Item-last-position list) Given two sequences, A=$\langle A_1 A_2 \dots A_m \rangle$ and B=$\langle B_1 B_2 \dots B_n \rangle$, the list of the last positions of the different frequent 2-length IE sequences in ascending order (or if same, based on alphabetic order) for these two sequences is called the *IE item-last-position list*, denoted as L_i. Furthermore, we denote $L_{i,n}$ as the $item\text{-}last\text{-}position\ list$ of the sequence, n. Each node of $L_{i,n}$ is associated with two values, i.e., an item and an element number (denoted as $D_{i,n}.item$ and $D_{i,n}.num$ for $D_{i,n} \in L_{i,n}$).

For example, we can see that Table 2 and Table 3 are the SE and IE $item\text{-}last\text{-}position\ lists$ of the database in Table 1.

[Lemma 1] (Sequence Extension checking) For a prefix sequence, C, in a sequence, i, if the prefix border position, $\mathcal{S}_{c,i}$, is smaller than the last position of a candidate SE item, α, in the same sequence, then C can be extended to $C \diamond \alpha$ as a $Sequence\ Extension$ in the sequence, i.

Proof: Since the last position of the candidate SE item α is larger than $\mathcal{S}_{c,i}$, at least one α appears behind the prefix sequence C in the sequence i, which means the $Sequence\ Extension\ C \diamond \alpha$ exists in the sequence, i.

LAPIN Algorithm :
Input : A sequence database, and the minimum support threshold, ε
Output : The complete set of sequential patterns

Function : Gen_Pattern(α, S, $CanI_s$, $CanI_i$)
Parameters : α = length k frequent sequential pattern; S = prefix border position set of (k-1)-length sequential pattern; $CanI_s$ = candidate sequence extension item list of length k+1 sequential pattern; $CanI_i$ = candidate itemset extension item list of length k+1 sequential pattern
Goal : Generate (k+1)-length frequent sequential pattern

Main():
1. Scan DB once to do:
  1.1 $P_s \leftarrow$ Create the position list representation of the 1-length SE sequences
  1.2 $B_s \leftarrow$ Find the frequent 1-length SE sequences
  1.3 $L_s \leftarrow$ Obtain the item-last-position list of the 1-length SE sequences
  1.4 $B_i \leftarrow$ Find the frequent 2-length IE sequences
  1.5 $P_i \leftarrow$ Construct the position lists of the frequent 2-length IE sequences
  1.6 $L_i \leftarrow$ Obtain the item-last-position list of the frequent 2-length IE sequences
2. For each frequent SE sequence α_s in B_s
  2.1 Call Gen_Pattern (α_s, 0, B_s, B_i)
3. For each frequent IE sequence α_i in B_i
  2.2 Call Gen_Pattern (α_i, 0, B_s, B_i)

Function Gen_Pattern(α, S, $CanI_s$, $CanI_i$)
4. $S_\alpha \leftarrow$ Find the prefix border position set of α based on S
5. $FreItem_{s,\alpha} \leftarrow$ Obtain the SE item list of α based on $CanI_s$ and S_α
6. $FreItem_{i,\alpha} \leftarrow$ Obtain the IE item list of α based on $CanI_i$ and S_α
7. For each item γ_s in $FreItem_{s,\alpha}$
  7.1 Combine α and γ_s as SE, results in θ and output
  7.2 Call Gen_Pattern (θ, S_α, $FreItem_{s,\alpha}$, $FreItem_{i,\alpha}$)
8. For each item γ_i in $FreItem_{i,\alpha}$
  8.1 Combine α and γ_i as IE, results in η and output
  8.2 Call Gen_Pattern (η, S_α, $FreItem_{s,\alpha}$, $FreItem_{i,\alpha}$)

Figure 1 LAPIN Algorithm pseudo code

[Lemma 2] (Itemset Extension checking) For a prefix sequence, C, in a sequence, i, if the prefix border position, $\mathcal{S}_{c,i}$, is smaller than, or equal to the last position of a candidate IE item, β, in the same sequence, then C can be extended to $C \diamond \beta$ as an $Itemset\ Extension$ in the sequence, i.

Proof: Since the last position of the candidate IE item β is larger than or equal to $\mathcal{S}_{c,i}$, at least one β appears behind the prefix sequence C in the sequence i, which means the $Itemset\ Extension$ $C \diamond \alpha$ exists in the sequence i.

[Theorem 1] (Frequent sequence) Given a user specified minimum support, ε, a sequence, S, is frequent if, by $Sequence\ Extension\ checking$, its support, $Sup(S), is \geqq \varepsilon$, or, by $Itemset\ Extension\ checking$, its support, $Sup(S), is \geqq \varepsilon$.

2.2 LAPIN: Design and Implementation

In this section, we describe the LAPIN algorithms used to mine sequential patterns in detail. As in other algorithms, certain key strategies were adopted, i.e., candidate sequence pruning, database partitioning, and customer sequence reducing. Combined with the LAPIN strategy, our algorithms can efficiently find the complete set of frequent patterns. We used the Depth First Search (DFS). The pseudo code of LAPIN is shown in Figure 1.

In Step 1, by scanning the DB once, we can obtain the SE position list table, as in Table 5 and all the 1-length frequent patterns. Based on the last element in each position list, we can sort and construct the $SE\ item\text{-}last\text{-}position\ list$ in ascending order, as shown in Table 2. To find the frequent 2-length IE sequences, during the first scan, we construct a 2-dimensional array indexed by the items' ID and update the counts for the corresponding 2-length IE sequences by using similar methods to those used in [6]. Then, we merge the SE position lists of the two items, which compose the frequent 2-length IE sequence, to obtain the 2-length IE sequence position list. Finally, we sort and construct the IE $item\text{-}last\text{-}position\ list$ of each frequent 2-length IE sequence in ascending order, as shown in Table 3. As Example 2 shows, the I-Step methodology is similar to the S-Step methodology in LAPIN. We

DB(i)

CID	Seq.
10	a a
20	a b
30	a c
40	a d
50	a e

DB(ii)

CID	Seq.
10	a a a a a a
20	a b b b b b

(a) Two special DBs

	Avg. suffix	Avg. local cand. item list	Suffix-oriented	LCI-oriented
DB(i)	1	5	5 times	25 times
DB(ii)	5	2	10 times	4 times

(b) Effect on different type of DBs

Figure 2 Performance of Suffix-oriented and LCI-oriented algorithms on different DB

will first describe the S-Step process, and the I-Step process will be explained in detail in Section 2.3.4.

In function $Gen\_Pattern$, to find the prefix border position set of k-length α (Step 4), we first obtain the position list of the last item of α, and then perform a binary search in the list for the (k-1)-length prefix border position. (We can do this because the position list is in ascending order.) For S-$Step$, we look for the first position that is larger than the (k-1)-length prefix border position.

Step 5, shown in Figure 1, is used to find the frequent SE (k+1)-length pattern based on the frequent k-length pattern and the 1-length candidate items. Step 5 can be justified based on Theorem 1 in Section 2.2. Commonly, support counting is the most time consuming part in the entire mining process. Here, we face a problem. "Where do the appended 1-length candidate items come from?" We can test each candidate item in the local candidate item list (LCI-$oriented$), which is similar to the method used in SPADE [6] and SPAM [4]. Another choice is to test the candidate item in the projected DB, just as PrefixSpan [5] does ($Suffix$-$oriented$). The correctness of these methods was discussed in [6] and [5], respectively.

We have found that LCI-$oriented$ and $Suffix$-$oriented$ have their own advantages for different types of data sets. Suppose that we have two sequence databases, as shown in Figure 2 (a), the prefix sequence is a, and the min_support = 1. To test the 2-length candidate sequences, whose prefix is a for DB (i), the $Suffix$-$oriented$ algorithm scans the projected DB, which requires a $1 \times 5 = 5$ scanning time. The LCI-$oriented$ algorithm scans the local candidate item list for each sequence, which requires a $5 \times 5 = 25$ scanning time. However, for DB (ii), the $Suffix$-$oriented$ algorithm requires a $5 \times 2 = 10$ scanning time, and the LCI-$oriented$ algorithm requires a $2 \times 2 = 4$ scanning time. The effect of these two data sets on the two approaches is shown in Table 2 (b).

The above example illustrates that, if the average suffix sequence length is less than the average local candidate item list size, as in DB (i), then $Suffix$-$oriented$ spends less time. However, if the average suffix sequence length is larger than the average local candidate item list size, as in DB (ii), then LCI-$oriented$ is faster.

Based on this discovery, we formed a series of algorithms categorized into two classes. One class was LCI-$oriented$, LAPIN_LCI, and the other class was $Suffix$-$oriented$, LAPIN_Suffix. We can dynamically compare the suffix sequence length with the local candidate item list size and select the appropriate search space to build a single general framework. However, because we used a space consuming bitmap strategy in LAPIN_LCI, which will be explained in Section 2.3.1, in order to save memory space and clarify the advantages and disadvantages of each method, we deconstructed the general framework into two approaches. Nevertheless, it is easy to combine these two approaches, and evaluate the efficiency of the entire general framework, whose runtime, T, and maximum memory space required, M, are

$$T \approx \{T_{LAPIN\_LCI}, T_{LAPIN\_Suffix}\}_{min}$$
$$M \approx \{M_{LAPIN\_LCI}, M_{LAPIN\_Suffix}\}_{max}.$$

LAPIN_LCI. LAPIN_LCI tests each item which is in the local candidate item list. In each customer sequence, it directly judges whether an item can be appended to the prefix sequence or not by comparing this item's last position with the prefix border position. Increment the support value of the candidate item by 1 if the candidate item's last position is larger than the prefix border position. As an optimization, we can use bitmap strategy to avoid such comparison process. A pre-constructed table, named ITEM_IS_EXIST_TABLE is constructed while first scanning to record the last position information. For example, Figure 3 (a), which is based on the example database shown in Table 1, shows one part of the ITEM_IS_EXIST_TABLE for the first sequence. The left-hand column denotes the position number and the top row is the item ID. In the table, we use a bit vector to represent all the 1-length frequent items existing for a specific position. If the bit value is unity, then it indicates that the corresponding item exists. Otherwise, the item does not exist. The bit vector size is equal to the size of the 1-length frequent items list. For example, when the current position is 5, we obtain the bit vector 1001, indicating that only items a and d exist in the same sequence after the current prefix. To accumulate the candidate sequence's support, we only need to check this table, and add the corresponding item's vector value, thus avoiding the comparison process.

Pos \ Item	a	b	c	d
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	0	0	1
6	0	0	0	1
7	0	0	0	0

(a)ITEM_IS_EXIST_TABLE

Pos \ Item	a	b	c	d
5	1	0	0	1
6	0	0	0	1

(b)Optimized ITEM_IS_EXIST_TABLE

Figure 3 Bitmap representation table

Input : S_α = prefix border position set of length k frequent sequential pattern α; BV_s = bit vectors of the ITEM_IS_EXIST_TABLE; $CanI_s$ = candidate sequence extension items; ε = user specified minimum support
Output : $FreItem_s$ = local frequent SE item list

1. For each sequence, F
2. $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in S_α
3. bitV $\leftarrow$ obtain the bit vector of the $S_{\alpha,F}$ indexed from BV_s
4. For each item β in $CanI_s$
5. Suplist[β] = Suplist[β] + bitV[β];
6. For each item γ in Suplist
7. if (Suplist[γ] $\geqq \varepsilon$)
8. $FreItem_s$.insert(γ);

Figure 4 Finding the SE frequent patterns using LAPIN_LCI

Input : S_α = prefix border position set of length k frequent sequential pattern α; $L_s = SE$ item-last-position list; ε = user specified minimum support
Output : $FreItem_s$ = local frequent SE item list

1. For each sequence, F
2. $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in S_α
3. $L_{s,F} \leftarrow$ obtain SE item-last-position list of F in L_s
4. M = Find the corresponding index for $S_{\alpha,F}$
5. while (M $<$ $L_{s,F}$.size)
6. Suplist[M.item]++;
7. M++;
8. For each item β in Suplist
9. If (Suplist[β] $\geqq \varepsilon$)
10. $FreItem_s$.insert(β);

Figure 5 Finding the SE frequent patterns using LAPIN_Suffix

Space Optimization of LAPIN_LCI. We found that only part of the table was useful, and that most was not. The optimized ITEM_IS_EXIST_TABLE is shown in Figure 3 (b), which stores only two bit vectors instead of the seven shown in Figure 3 (a). We used an array to map each specific position to the index in the optimized ITEM_IS_EXIST_TABLE. For a dense data set, this space saving strategy proved more efficient. The pseudo code of LAPIN_LCI is shown in Figure 4.

Example 3. Let us assume that we have obtained the prefix border position set of the pattern $\langle a \rangle$ in Table 1, i.e., (1,3,3). We also know that the $local\ candidate\ item\ list$ is (a, b, c, d). Then, in-

Table 6 IE Position List of DB

SID	$Item\ Positions$
10	$(ab):5 \rightarrow null$
	$(ac):5 \rightarrow null$
	$(bc):3 \rightarrow 5 \rightarrow null$
20	$(bd):5 \rightarrow null$
	$(cd):2 \rightarrow null$
30	$(ac):3 \rightarrow null$
	$(bc):2 \rightarrow null$
	$(cd):4 \rightarrow null$

stead of comparing each last position of the candidate item with the prefix border position, we obtain the bit vector mapped from the specific position. Here, we obtain the bit vectors 1111, 0111, and 0011 with respect to the pattern $\langle a \rangle$'s prefix border position set, (1,3,3), and accumulate them, resulting in $\langle a \rangle : 1, \langle b \rangle : 2, \langle c \rangle : 3$, and $\langle d \rangle : 3$. From here, we can deduce that $\langle ab \rangle$, $\langle ac \rangle$, and $\langle ad \rangle$ are frequent patterns.

LAPIN_Suffix. When the average size of the candidate item list is larger than the average size of the suffix, then scanning in the suffix to count the support of the (k+1)-length sequences is better than scanning in the local candidate item list, such as for DB (i) in Figure 2. Therefore, we proposed a new algorithm, LAPIN_Suffix. In the $item\text{-}last\text{-}position\ list$, i.e., Table 2, we look for the first element whose last position is larger than the prefix border position. Then, we go to the end of this list and increment each passed item's support. Obviously, we only pass and count once for each different item in the suffix (projected database) because, in $item\text{-}last\text{-}position\ list$, we record the last position of each item for a specific sequence. In contrast, PrefixSpan needs to pass every item in the projected database regardless of whether or not they are the same as before. Therefore, LAPIN_Suffix will save much time because our search space is only a subset of the one used in PrefixSpan. The pseudo code of LAPIN_Suffix is shown in Figure 5. Example 2 in Section 1.3 describes the flow of LAPIN_Suffix.

I-Step of LAPIN. In LAPIN, the $I\text{-}Step$ is similar to the $S\text{-}Step$. From Step 1 in Figure 1, we can obtain the frequent 2-length IE sequence position list, as shown in Table 6, and the $I\text{-}Step\ item\text{-}last\text{-}position\ list$, as shown in Table 3. In Step 4 of Figure 1, we first obtain the position list of the last 2-length IE item of α, and then perform a binary search in Table 6. Here, we look for the first position that is equal to, or larger than the (k-1)-length prefix border position. To find the frequent (k+1)-length IE sequences in Step 6 of Figure 1, which is similar to S-Step, there are two classes of algorithm. One is $LCI\text{-}oriented$, which directly compares the last position of the 2-length IE sequences with the prefix border positions to judge whether or not the frequent k-length sequence can be appended to the 2-length IE sequence to form a (k+1)-length IE sequence. The first item of the 2-length IE sequence should be the same as the last item of the k-length prefix sequence. The other algorithm is $Suffix\text{-}oriented$, which uses Table 3 to facilitate the $I\text{-}Step$ support counting.

3. Performance Study

In this section, we will describe our experiments and evaluations conducted on both synthetic and real data, and compare LAPIN with PrefixSpan to demonstrate the efficiency of the proposed algorithms. We performed the experiments using a 1.6 GHz Intel Pentium(R)M PC machine with a 1 G memory, running Microsoft Windows XP. All three algorithms are written in C++ software, and were compiled in an MS Visual C++ environment. The output of the programs was turned off to make the comparison equitable.

3.1 Comparing PrefixSpan with the LAPIN Algorithms

We first compared PrefixSpan and our algorithms using synthetic and real data sets, and showed that LAPIN outperformed PrefixSpan by up to an order of magnitude on dense data sets with long patterns and low minimum support.

Table 7 Parameters used in data set generation

Symb.	Meaning
D	Number of customers in the data set
C	Average number of transactions per customer
T	Average number of items per transaction
S	Average length of maximum sequences
I	Average length of transactions within maximum sequences
N	Number of different items in the data set

Synthetic Data. The synthetic data sets were generated by an IBM data generator, as described in [8]. The meaning of the different parameters used to generate the data sets is shown in Table 7. In the first experiment, we compared PrefixSpan and our algorithms using several small-, medium-, and large- sized data sets for various minimum supports. The statistics of these data sets is shown in Figure 6 (a).

PrefixSpan vs. LAPIN: We defined $search\ space$ as in PrefixSpan, to be the size of the projected DB, denoted as S_{ps}, and in LAPIN the sum of the number of different items for each sequences in the suffix (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI), denoted as S_{lapin}. Figure 6 (b) and Figure 6 (c) show the running times and the searched space comparison between PrefixSpan and LAPIN and clearly illustrate that PrefixSpan is slower than LAPIN using the medium data set (C30T20S30I20N200D20K) and the large data set (C50T20S50I20N300D100K). This is because the searched spaces of the two data sets in PrefixSpan were much larger than that in LAPIN. For the small data set (C10T5S5I5N100D1K), the ineffectiveness of searched space saving and the initial overhead needed to set up meant that LAPIN was slower than PrefixSpan. Overall, our runtime tests showed that LAPIN excelled at finding the frequent sequences for many different types of large data sets.

Formula (1) in Section1.3 illustrates the relationship between the runtime of PrefixSpan and the runtime of LAPIN in the support counting part. However, for the entire mining time, we also need to consider the initialization part and the implementation detail, which are very difficult to evaluate because of the complexity of the sequential pattern mining problem. Commonly, support counting is usually the most costly step in the entire mining process. Hence, we can approximately express the relationship between the entire mining time of PrefixSpan and that of LAPIN based on Formula (1), where we generalize the meaning of $\bar{N}$ to denote the average total number of the distinct items in either the projected DB (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI), and the meaning of m to denote either the distinct item recurrence rate of the projected DB (LAPIN_Suffix) or the local candidate list (LAPIN_LCI). Formula (1) illustrates that, the higher the value of m is, then the faster LAPIN becomes compared to PrefixSpan. However, the entire mining time of LAPIN is not faster than that of PrefixSpan m times because of the initialization overhead, but near to m times because of the importance of the support counting in the entire mining process. The experimental data shown in Figure 6 (b) and Figure 6 (c) is in accordance with our theoretical analysis, where the $searched\ space$ comparison determines the value of m, $m = S_{ps}/S_{lapin}$.

LAPIN_Suffix vs. LAPIN_LCI: Because LAPIN_Suffix and LAPIN_LCI are implemented in the same framework, in addition to the small difference in the initial phase, the only implementation difference is in the support counting phase: LAPIN_Suffix searches in the suffix, whereas LAPIN_LCI searches in the local candidate item list. Let $\bar{N}_{Suffix}$ be the average total number of the distinct items in the projected DB, $\bar{N}_{LCI}$ be the average total number of the distinct items in the local candidate item list, m_{Suffix} be the distinct item recurrence rate of the projected DB, m_{LCI} be the distinct

Dataset	# sequences	Avg length	Total size
C10T5S5I5N100D1K	1000	46	270K
C30T20S30I20N200D20K	20000	518	46M
C50T20S50I20N300D100K	100000	903	401M

(a) Dataset characteristics

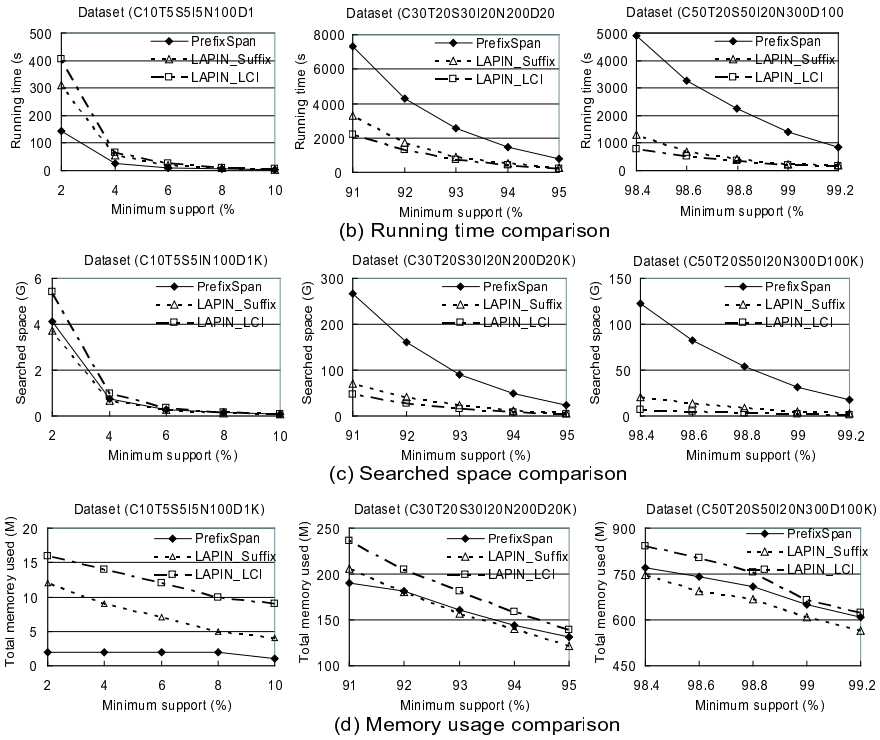

Figure 6 The different sizes of the data sets

item recurrence rate of the local candidate item list. We can express the relationship between the entire mining time of LAPIN_Suffix (T_{Suffix}) and that of LAPIN_LCI (T_{LCI}) as

$$T_{Suffix}/T_{LCI} \approx S_{Suffix}/S_{LCI} = m_{LCI}/m_{Suffix} \quad (2).$$

where we have the searched space of LAPIN_Suffix, S_{Suffix} = $\bar{D} \times \bar{N}_{Suffix}$ = $\bar{D} \times \bar{L}/m_{Suffix}$, and the searched space of LAPIN_Suffix, S_{LCI} = $\bar{D} \times \bar{N}_{LCI}$ = $\bar{D} \times \bar{L}/m_{LCI}$. Formula (2) is in accordance with the experimental data shown in Figure 6 (b) and Figure 6 (c). LAPIN_Suffix is faster than LAPIN_LCI for small data sets because the former one searches smaller spaces than the latter one does. However, for medium and large dense data sets, which have many long patterns, LAPIN_LCI is faster than LAPIN_Suffix because the situation is reversed.

Memory usage analysis: As Figure 6 (d) shows, LAPIN_Suffix expends almost the same amount of memory as PrefixSpan does, except for small data sets because LAPIN_ Suffix uses a little more memory than PrefixSpan to store initialization information. LAPIN_LCI, because it needs to store the items' last position information in bit vector format, requires more space than LAPIN_Suffix and PrefixSpan do. Let C' be the average number of the $key\ positions$ per customer. LAPIN_LCI requires $(D \times C' \times N)/8$ bytes to store the last position information for all the items. From Figure 6, it can be seen that there is a trade-off between LAPIN_Suffix and LAPIN_LCI in terms of speed and space.

Different parameters analysis: In the second experiment, we compared the performance of the algorithms as several parameters in the data set generation were varied. The meaning of these parameters are shown in Table 7. As Figure 7 shows, when C increases, T increases, and N decreases, then the performance of LAPIN improves even more relative to PrefixSpan, by up to an order of magnitude. Let us consider Formula (1), m=$\bar{L}/\bar{N}$=$\bar{C} \times \bar{T}/\bar{N}$, where $\bar{C}$ is the average number of transactions per customer in the projected DB, and $\bar{T}$ is the average number of items per transaction in the projected DB. On keeping the other parameters constant, increasing C, T and decreasing N, respectively, will result in an increase in the distinct item recurrence rate, m, which is in accordance with the experimental data shown in Figure 7. This confirms the correctness of Formula (1).

With regards to the other three parameters, as S, I and D varies, the discrepancy between the running times does not change significantly because these parameters do not apparently contribute to the variance of the distinct item recurrence rate, m, which means that the discrepancy between the searched space does not change much as these three parameters are varied. Between the two LAPIN algorithms, LAPIN_LCI and LAPIN_Suffix, the former one is always the fastest because its searched space is less than that of the latter one. Due to limited space, we do not show the searched space comparison here.

Real Data. We consider that results from real data will be more convincing in demonstrating the efficiency of our proposed algorithm. In this section, we discuss tests on two real data sets, Gazelle and Protein. A portion of Gazelle was used in KDD-Cup 2000. More details on the information in this data set can be found in [10]. The second real data set used, Protein, was extracted from the web site of the National Center for Biotechnology Information (USA) (注1). This was extracted using a conjunction of: (1) search category = "Protein", (2) sequence length range = [400:600], and (3) data submission period = [2004/7/1, 2004/12/31]. The statistics of these data sets is shown in Figure 8 (a).

As shown in Figure 8 (b), LAPIN outperformed PrefixSpan for both the Gazelle and Protein data sets. The reason why LAPIN performed so well was similar to that for the synthetic data sets in Section 3.1.1, and was based on the searched space saving, as shown in Figure 8 (c). This experiment confirmed the superiority of the

(注1)：http://www.ncbi.nlm.nih.gov

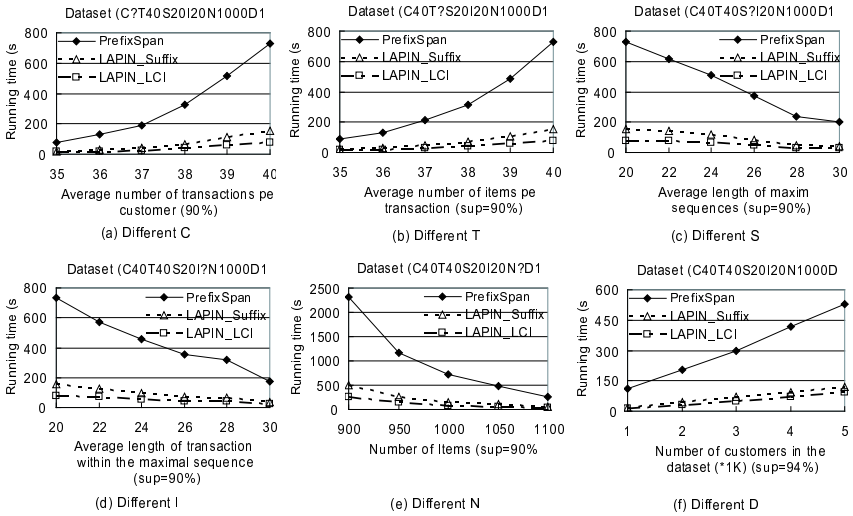

Figure 7 Varying the parameters of the data sets

Dataset	# sequences	# items	Min len	Max len	Avg. len	Total size
Gazelle	59602	497	1	267	2.5	1.4M
Protein	116142	24	400	600	482	438M

(a) Dataset characteristics

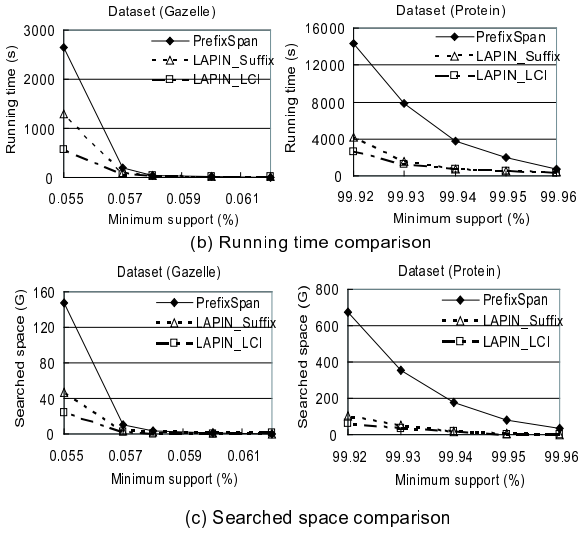

Figure 8 Real data sets

proposed method using real-life data.

4. Conclusions

In this work, we have proposed a series of novel algorithms, LAPIN, for efficient sequential pattern mining. Our main idea is that the last position of an item, α, in each sequence is very useful, and is the key to judging whether or not a k-length frequent sequence could grow to a frequent (k+1)-length sequence by appending the item α to it. Therefore, LAPIN can reduce searching significantly by only scanning a small portion of the projected database or the ID-List, as well as handling dense data sets efficiently, which is inherently difficult for most existing algorithms. By thorough experiments and evaluations, we have demonstrated that LAPIN outperforms PrefixSpan by up to an order of magnitude, which is in accordance with our theoretical analysis. Our experimental results also show that LAPIN is very efficient at using synthetic data and using real-world data, such as web access patterns and protein sequences.

We plan to continue our work by applying our algorithm to other application domains, and will investigate how to extend it to determine a closed frequent sequence.

References

[1] B. A. Davey and H. A. Priestley, "Introduction to Lattices and Order," *Cambridge University Press*, 1990.

[2] D. Chiu, Y. Wu, and A. L. P. Chen, "An Efficient Algorithm for mining Frequent Sequences by a New Strategy without Support Counting," In *20th Int'l Conf. of Data Engineering (ICDE'04)*, pp. 375-386, Boston, USA, Mar. 2004.

[3] E. Eskin and P.A. Pevzner, "Finding Composite Regulatory Patterns in DNA Sequences," In *10th Int'l Conf. on Intelligent Systems for Molecular Biology (ISMB'2002)*, pp. 354-363, Edmonton, Canada, Aug. 2002.

[4] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential Pattern Mining using A Bitmap Representation," In *8th ACM SIGKDD Int'l Conf. Knowledge Discovery in Databases (KDD'02)*, pp. 429-435, Alberta, Canada, Jul. 2002.

[5] J. Pei, J. Han, M. A. Behzad, and H. Pinto, "PrefixSpan:Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," In *17th Int'l Conf. of Data Engineering (ICDE'01)*, Heidelberg, Germany, Apr. 2001.

[6] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," In *Machine Learning*, Vol. 40, pp. 31-60, 2001.

[7] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," In *20th Int'l Conf. on Very Large Databases (VLDB'94)*, pp. 487-499, Santiago, Chile, Sep. 1994.

[8] R. Agrawal and R. Srikant, "Mining sequential patterns," In *11th Int'l Conf. of Data Engineering (ICDE'95)*, pp. 3-14, Taipei, Taiwan, Mar. 1995.

[9] R. J. Bayardo, "Efficiently mining long patterns from databases," In *ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD'98)*, pp. 85-93, Seattle, WA, Jun. 1998.

[10] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng, "KDD -cup 2000 organizer's report: Peeling the Onion," In *SIGKDD Explorations*, vol. 2, pp. 86-98, 2000.

[11] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," In *5th Int'l Conf. Extending Database Technology (EDBT'96)*, pp. 13-17, Avignon, France, Mar. 1996.