Info. and Comm. Engineering Dept. 2005/6/17

Effective Sequential Pattern Mining Algorithms by Last Position Induction

Kitsuregawa Lab. D2 Zhenglu Yang

Abstract

Sequence pattern mining is very important since it is the basis of many applications. Although a great deal of effort has been done on sequential pattern mining in recent years, its performance is still far from satisfying due to two main challenges: large search space and ineffectiveness to handle long pattern in database. In this paper, in order to offer a solution for the above challenges, we propose a novel series of algorithms called LAst Position INduction sequential pattern mining (abbreviated as LAPIN) based on the idea that the last position of an item s is the key to judge whether a frequent k-length sequence pattern could be extended to be a frequent (k+1)-length pattern by appending it with the item s. LAPIN could largely reduce the search space during mining process and is considerable effectiveness in mining long pattern. Our experimental results and performance study show that LAPIN outperforms PrefixSpan up to near an order of magnitude on long pattern dataset.

1 Introduction

Sequential pattern mining, which extracts frequent subsequences from a sequence database, has attracted great interests in the recent surge of data mining research since it is the basis of many applications such as customer behavior, Web access pattern, stock sequence analysis, DNA sequences and so on. The sequential mining problem was first introduced in [4], e.g."80% of the persons who buy television also buy video camera within a week", or "every time Microsoft stock drops 5%, IBM stock will also drops at least 4% within three days" are two sequence patterns. With those patterns, we could efficiently use shelf space to convenient customer or could properly plan our next step during economic crisis. Sequential mining is also very important for biological data and stock data analysis, in which long patterns are frequently appeared.

Sequence discovery can essentially be thought of as association discovery over a temporal database. While association rules [3] discover only intra-event patterns (itemsets), sequential pattern mining discovers inter-event patterns (sequences).

Many works have been done on mining frequent patterns as in [3] [2] [5] [6] [1] [9]. Yet all of these works suffer from the problems that large search space and ineffectiveness to handle long patterns. In this paper, we propose a new strategy to largely reduce the space necessary to be searched. Instead of searching the whole projected database of each item as PrefixSpan did, we only search a small portion of it by recording the last position of each item in each sequence. Because support counting usually is the most costly step in sequential pattern mining, LAPIN could improve the performance greatly by avoiding cost comparisons and ANDing operations with 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., $(t_i \subseteq I)$ for $1 \leq$ j $\leq$ l. 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 $1 \leq$ k $\leq$ 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 exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}, \ldots, a_n \subseteq b_{i_n}$. We can call 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, $s \diamond \alpha$ means s concatenates with α, which has two possible forms such as *Itemset Extension* (abbreviated as IE), $s \diamond \alpha = \langle s_1, s_2, \ldots, s_l \cup \{\alpha\} \rangle$; or *Sequence Extension* (abbreviated as SE), $s \diamond \alpha = \langle s_1, s_2, \ldots, s_l, \{\alpha\} \rangle$. If $s' = p \diamond s$, p is a $prefix$ of s' and s is a $suffix$ of s'.

A *sequence database* S is a set of customer $\langle cid, s \rangle$, where cid is a customer_id and s is a customer sequence. A customer $\langle cid, s \rangle$ is said to contain a sequence α, if α is a *subsequence* of s. The support of a sequence α in a sequence database S is the number of customer sequences in the database containing α, denoted as $support(\alpha)$. Given a user specified positive integer ε, a sequence α is called a frequent sequential pattern if $support(\alpha) \geq \varepsilon$. In this paper, our objective is to find the complete set of sequential patterns of database S in an efficient way.

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

Table 1: Sequence Database

CID	*Customer Sequence*
10	$ac(bc)d(abc)ad$
20	$b(cd)ac(bd)$
30	$d(bc)(ac)(cd)$

Table 2: Last Position of DB (S-Step)

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

1.2 Related Works

Sequential pattern mining algorithms could be grouped into two categories. One is candidate-generate-test such as GSP [2], SPADE [5] and SPAM [1], the other is projected based pattern growth such as PrefixSpan [6].

Srikant and Agrawal proposed the GSP algorithm [2], which 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. Zaki proposed SPADE [5] to find frequent sequences using efficient lattice [7] search techniques and simple join operations. It 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 corresponding items and the time stamp. It 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. SPADE wastes a lot of time on merging the ID-Lists of the candidate sequences. Ayres et al. [1] proposed the SPAM algorithm, which is built based on SPADE's lattice concept but represents each ID-List as a vertical bitmap.

On the other hand, Pei et al. proposed a pattern growth algorithm named PrefixSpan [6], which adopts projection strategy to project the customer sequences into different groups called *projected databases*. All the customer sequences in each group have the same prefix. For the example database in Table 1, 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, each group is the projection of the sequence database with respect to the corresponding 1-sequence. For these projected databases, the PrefixSpan algorithm continues the discovery of frequent 1-sequences to form the frequent 2-sequences with the corresponding prefix. Recursively, the PrefixSpan algorithm generates the projected database for each frequent k-sequence to find frequent (k+1)-sequences. To get the sequence pattern, PrefixSpan constructs a S-Matrix in each recursive step. It spends a lot of time because it needs to scan the whole projected database, which is possibly very large.

The main difference between LAPIN and the previous works is the scope of the search space. PrefixSpan scans the whole projected database to find the frequent pattern. SPADE temporal joins the whole ID-List of the candidates to get the frequent pattern of next layer. LAPIN can get the same result by scanning only part of the search space of PrefixSpan and SPADE, which are indeed the last positions of the items. Table 2 shows the search space of LAPIN based on Table 1 (*S-Step*). We can avoid to scan the $*$ part in the projected database or in the ID-List, which obviously save much more time than PrefixSpan and SPADE for those dense database with long patterns and small number of items.

The rest of this paper is organized as follows. In Section 2 we introduce a series of LAPIN algorithms in detail. In Section 3 the experiments of comparing LAPINs and other algorithms are presented. We conclude the whole paper and give future work direction in Section 4.

2 LAPIN (LAst Position INduction) sequential pattern mining

2.1 Lexicographic Tree

We use the lexicographic tree [1][8] as the search path of our algorithm. Furthermore, we adopt a lexicographic order which is defined in the same way as in [10]. It uses the Depth First Search (DFS) strategy. Figure 1 shows a sample of the lexicographic tree. We obey the following rules based on DFS:

(a) if $\gamma' = \gamma \diamond \theta$, then $\gamma < \gamma'$; (Search the prefix first, then the sequence. For example, we first search $\langle a \rangle$, then $\langle (ab) \rangle$.)

(b) if $\gamma = \alpha \diamond_s \theta$ and $\gamma' = \alpha \diamond_i \theta$, then $\gamma < \gamma'$; (Search the sequence-extension first, then the itemset-extension. For example, we first search $\langle ab \rangle$, then $\langle (ab) \rangle$.)

(c) if $\gamma = \alpha \diamond \theta$ and $\gamma' = \alpha \diamond \theta'$, $\theta < \theta'$ indicates $\gamma < \gamma'$. (For two sequences which have the same prefix, search them based on the alphabetic order of the suffix. For example, we first search $\langle aa \rangle$, then $\langle ab \rangle$.)

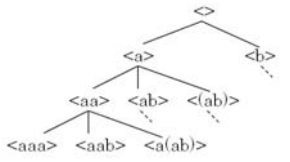


Fig. 1: Lexicographic Tree

2.2 LAPIN: Design and Implementation

In this section, we describe LAPIN algorithms to mine sequential patterns in detail. Like other algorithms, we also adopt some key strategies, 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 the frequent patterns. The pseudo code of LAPIN is shown as Figure 2.

In the step 1 of Figure 2, by scanning DB once, we can get the position list table as in Table 3 and all 1-length frequent patterns. Based on the last element in each position list we can sort and construct the *S-Step item-last-position list* in ascending order. To find the frequent 2-length IE sequences, while first scanning, we construct a 2-dimensional array indexed by the items ID and update the counts for corresponding 2-length IE sequences with similar methods as those used in [5]. Then we merge the SE position lists of the two items, which compose the frequent 2-length IE sequence, to get the 2-length IE sequence position list. Finally, we sort and construct the *I-Step item-last-position list* of each frequent 2-length IE sequence in ascending order. The I-Step is similar to S-Step in LAPIN. We will first describe the S-Step. The I-Step will be explained in Section 2.3.4 in detail.

Table 3: 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	$a:3 \rightarrow null$ $b:2 \rightarrow null$ $c:2 \rightarrow 3 \rightarrow 4 \rightarrow null$ $d:1 \rightarrow 4 \rightarrow null$

In function *Gen_Pattern*, to find the prefix border position set of k-length α (step 4), we first get the position list of the last item of α, then do 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 which is larger than the (k-1)-length prefix border position (in STL, the function is *upper_bound*).

The step 5 in Figure 2 is used to find the frequent SE (k+1)-length pattern based on the frequent k-length pattern and the 1-length candidate items. Commonly, support counting is the most time consuming part in the whole mining process. We have a series of algorithms categorized as two classes. One is *LCI-oriented*, named as LAPIN_LCI, and the other is *Suffix-oriented*, named as LAPIN_Suffix.

LAPIN Algorithm :
Input : A sequence database DB, and the minimum support threshold *min_sup*
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 $L_s \leftarrow$ Get 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 $L_i \leftarrow$ Get 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. $FreItem_{s,\alpha} \leftarrow$ Get the SE item list of α based on $CanI_s$
6. $FreItem_{i,\alpha} \leftarrow$ Get the IE item list of α based on $CanI_i$
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, $FreItem_{s,\alpha}$, $FreItem_{i,\alpha}$)

Fig. 2: LAPIN Algorithm pseudo code

2.2.1 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. Incre-

ment the support value of the candidate item by 1 if the candidate item's last position is larger than the prefix border position. However, the comparison precess will consume much time because the recursive property. Can we avoid such comparison and directly accumulate the candidates support?

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

Fig. 3: ITEM_IS_EXIST_TABLE

We can avoid the comparison operations in LAPIN_LCI by using a pre-constructed table, named ITEM_IS_EXIST_TABLE. The last position information is recorded in a bit vector for each specific position. For example, Figure 3, which is built based on the example database in Table 1, shows one part of the ITEM_IS_EXIST_TABLE for the first customer. The left column is 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 existence for a specific position. If bit value is 1, 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 can get the bit vector as 1001, indicates that only item a and d exist in the same customer 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, avoiding the comparison process.

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

Fig. 4: Optimized ITEM_IS_EXIST_TABLE

Space Optimization of LAPIN_LCI
It can easily show that the main memory used in LAPIN_LCI is no more than that used in SPAM [1] because we use each bit to represent each item's existence for every specific position in each customer sequence. Yet this is a significantly high space cost if the original database is big. After consideration, we found that only part of the table is useful and most are not. For example, in Figure 3, when the prefix border position is smaller than 5, all items exist and when the

---

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; min_sup: user specified minimum support
Output : $FreItem_s$: local frequent SE item list

1. For each customer sequence F
2. $S_{\alpha,F} \leftarrow$ get prefix border position of F in S_α
3. bitV $\leftarrow$ get 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] $\geq min_sup$)
8. $FreItem_s$.insert(γ);

Fig. 5: Improved version of LAPIN_LCI

position is larger than 6, there is no item existing. So the useful information is stored in some key positions' lines. We define *key position* as follows: Given a position, if its corresponding bit vector is different from that of the position one smaller than it (except the one whose bit vector is equal to 0), this position is defined as *key position*. For example, in Figure 3, the position 5 and 6 are *key positions* and others are not (position 7 is not because its bit vector is equal to 0). We can find that these *key positions* are indeed the last positions of the 1-length frequent items (except the last one). The optimized ITEM_IS_EXIST_TABLE is shown in Figure 4, which stores only two bit vectors instead of the seven ones shown in Figure 3. We use an array to map each specific position to the index in the optimized ITEM_IS_EXIST_TABLE. For long pattern dataset, this space saving strategy is more efficient. The improved version of LAPIN_LCI is shown as Figure 5.

2.2.2 LAPIN_Suffix

When the average size of the candidate item list is larger than the average size of the suffix, scanning in the suffix to count the support of (k+1)-length sequences is better than scanning in the local candidate item list. So we propose a new algorithm, named LAPIN_Suffix. In the *item-last-position list*, 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 every passed item's support. Obviously, we only pass and count once for each different item in the suffix (projected database), because in *item-last-position list* we record the last position of each item for a specific customer sequence. In contrast, PrefixSpan needs to pass every items in the projected database no matter they are the same as before or not. So LAPIN_Suffix will save much time because our search space is only a subset of the one in Pre-

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

1. For each customer sequence F
2. $S_{\alpha,F} \leftarrow$ get prefix border position of F in S_α
3. $L_{s,F} \leftarrow$ get 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] $\geq min_sup$)
10. $FreItem_s$.insert(β);

Fig. 6: Finding SE frequent patterns in LAPIN_Suffix

fixSpan. The pseudo code of LAPIN_Suffix is shown as Figure 6.

Space Optimization of LAPIN_Suffix. For common case, although the *item-last-position list* (assumed as L) is much smaller than the original database, it is space consuming to construct it. As an optimization, we can combine L and the item position lists of DB together. It can be done because in each item position list, the last column which is non-Null position number of each item is in fact the item's last position. We need to sort the position lists by their last element in each customer sequence in ascending order.

Table 4: *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	$(ac) : 3 \rightarrow null$
	$(bc) : 2 \rightarrow null$
	$(cd) : 4 \rightarrow null$

2.2.3 I-Step of LAPIN

In LAPIN, the *I-Step* is similar to the *S-Step*. From the step 1 of Figure 2, we can get the frequent 2-length IE sequence position list as shown in Table 4 and the *I-Step item-last-position list*. In the step 4 of Figure 2, we first get the position list of the last 2-length IE item of α, then do a binary search in Table 4. Here we look for the first position which is equal to or larger than the (k-1)-length prefix border position (in STL, the function is named *lower_bound*). To find the frequent (k+1)-length IE sequences in the step 6 of Figure 2, similar to S-Step, we have two classes of algorithms, one is *LCI-oriented* which directly compares the last positions of 2-length IE sequence with the prefix border positions to judge whether the frequent k-length sequence could be appended with the 2-length IE sequence, to be 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 is $Suffix$-*oriented* which uses the *item-last-position list* to facilitate the *I-Step* support counting.

Table 5: Parameters used in datasets generation

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

3 Experimental Evaluation

In this section, we describe our experiments and evaluations conducted on both synthetic data and real data and compare LAPIN with PrefixSpan and SPAM to demonstrate the efficiency of the proposed algorithms. We perform the experiments on a 1.6GHz Intel Pentium(R)M PC machine with 1G memory, running Microsoft Windows XP. We obtained the executable file of PrefixSpan from Jiawei Han, and the source code of SPAM from Jay Ayres. All three algorithms are written in C++, and compiled in MS Visual C++ environment. The output of the programs was turned off to make the comparison fair.

3.1 Comparing PrefixSpan with LAPIN Algorithms

We first compare PrefixSpan and our algorithms on synthetic and real datasets, from where show that LAPINs outperform PrefixSpan by up to near an order of magnitude on datasets with long patterns and low minimum support.

3.1.1 Synthetic Data

The synthetic datasets are generated by the IBM data generator described in [2]. The meaning of the different parameters used to generate the datasets is shown in Table 5. In the first experiment, we compared PrefixSpan and our algorithms on several small, medium and large size of datasets for various minimum support. The statistics of these datasets is shown in Figure 7 (a).

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

(a) Dataset characteristics

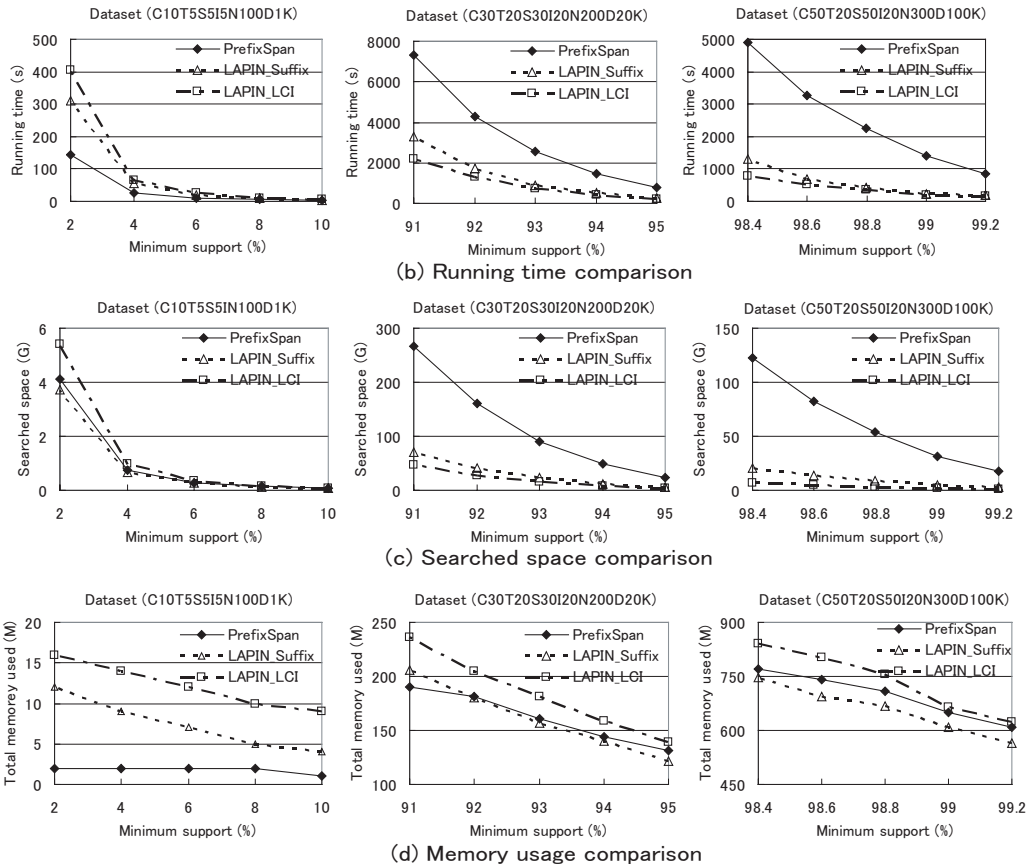


Fig. 7: Different size of the datasets

PrefixSpan V.S. LAPINs: We define *search space* as in PrefixSpan the size of the projected DB, denoted as S_{ps} and in LAPINs the sum of the number of different items for each customer sequences in the suffix (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI), denoted as S_{lapin}. Figure 7 (b) and Figure 7 (c) show the running time and the searched space comparison between PrefixSpan and LAPINs, which clearly illustrates the reason why PrefixSpan is slower than LAPINs on the medium (C30T20S30I20N200D20K) and the large datasets (C50T20S50I20N300D100K), because the searched space of the former is much larger than that of the latter. For the small dataset (C10T5S5I5N100D1K), the ineffectiveness of searched space saving and the initial overhead needed to set up let LAPINs slower than PrefixSpan. Overall, our runtime tests show that LAPINs excels at finding the frequent sequences for many different types of large datasets. From Figure 7 (b) and Figure 7 (c), we have:

$$T_{ps}/T_{lapin} \propto S_{ps}/S_{lapin} \tag{1}$$

Let D be the number of customers in the database, L the average sequence length of the projected DB, n the average total number of the distinct items in the projected DB (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI), and m the distinct item recurrence rate of the projected DB (LAPIN_Suffix) or of the local candidate list (LAPIN_LCI), then $m{=}L/n$ $(m \geq 1)$. So we have $S_{ps}/S_{lapin} = (D * L)/(D * n) = (D * L)/(D * L/m) = m$. Formula (1) could be changed to:

$$T_{ps}/T_{lapin} \propto m \tag{2}$$

Formula (2) illustrates that as the distinct item recurrence rate m increases, our LAPINs will become faster than PrefixSpan. For example, suppose the prefix sequence is a and there are one hundred item b exist in the projected DB for a customer sequence, in PrefixSpan it needs to scan one hundred times on item b but in LAPIN it only needs once to judge whether the item b could be appended to a or not. Although by the bi-level optimization strategy, PrefixSpan can judge the three candidate sequences $\langle ab \rangle$, $\langle ba \rangle$, and $\langle (ab) \rangle$ during the one hundred times scanning, in LAPINs

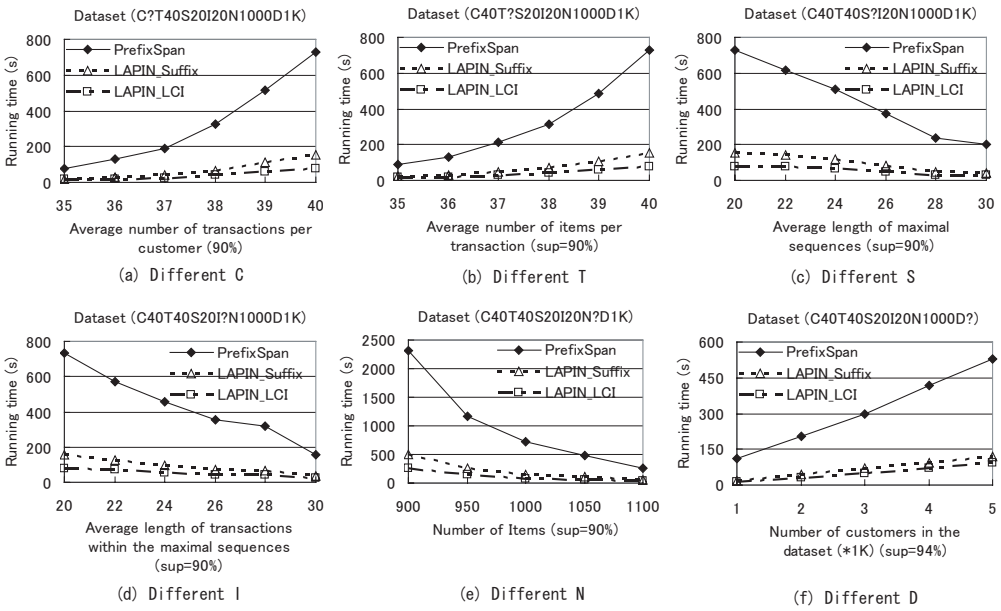


Fig. 8: Varying parameters of the datasets

we can get the same result by only scanning three times. Hence, scanning most of the duplicate items in the projected DB is useless and time consuming. As Formula (2) describes, the more m is, the faster of LAPINs compared with PrefixSpan.

LAPIN_Suffix V.S. LAPIN_LCI: Because LAPIN_Suffix and LAPIN_LCI are implemented in the same framework, in addition to a little difference at the initial phase, the only implementation difference is at the support counting step: LAPIN_Suffix searches in the suffix while LAPIN_LCI searches in the local candidate item list. We have:

$$T_{Suffix}/T_{LCI} \approx S_{Suffix}/S_{LCI} = m_{Suffix}/m_{LCI} \quad (3)$$

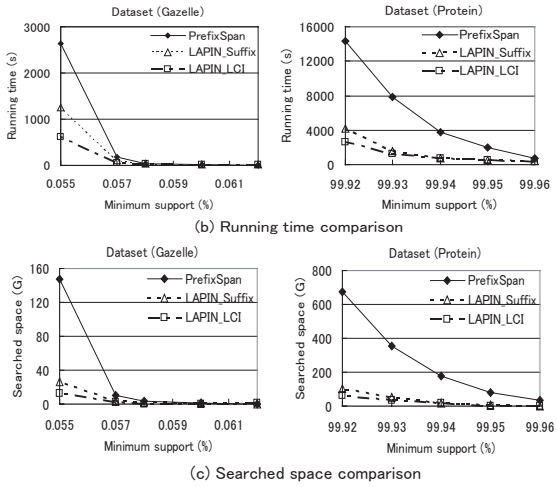
where m_{Suffix} is the distinct item recurrence rate of the projected DB, and m_{LCI} is the distinct item recurrence rate of the local candidate item list. Formula (3) can clearly explain the relationship of LAPIN_Suffix and LAPIN_LCI, which is shown in Figure 7 (b) and Figure 7 (c). LAPIN_Suffix is faster than LAPIN_LCI on the small dataset because the former searches smaller space than the latter does. While for the medium and the large datasets which have many long patterns, LAPIN_LCI is faster than the LAPIN_Suffix because the situation is reverse.

Space analysis: As Figure 7 (d) shown, LAPIN_Suffix spends almost the same memory as PrefixSpan does, except on small dataset for storing some initialization information. For LAPIN_LCI, because it needs to store the item last position information in bit vector format, it needs more space than LAPIN_Suffix and PrefixSpan. Let C' 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 7, we can know that there is a trade-off between LAPIN_Suffix and LAPIN_LCI on considering speed and space.

Different parameter analysis: In the second experiment, we investigate the different parameters used to generate the dataset on the effect of the performance. As Figure 8 shown, when the average number of transactions per customer (C) increases and the average number of items per transaction (T) increases, and when the number of items (N) decreases, the performance of LAPINs improves even further relative to comparing with PrefixSpan, up to an order of magnitude. Consider Formula (2), $m{=}L/n{=}\bar{C} * \bar{T}/\bar{N}$, where $\bar{C}$ is the average number of transactions per customer in the projected DB, $\bar{T}$ is the average number of items per transaction in the projected DB, and $\bar{N}$ is the average number of different items in the projected DB (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI). Keep other parameters static, increasing C, T and decreasing N respectively will result to the increase of the distinct item recurrence rate m, which fits to the experimental result shown in Figure 8. It confirms the correctness of Formula (2). For the other three parameters, as the average length of maximal sequences (S) decreases and average length of transactions within the maximal sequences (I) decreases, and as the number of customers (D) increases, although LAPINs continues to outperform PrefixSpan, the discrepancy between the running times did not increase significantly because these parameters do not apparently contribute to the increase of the distinct item recurrence rate m.

Dataset	#Seq	#Items	Min. len.	Max. len.	Avg. len.
Gazelle	59602	497	1	267	2.5
Protein	116142	24	400	600	482

(a) Dataset characteristics



(b) Running time comparison

(c) Searched space comparison

Fig. 9: Real datasets

3.1.2 Real Data

We think that results test on real data will be more convincing to demonstrate the efficiency of the proposed algorithm. In this section, we test on two real datasets called Gazelle and Protein. A portion of the Gazelle was used in KDD-Cup 2000. More detailed information about this dataset can be found in [11]. The second real dataset Protein was extracted from the web site of National Center for Biotechnology Information [12]. It was extracted by the 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 datasets is given in Figure 9 (a). It can be seen that the Gazelle is a sparse small dataset and the Protein is a dense large dataset. As shown in Figure 9 (b), LAPINs outperform PrefixSpan on Gazelle and Protein datasets. The reason why LAPIN performs so well is similar to that for synthetic datasets in Section 3.1.1, based on the searched space saving as shown in Figure 9 (c). This experiment confirmed the superiority of the proposed methods on real life data.

4 Conclusion

In this paper, we propose a novel series of algorithms called LAPIN for efficient sequential pattern mining. Our main idea is that the last position of an item s in each customer sequence is very useful and key to judge whether a k-length frequent sequence could grow to a frequent (k+1)-length sequence by appending it with s. So LAPIN could reduce searching greatly by only scanning a small portion of the projected database or the ID-List as well as handle long pattern efficiently, which is inherently difficult for most existing algorithms. By thorough experiments and comparison, we demonstrated that LAPINs outperform PrefixSpan up to an order of magnitude. Our experimental results also show that LAPIN is very efficient not only on synthetic data but also on real world data such as web access pattern and DNA sequence.

We plan to continue our work by applying the proposed algorithm to other application domains and investigate how to extend it to discover closed frequent sequence.

References

[1] J.Ayres, J.Flannick, J.Gehrke, and T.Yiu, "Sequential Pattern Mining using A Bitmap Representation", In *ACM SIGKDD Conference*, pp.429-435, Alberta, Canada, Jul. 2002.

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

[3] R. Agrawal, R. Srikant: "Fast Algorithms for Mining Association Rules", In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, Sep. 1994.

[4] R. Agrawal, R. Srikant: "Mining sequential patterns.", In *Proc. of the Int'l Conference of Data Engineering (ICDE'95)*, pp.3-14, Taipei, Taiwan, Mar. 1995.

[5] M.J.Zaki, "SPADE:An Efficient Algorithm for Mining Frequent Sequences",In *Machine Learning Journal*, Vol 42, No.1/2, 2001.

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

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

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

[9] D.Chiu, Y.Wu, A.L.P.Chen, "An Efficient Algorithm for mining Frequent Sequences by a New Strategy without Support Counting", In *20th Intl. Conf. Data Engineering*, Mar. 2004.

[10] X.Yan, J.Han, and R.Afshar, "CloSpan: Mining closed sequential patterns in large datasets", In *Proc. 2003 SIAM Int. Conf. Data Mining (SDM'03)*, San Fransisco, CA, May 2003.

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

[12] http://www.ncbi.nlm.nih.gov.