Effective Algorithms for Sequential Pattern Mining

Zhenglu YANG[♥] Yitong WANG[♦]
Masaru KITSUREGAWA [♠]

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 by up to an order of magnitude on long pattern dense data sets.

1. Introduction

Sequential pattern mining, which extracts frequent subsequences from a sequence database, 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 [4]. From then on, much work has been carried out on mining frequent patterns, as for example, in [1][2][3][5]. 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 [2] 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.

Table 1: Sequence Database

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

Let our running database be the sequence database *S* shown in Table 1 with min_support = 2. 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<ac> is contained in the sequence 10, 20, and 30, respectively, and its support is equal to 3. Therefore, <ac> is a frequent pattern.

1.1 Overview of Our Algorithm

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 1. 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 <a>, 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 <a>'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 <a>:1, <b>:2, <c>:3, and <d>:3, from which, we can determine that <ab>, <ac> and<ad> are the frequent patterns. 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 <a>, 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 <(ab)>:1, and<(ac)>:2. We can also obtain the support of the 3-length sequences <a(bc)>:1, <a(bd)>:1, and <a(cd)>:1, which is similar to the bi-level strategy of PrefixSpan, but we avoid scanning the entire projected database.

[♥] Student Member Institute of Industrial Science, the University of Tokyo
yangzl@tkl.iis.u-tokyo.ac.jp
[♦]Computer Science Department, Fudan Univerisy, China
yitongw@fudan.edu.cn
[♠] Regular Member Institute of Industrial Science, the University of Tokyo
kitsure@tkl.iis.u-tokyo.ac.jp

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

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. Let $\overline{D}$ be the average number of customers (i.e., sequences) in the projected DB, $\overline{L}$ be the average sequence length in the projected DB, $\overline{N}$ 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 $m=\overline{L}/\overline{N}$ ($m\geq 1$), and the relationship between the runtime of PrefixSpan (T_{ps}) and the runtime of LAPIN (T_{lapin}) in the support counting part is

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

Because support counting is usually the most costly step in the entire mining process, Eq. (1) illustrates the main reason why our LAPIN algorithm is faster than PrefixSpan for dense data sets, whose m (density) can be very high.

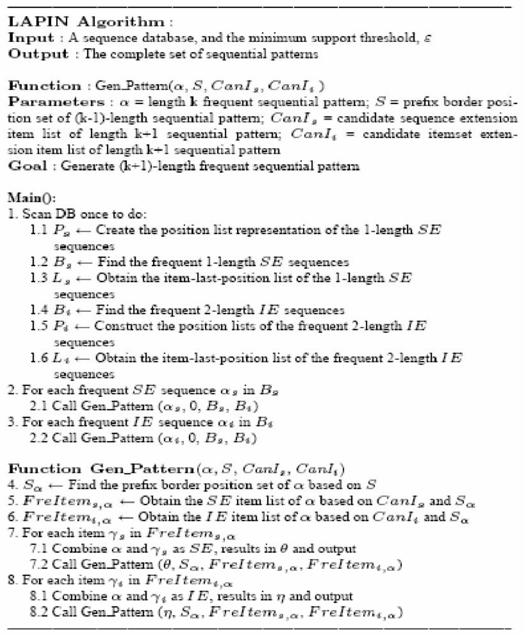
The remainder of this paper is organized as follows. In Section 2, we introduce a series of LAPIN algorithms in detail. Our experimental results and performance analysis are reported in Section 3. We conclude the paper in Section 4.

2. LAPIN: Design and Implementation

In this section, we describe the LAPIN algorithms in detail. We use a lexicographic tree [1] as the search path of LAPIN and adopt a lexicographic order [1], which employs the Depth First Search (DFS) strategy. The pseudo code of LAPIN is shown in Fig. 1.

As Example 1 in Section 1.1 shows, the I-Step methodology is similar to the S-Step methodology in LAPIN. We will first describe the S-Step process. In Step 1, by scanning the DB once, we obtain the *SE* position list table and all the 1-length frequent patterns. Based on the last element in each position list, we sort and construct the *SE item-last-position list* in ascending order, as shown in Table 2.

In function *Gen_Pattern*, to find the prefix border position set of k-length α (Step 4), we first obtain the



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

Function : Gen_Pattern($\alpha, S, CanI_s, CanI_i$)
Parameters : $\alpha$ = 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 $\alpha_s$ in $B_s$
    2.1 Call Gen_Pattern ($\alpha_s$, 0, $B_s$, $B_i$)
3. For each frequent $IE$ sequence $\alpha_i$ in $B_i$
    2.2 Call Gen_Pattern ($\alpha_i$, 0, $B_s$, $B_i$)

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

Figure 1. LAPIN Algorithm pseudo code

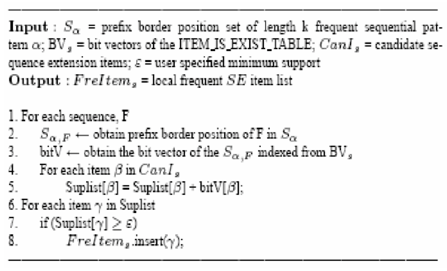
position list of the last item of α, and then perform a binary search in the list for the (k-1)-length prefix border position. For *S-Step*, we look for the first position that is larger than the (k-1)-length prefix border position.

Step 5, shown in Fig. 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 in [6]. 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 [3]. Another choice is to test the candidate item in the projected DB, just as PrefixSpan [2] does (*Suffix-oriented*). The correctness of these methods was discussed in [2] and [3], respectively.

We find that *LCI-oriented* and *Suffix-oriented* have their own advantages for different types of data sets [6]. Based on this discovery, we propose two algorithms categorized into two classes. One class is *LCI-oriented*, *LAPIN_LCI*, and the other class is *Suffix-oriented*, *LAPIN _Suffix*.

2.1 LAPIN LCI

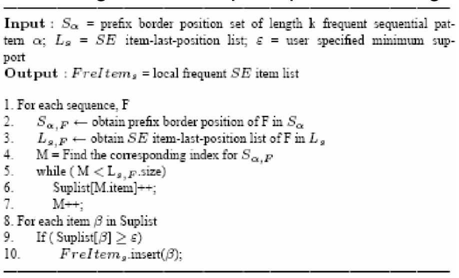
LAPIN_LCI tests each item which is in the local candidate item list. In each customer sequence, it directly judges whether an item can be appended to the prefix sequence or not by comparing this item's last position with the prefix border position. Increment the support value of the candidate item by 1 if the candidate item's last position is larger than the prefix border position. As an optimization, we use bitmap strategy to avoid such comparison process. A pre-constructed table,



```
Input : $S_\alpha$ = prefix border position set of length k frequent sequential pattern $\alpha$; $BV_s$ = bit vectors of the ITEM_IS_EXIST_TABLE; $CanI_s$ = candidate sequence extension items; $\varepsilon$ = 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 $S_\alpha$
3.     bitV $\leftarrow$ obtain the bit vector of the $S_{\alpha,F}$ indexed from $BV_s$
4.     For each item $\beta$ in $CanI_s$
5.         Suplist[$\beta$] = Suplist[$\beta$] + bitV[$\beta$];
6. For each item $\gamma$ in Suplist
7.     if (Suplist[$\gamma$] $\geq \varepsilon$)
8.         $FreItem_s$.insert($\gamma$);
```

Figure 2. Finding the SE frequent patterns using LAPIN_LCI



```
Input : $S_\alpha$ = prefix border position set of length k frequent sequential pattern $\alpha$; $L_s$ = $SE$ item-last-position list; $\varepsilon$ = 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 $S_\alpha$
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 $\beta$ in Suplist
9.     If ( Suplist[$\beta$] $\geq \varepsilon$)
10.        $FreItem_s$.insert($\beta$);
```

Figure 3. Finding the SE frequent patterns using LAPIN_Suffix

named ITEM_IS_EXIST_TABLE is constructed while first scanning to record the last position information. In the table, we use a bit vector to represent all the 1-length frequent items existing for a specific position. 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.

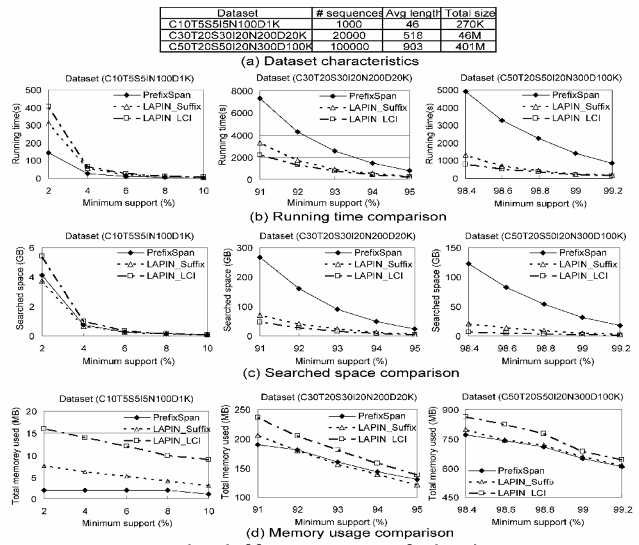
Space Optimization of LAPIN_LCI. We find that only part of the table is useful, and that most is not. The useful information is stored in some key positions' lines, which indicate the last positions of the 1-length frequent items (except the last one). Hence, we only store these vectors. The pseudo code of LAPIN_LCI is shown in Fig. 2. Please refer [6] for detail.

Example 2. Let us assume that we have obtained the prefix border position set of the pattern <a> in Table 1, i.e., (1,3,3). We also know that the *local candidate item list* is (a, b, c, d). Then, we can obtain the bit vector mapped from the specific position, which are 1111, 0111, and 0011 with respect to the pattern <a>'s prefix border position set, (1,3,3), and accumulate them, resulting in <a>:1, <b>:2, <c>:3, and <d>:3. From here, we can deduce that <ab>, <ac>, and <ad> are frequent patterns.

2.2 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. Therefore, we propose a new algorithm, LAPIN_Suffix. In the *item-last-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). The pseudo code of LAPIN_Suffix is shown in Fig. 3. Example 1 in Section 1.1 describes the flow of LAPIN_Suffix.

Note that in LAPIN, the *I-Step* is similar to the *S-Step*. Due to limited space, we do not describe it here. Interested readers can refer [6] for detail.



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

Figure 4. The different sizes of the data sets

3. Performance Study

We perform the experiments using a 1.6 GHz Intel Pentium(R)M PC machine with a 1 G memory, running WinXP. We conducted experiments on both synthetic and real datasets. However, due to space limitation, we will only report results on synthetic data here.

The synthetic data sets are generated by an IBM data generator [4]. We first test on different sized data sets for various minimum supports. The statistics of these data sets is shown in Fig. 4(a).

PrefixSpan vs. LAPIN. We define *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}. Fig. 4(b) and Fig. 4(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 and the large data set. This is because the searched spaces of the two data sets in PrefixSpan are much larger than that in LAPIN. For the small data set, the ineffectiveness of searched space saving and the initial overhead needed to set up meant that LAPIN is slower than PrefixSpan. Overall, our runtime tests show that LAPIN excels at finding the frequent sequences for many different types of large data sets.

Eq.(1) in Section 1.1 illustrates the relationship between the runtime of PrefixSpan and that of LAPIN in the support counting part, which also approximately expresses the relationship between the entire mining time

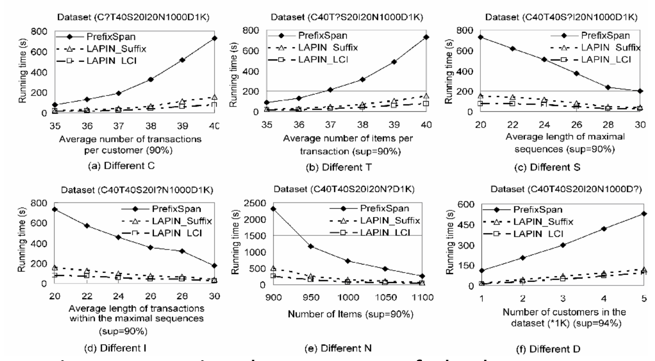


Figure 5. Varying the parameters of the data sets

of PrefixSpan and that of LAPIN because support counting is usually the most costly step in the entire mining process. Eq.(1) illustrates that, the higher the value of *m* is, the faster LAPIN becomes compared to PrefixSpan. The experimental data shown in Fig. 4(c) and Fig. 4(c) is in accordance with our theoretical analysis.

LAPIN_Suffix vs. LAPIN_LCI. The main difference of LAPIN_Suffix and LAPIN_LCI is in the support counting phase: LAPIN_Suffix searches in the suffix, whereas LAPIN_LCI searches in the local candidate item list. Let m_{Suffix} be the distinct item recurrence rate of the projected DB, m_{LCI} be the distinct item recurrence rate of the local candidate item list. The relationship between the entire mining time of LAPI_Suffix (T_{Suffix}) and that of LAPIN_LCI (T_{LCI}) is as

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

Eq.(2) is in accordance with the experimental data shown in Fig. 4(b) and Fig. 4(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, LAPIN_LCI is faster than LAPIN_Suffix because the situation is reversed.

Memory usage analysis. As Fig. 4(d) shows, LAPIN_Suffix expends almost the same amount of memory as PrefixSpan does, except for small data sets because LAPIN_Suffix uses 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 (DC'N)/8 bytes to store the last position information for all the items. From Fig. 4, 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. Please refer [4] for the meaning of the different parameters. As Fig. 5 shows, when *C* increases, *T* increases, and *N* decreases, the performance of LAPIN improves even more relative to PrefixSpan. Let us consider Eq.(1), $m = \overline{L} / \overline{N} = \overline{C}\ \overline{T} / \overline{N}$, where $\overline{C}$ is the average number of transactions per customer in the projected DB, and $\overline{T}$ 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 Fig. 5. With regards to the other parameters, the discrepancy between the running times does not change significantly because these parameters do not contribute to the variance of the distinct item recurrence rate, *m*. For LAPIN_LCI and LAPIN_Suffix, the former is always the fastest because its searched space is less than that of the latter.

4. Conclusion

In this work, we have proposed novel algorithms for efficient sequential pattern mining. 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. 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.

[References]

[1] J.Ayres, J.Flannick, J.Gehrke, and T.Yiu, “Sequential Pattern Mining using A Bitmap Representation,” In KDD, pp. 429-435, 2002.
[2] J.Pei, J.Han, M.A.Behzad, and H.Pinto, “PrefixSpan:Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,”In ICDE, 2001.
[3] M. J. Zaki, “SPADE: An Efficient Algorithm for Mining Frequent Sequences,” In Machine Learning, Vol. 40, pp. 31-60, 2001.
[4] R. Agrawal and R. Srikant, “Mining sequential patterns,” In ICDE, pp. 3-14, 1995.
[5] R.Srikant and R.Agrawal,“Mining sequential patterns: Generalizations and performance improvements,” In EDBT, pp. 13-17, 1996.
[6] Z. Yang, Y. Wang, and M. Kitsuregawa. LAPIN: Effective Sequential Pattern Mining Algorithms by Last Position Induction. Technical Report, Tokyo University, 2005.
http://www.tkl.iis.u-tokyo.ac.jp/»yangzl/Docu-ment/LAPIN.pdf

Zhenglu YANG
Ph.D student of Graduate School of Information Science and Technology, the University of Tokyo. He received the Master degree in the above graduate school. His research interests include sequence mining and data mining.

Yitong WANG
Associate professor at Fudan University, China. She received Ph.D degree in computer science in 1999 from the above university. Her research interests include data clustering and data mining.

Masaru KITSUREGAWA
Professor and the director of center for information at Institute of Industrial Science, the University of Tokyo. He received the Ph.D degree in information engineering in 1983 from the University of Tokyo. His research interests include parallel processing and database engineering. He is a member of steering committee of IEEE ICDE, PAKDD and WAIM, and has been a trustee of the VLDB Endowment. He was the chair of data engineering special interest group of Institute of Electronic, Information, Communication, Engineering, Japan, the chair of ACM SIGMOD Japan, Chapter. He is currently a trustee of DBSJ.