PAID: Mining Sequential Patterns by Passed Item Deduction in Large Databases

Zhenglu Yang Masaru Kitsuregawa
Info. and Comm. Engineering Department
The University of Tokyo
Tokyo, Japan
{yangzl, kitsure}@tkl.iis.u-tokyo.ac.jp

Yitong Wang
Computer Science Department
Fudan university
Shanghai, China
yitongw@fudan.edu.cn

Abstract

Sequential pattern mining is very important because it is the basis of many applications. Yet how to efficiently implement the mining is difficult due to the inherent characteristic of the problem - the large size of the dataset. Although there has been a great deal of effort on sequential pattern mining in recent years, its performance is still far from satisfactory. In this paper, we have proposed a new algorithm called PAssed Item Deduced sequential pattern mining (abbreviated as PAID), which can efficiently get all the frequent sequential patterns from a large database. The main difference between our strategy and the existing works is that other algorithms accumulate the candidate support in each iteration from scratch, in contrast, PAID makes good use of the temporary results (support value) of k-length frequent patterns on discovering (k+1)-length patterns, which can reduce the search space greatly in mining sequential patterns. Our experimental results and performance studies show that PAID outperforms the previous works by meaningful margins on large datasets.

1 Introduction

Sequential pattern mining [2], which can be seen as association rule discovery [1] with regard to time constraint, 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 biological data analysis. Its main goal is to extract frequent subsequences from a sequence database. One typical example is:

From an electronic goods store's transaction list, we know that 70% of the people who buy a television also buy a video camera within one month.

In the above example, we need not only know the two purchased items that are always bought by the same customer (association rule), the most important thing is that we need also know the order in which those items are bought (sequential pattern). 70% here represents the percentage of customers who have this purchasing habit and can be used to predict customer's behavior in the near future.

Efficient sequential pattern mining methodologies have been studied extensively in many related problems, including the general sequential pattern mining [2] [13] [18] [11] [3] [4], constraint-based sequential pattern mining [5], incremental sequential pattern mining [10], frequent episode mining [9], approximate sequential pattern mining [7], partial periodic pattern mining [6], temporal pattern mining in data stream [14], maximal and closed sequential pattern mining [8] [16] [15].

Although there are so many problems related to sequential pattern mining explored, we realize that the general sequential pattern mining algorithm development is the most basic one because all the others can benefit from the strategies it employs, i.e., Apriori heuristic and projection-based pattern growth. Hence we aim to develop an efficient general sequential pattern mining algorithm in this paper. Much work has been carried out on it [2] [13] [18] [11] [3] [4]. However, all of these works suffer from the problems of having a large search space and the ineffectiveness in handling long patterns on large datasets. In this work, we propose a new algorithm to reduce the space necessary to be searched. Instead of counting the support of each candidate sequence from scratch in every iteration, as PrefixSpan [11] and SPADE [18] do, we get the support of (k+1)-length candidate sequence from already found k-length frequent pattern's support, which scans much smaller space than other algorithms do on large datasets. Since support counting is usually the most costly step in sequential pattern mining, the PAssed Item Deduction (PAID) technique improves the performance greatly by avoiding cost scanning.

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., $(t_j \subseteq I)$ for $1 \leq$ j $\leq l$. 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 $1 \leq$ 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 exists 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 denote s_a as a $subsequence$ of s_b, and s_b as 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

Table 1. Sequence Database

SID	Sequence
10	$bdbcbdabad$
20	$dcaabcbdab$
30	$cadadcadca$

concatenates with α, which has two possible forms, such as $Itemset\ Extension$ (IE), $s \diamond \alpha$=$\langle s_1, s_2, \dots, s_l \cup \{\alpha\}\rangle$, or $Sequence\ Extension$ (SE), $s \diamond \alpha$=$\langle s_1, s_2, \dots, s_l, \{\alpha\}\rangle$. 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 $support(\beta) \geq \varepsilon$. In this work, the objective was to find the complete set of sequential patterns of database S in an efficient manner.

Example 1. Let our running database be the sequence database S shown in Table 1 with min_support = 2. We will use this database throughout the paper. This database only covers the $Sequence\ Extension$ (SE) case because we want to clearly illustrate our key idea by this simplified example. In our implementation, we can also deal with the $Itemset\ Extension$ (IE) case, which will be explained at the end of Section 2.

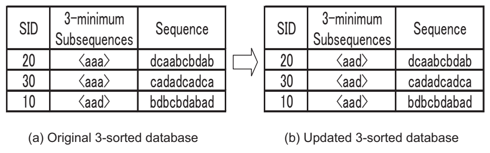
1.2 Related Work

Agrawal and Srikant first proposed the sequential pattern mining problem in [2]. Then the same authors proposed the GSP algorithm [13], which iteratively generates candidate (k+1)-length sequences from frequent k-length sequences based on the anti-monotone property that all the subsequences of a frequent sequence must be frequent.

Zaki et al. proposed SPADE [18] to elucidate frequent sequences using efficient lattice search techniques and simple join operations based on the ID-List data structure. An ID-list of a sequence keeps a list of pairs, which indicate the positions that it appears in the database. In a pair, the first value stands for a customer sequence and the second refers to an element (position) in it. For the example database in Table 1, the ID-list of sequence ⟨ab⟩ is ⟨(10,8), (20,5), (20,7), (20,10)⟩, where the pair (10,8) means that this sequence appears in customer sequence 10 and ends in the eighth element. SPADE computes the support count of a candidate k-sequence generated by merging the ID-lists of any two frequent (k-1)-sequences with the same (k-2)-prefix. Consider the same database in Table 1. To compute the support count of sequence ⟨abc⟩, the SPADE algorithm merges the two ID-lists of sequences ⟨ab⟩ and ⟨ac⟩, which are ⟨(10,8), (20,5), (20,7), (20,10))⟩ and ⟨(20,6), (30,6), (30,9)⟩ respectively. As a result, the ID-list of sequence ⟨abc⟩ is ⟨(20,6)⟩, indicating that this sequence appears in customer sequence 20 and therefore has a support count of 1. Ayres et al. proposed the SPAM algorithm [3], which uses SPADE's lattice concept, but represents each ID-List as a vertical bitmap. All the above algorithms followed the Apriori heuristic.

Table 2. ⟨a⟩-Projected Database

SID	Sequence
10	bad
20	$abcbdab$
30	$dadcadca$



SID	3-minimum Subsequences	Sequence
20	⟨aaa⟩	dcaabcbdab
30	⟨aaa⟩	cadadcadca
10	⟨aad⟩	bdbcbdabad

(a) Original 3-sorted database

SID	3-minimum Subsequences	Sequence
20	⟨aad⟩	dcaabcbdab
30	⟨aad⟩	cadadcadca
10	⟨aad⟩	bdbcbdabad

(b) Updated 3-sorted database

Figure 1. The 3-sorted database in the DISC algorithm

On the other hand, Pei et al. proposed a projection-based algorithm, PrefixSpan [11], which projects sequences into different groups called $projected\ databases$. For the example database in Table 1, assuming min_support=2, the PrefixSpan algorithm first scans the database to find the frequent 1-sequences, i.e. ⟨a⟩, ⟨b⟩, ⟨c⟩, ⟨d⟩. Then the algorithm constructs the projected database for each already found frequent 1-sequence. For example, Table 2 shows the projected database of ⟨a⟩. The PrefixSpan algorithm continues the discovery of frequent 1-sequences in Table 2 to form the frequent 2-sequences with prefix ⟨a⟩. Here by scanning Table 2 it gets 1-length sequences with their support, i.e. $\langle a\rangle : 3$, $\langle b\rangle : 2$, $\langle c\rangle : 2$ and $\langle d\rangle : 3$, indicating that the supports of 2-length sequences are ⟨aa⟩ : 3, ⟨ab⟩ : 2, ⟨ac⟩ : 2, ⟨ad⟩ : 3. Recursively, the PrefixSpan algorithm generates the projected database for each frequent k-sequence to find frequent (k+1)-sequences. PrefixSpan uses a lot of time because it needs to scan the entire projected database, which can be very large.

Chiu et al. proposed the DISC strategy [4], which uses a k-sorted database to find all the frequent k-sequences and skips most non-frequent k-sequences by checking only the conditional k-minimum subsequences. For the example database in Table 1, to find frequent 3-sequence patterns, the DISC strategy first finds the 3-minimum subsequences of each customer sequence and then, sorts the sequences according to the ascending order of these 3-minimum subsequences, as shown in Fig. 1 (a). The k-minimum subsequence at the θ-th position is denoted by α_θ. Assuming min_support = 3, by comparing α_1, i.e. ⟨aaa⟩, and $\alpha_{min\_support}$, i.e. ⟨aad⟩, it not only knows that ⟨aaa⟩ is non-frequent, but also knows that all the subsequences between ⟨aaa⟩ and ⟨aad⟩ are not frequent. By this way, DISC algorithm can skip many non-frequent candidate subsequences. It continues to test the remaining 3-subsequences by updating α_1 and α_2 in customer sequences 20 and 30, respectively, to the smallest one which is equal to or larger than $\alpha_{min\_support}$. The result is shown in Fig. 1 (b), from where we know that ⟨aad⟩ is frequent. Following this strategy, DISC continues to update the 3-minimum subsequences for each candidate customer sequence. It terminates the 3-subsequences testing process if the largest 3-minimum subsequence has been traversed. The updating process in DISC, is indeed searching in the (k-1)-prefix projected database, which is similar to the mining process of PrefixSpan.

We noticed that the last position of an item is very important to judge whether a k-length pattern could grow to

Table 3. Item Last Position Table

SID	Last Position of Item
10	$c_{last} = 4$ $b_{last} = 8$ $a_{last} = 9$ $d_{last} = 10$
20	$c_{last} = 6$ $d_{last} = 8$ $a_{last} = 9$ $b_{last} = 10$
30	$d_{last} = 8$ $c_{last} = 9$ $a_{last} = 10$

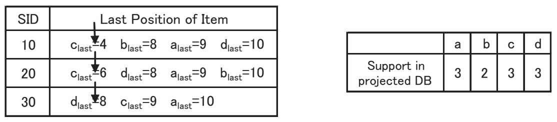
a (k+1)-length pattern. With this finding, we proposed our work in LAPIN [17], which could improve the efficiency significantly by reducing search space greatly. For the example database in Table 1, similar to the PrefixSpan algorithm, LAPIN first scans the database to find the frequent 1-sequences and constructs the item last position table as shown in Table 3. To find those 2-sequence whose common prefix is ⟨a⟩, LAPIN only scans the last position of each distinct item in ⟨a⟩-projected database. Here LAPIN spends 11 scanning times, while PrefixSpan requires 18 scanning times to get the same result. However, this improvement is at the price of much memory consuming when building the list of item's last position because LAPIN uses a bitmap strategy [17]. This problem motivates the work in this paper. We aim to obtain an efficient and balanced pattern mining algorithm with low memory consuming.

All of the above algorithms count the support of a candidate sequence from scratch. Different from them, we have found a methodology that can make good use of the supports of already found k-length frequent patterns, to count the supports of (k+1)-length candidate sequences. Although there are some other algorithms were also built based on reuse strategy, such as ISM [10], its mining object is the updated database, not the original database. To the best of our knowledge, the proposed PAID algorithm in this paper is the first one to make good use of not only the position of item but also the intermediate value (support value) of k-length pattern when fining (k+1)-length pattern.

1.3 Overview of Our Algorithm

Our algorithm follows the pattern growth strategy, which finds (k+1)-length patterns based on k-length frequent patterns. However, instead of counting candidates support from scratch as most existing algorithms do, PAID makes good use of the intermediate result (support value) of k-length frequent patterns in order to count supports of (k+1)-length candidate sequences.

Discovering (k+1)-length frequent patterns. For any sequence 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. This strategy is named LAPIN, as described in [17]. Moreover, we can deduce which items "disappear" after growing k-length sequence to (k+1)-length sequence, based on their last positions are larger than the (k+1)-length sequence border position or not. Then, we can reduce the supports of these "disappeared" items from already found results (support value) of k-length frequent patterns to get (k+1)-length candidate sequence' supports, avoiding to scan k-length frequent pattern's projected database, which is probably very large. Because the reuse strategy

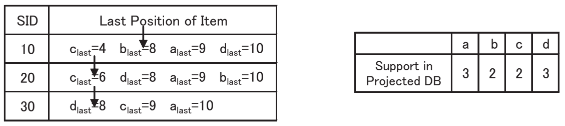


SID	Last Position of Item
10	c_{last}=4 b_{last}=8 a_{last}=9 d_{last}=10
20	c_{last}=6 d_{last}=8 a_{last}=9 b_{last}=10
30	d_{last}=8 c_{last}=9 a_{last}=10

	a	b	c	d
Support in projected DB	3	2	3	3

(a) Indices in Last Position of Item Table (b) Support in ⟨ ⟩-projected DB

Figure 2. Prefix sequence is ⟨ ⟩



SID	Last Position of Item
10	c_{last}=4 b_{last}=8 a_{last}=9 d_{last}=10
20	c_{last}=6 d_{last}=8 a_{last}=9 b_{last}=10
30	d_{last}=8 c_{last}=9 a_{last}=10

	a	b	c	d
Support in Projected DB	3	2	2	3

(a) Indices in Last Position of Item Table (b) Support in ⟨a⟩-projected DB

Figure 3. Prefix sequence is ⟨ a ⟩

in this algorithm is based on passed (disappeared) items, we named this algorithm as PAID (PAssed Item Deduction).

Example 2. When scanning the database in Table 1 for the first time, we obtain Fig. 2 (a), which is a list of the last positions of the 1-length frequent sequences in ascending order. "↓" means the first index whose position is larger than the position of the last item in the corresponding prefix sequence. The initialization value for these indices is "1", whose corresponding prefix is ⟨ ⟩. We can also get 1-length frequent sequences with their support, $\langle a \rangle$: 3, $\langle b \rangle$: 2, $\langle c \rangle$: 3, and $\langle d \rangle$: 3, as shown in Fig. 2 (b), which is indeed the support count of each item in the ⟨ ⟩-projected DB.

Following the Depth First Search (DFS) path, to find the 2-length frequent patterns whose common prefix is item a, we first get item a's positions in Table 1 are 10:7, 20:3, 30:2, where sid:eid represents the sequence ID and the element ID. Then, we check Fig. 2 (a) to obtain the first indices whose positions are larger than $\langle a \rangle$'s, resulting in 10:2, 20:1, 30:1, i.e. (10:b$_{last}$ = 8, 20:c$_{last}$ = 6, and 30:d$_{last}$ = 8), as shown in Fig. 3 (a). From Fig. 2 (a) and Fig. 3 (a), we can know that only the index for the first customer sequence changed, which indicates item c "disappeared" by extending from $\langle \rangle$ to $\langle a \rangle$ for the customer whose SID is 10. So we reduce the support value of item c by 1 from Fig. 2 (b) to get each candidate item's support, resulting in ⟨a⟩ : 3, ⟨b⟩ : 2, ⟨c⟩ : 2, and ⟨d⟩ : 3, as shown in Fig. 3 (b). They are indeed the supports of 2-length sequences, i.e. ⟨aa⟩ : 3, ⟨ab⟩ : 2, ⟨ac⟩ : 2, ⟨ad⟩ : 3. Thus we can determine that $\langle aa \rangle$, $\langle ab \rangle$, $\langle ac \rangle$ and $\langle ad \rangle$ are the 2-length frequent patterns whose common prefix is a.

From the above example, we can show that the main difference between PAID and previous works is to make use of the intermediate result (support value) or not. PrefixSpan, SPADE and LAPIN accumulate the support of each candidate item from scratch. However, PAID can obtain the same result by reducing the support of "disappeared" candidate item from previously found prefix pattern's support. In PAID, it judges an item's disappearance by checking its last position. Obviously, by making good use of the intermediate result, PAID can scan much smaller space than the other algorithms do. For the above example, to find the 2-length frequent patterns whose common prefix is a, PrefixSpan needs 18 scanning times in the projected DB and LAPIN needs 11 scanning times. However, PAID only needs 4 comparison times to discover the same result.

Our contributions can be summarized as follows:

- A novel pruning technique is developed to minimize the search space. By making use of intermediate value (support value) of k-length frequent pattern to calculate the support of k+1-length candidates, we can efficiently avoid the cost of scan as other algorithms do.
- By combining the techniques developed, we proposed a new algorithm, PAID to efficiently discover frequent patterns from large datasets. Several optimizations were used, i.e. LAPIN [17] methodology and binary search for vertical representation of database.
- We conducted comprehensive experiments on both synthetic and real datasets[1]. Experimental results and performance study demonstrated that PAID is much more efficient than the state-of-the-art algorithms by up to more than an order of magnitude for large datasets.

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

2 PAssed Item Deduction Sequential Pattern Mining

PAID follows the pattern growth strategy. To grow to (k+1)-length sequence, we have to first get the knowledge of its k-length prefix frequent pattern.

Definition 1 (Prefix border position set)**.** *Given two sequences, A=$\langle A_1A_2\dots A_m\rangle$ and B=$\langle B_1B_2\dots B_n\rangle$, suppose that there exists C=$\langle C_1C_2\dots C_l\rangle$ for $l \leq m$ and $l \leq n$, 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 instance, 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. To get the prefix border position, PrefixSpan needs $O(\bar{D} \times \bar{L})$ time, where $\bar{D}$ is the average number of customers and $\bar{L}$ is the average sequence length in the projected database, because it uses pseudo projection in sequential search order, while PAID applies the binary search, whose time complexity is $O(\bar{D} \times log(\bar{L}))$.

To test (k+1)-length candidate sequences, PrefixSpan searches in the prefix k-length frequent pattern's projected database. In contrast, PAID scans in the prefix k-length frequent pattern's projected item-last-position list.

Definition 2 (Item-last-position list)**.** *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 a sequences is called the 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 $D_{s,n} \in L_{s,n}$)*

[1]Interested readers can test PAID at http://www.tkl.iis.u-tokyo.ac.jp/∼yangzl/software/PAID/ index.html to confirm its efficiency.

Definition 3 (Candidate border index set)**.** *Given two sequences, A=$\langle A_1A_2\dots A_m\rangle$ and B=$\langle B_1B_2\dots B_n\rangle$, suppose that there exists C=$\langle C_1C_2\dots C_l\rangle$ for $l \leq m$ and $l \leq n$, and that C is a common prefix for A and B. Then we have the prefix border position set $S_{C,i}$ and the item-last-position list $L_{s,i}$ for customer sequence i. We denote the index $CanI_{C,i}$, which points to the node D_i, as the candidate border index of customer sequence i, and the index set $CanI_C$ as the candidate border index set of the common prefix C, if the following conditions hold:*
(a) $D_i \in L_{s,i}$
(b) $D_i.num > S_{C,i}$
(c) $\forall E_i \in L_{s,i}$ before $D_i, E_i.num <= S_{C,i}$

For instance, for the example DB in Table 1, if we have the prefix sequence which is $\langle a\rangle$, then we can get its $candidate\ border\ index\ set$, 10:2, 20:1, 30:1, i.e. (10:b$_{last}$ = 8, 20:c$_{last}$ = 6, and 30:d$_{last}$ = 8), symbolized as "↓", as shown in Fig. 3 (a).

Definition 4 (Projected item-last-position list)**.** *Let C be a sequential pattern in a sequence database S. The C-projected item-last-position list, denoted as $P|_C$, is the collection of suffixes in the item-last-position list with regards to prefix C.*

Definition 5 (Support counting in projected item-last-position list)**.** *Let C be a sequential pattern in sequence database S, and A be a sequence with prefix C. The support counting of A in C-projected item-last-position list $P|_C$, denoted as $support_{P|_C}(A)$, is the number of sequences α in $P|_C$ such that A $\sqsubseteq$ C $\cdot\alpha$.*

The time complexity of searching in the projected database is $O(\bar{D} \times \bar{L})$, while searching in the projected item-last-position list is $O(\bar{D} \times \bar{N})$, where $\bar{N}$ is the average total number of the distinct items in the projected DB. Here we have $\bar{L}/\bar{N} = m$ $(m \geq 1)$, where m denotes the distinct item recurrence rate of the projected DB, or the density of the projected DB. The worst case of PAID is that when there is no duplicate item existing in the database (i.e. association rule mining), m is equal to 1, which means that the time used in searching the projected item-last-position list is the same as that used in searching the projected DB. However, for common datasets, especially for dense database such as DNA sequence, m is probably very large. Here we have a question that, is the result got by searching in the projected DB the same as searching in the projected item-last-position list? The answer can be got from the following two Lemmas.

Lemma 1 (Projected item-last-position list)**.** *Let A and C be two sequential patterns in a sequence database S such that C is a prefix of A.*

1. *$P|_A = (P|_C)|_A$, and*
2. *for any sequence B with prefix C, $support_{P(B)} = support_{P|_C}(B)$.*

Proof. The proof of the Lemma 1 is similar to the proof in [11] (Lemma 3.2). The first part of the lemma follows the fact that, for a sequence B, the suffix of B with regards to A, B/A, equals to the sequence resulted from first doing projection of B with regards to C, i.e., B/C, and then doing projection B/C with regards to A. That is $B/A = (B/C)/A$. The second part of the lemma states that to collect support count of a sequence B, only the sequences in the database sharing the same prefix should be considered. Furthermore, only those suffixes with the prefix being a super-sequence of B should be counted. □

Lemma 2 (Support counting equivalency)**.** *Let C be a sequential pattern in sequence database S, then support counting in C-projected item-last-position list gets the same result as support counting in C-projected database.*

Proof. From Definition 4, we know that the only difference between C-projected item-last-position list and C-projected database is that the former records the last position of different items, and the latter records all positions list of different items. From the support definition, we know that duplicate appearance in the same customer sequence does not contribute to the support counting, which means that the additional information stored in C-projected database is useless. Hence, support counting in C-projected item-last-position list gets the same result as support counting in C-projected database. □

From Lemma 1 and Lemma 2, we can know that if we want to get the support of (k+1)-length sequence, we can count the support of its k-length prefix's projected item-last-position list. However, for large datasets, this naive support counting in projected item-last-position list is not efficient than making use of already found results (support value) of k-length frequent pattern strategy. Hence, we have the following Lemma.

Lemma 3 ((k+1)-length sequence support counting)**.** *Let C be a k-length sequential pattern and A be a (k+1)-length sequential pattern in sequence database S, such that C is a prefix of A. Then we have* $support_{P|_A} = support_{P|_C} - support_{P|_C - P|_A}$.

Proof. From Definition 4, we know that the relationship between C-projected item-last-position list and A-projected item-last-position list is $P|_A = P|_C - (P|_C - P|_A)$. Because item-last-position list only records each distinct item once in every customer sequence, the relationship of support counting between the two projected item-last-position lists $P|_A$ and $P|_C$ is $support_{P|_A} = support_{P|_C} - support_{P|_C - P|_A}$. □

Lemma 3 illustrates the core idea of PAID, which makes good use of the intermediate result (support value) of frequent k-length sequences, to get the supports of (k+1)-length candidate sequences by reducing the supports of those "disappeared" items. Based on the above discussion, the pseudo code of PAID is presented in Fig. 4.

We used a lexicographic tree [3] as the search path of our algorithm and adopted a lexicographic order [3]. This used the Depth First Search (DFS) strategy.

In Step 1, by scanning the DB once, we can obtain the position list table, as in Table 4 and all the 1-length frequent patterns. Based on the last element in each position list, we can sort and construct the *item-last-position list* in ascending order, as shown in Fig. 2 (a). After scanning the DB once, we can also get the support count of $\langle\rangle$-projected item-last-position list, which is indeed the 1-length frequent sequences support.

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

Function : Gen_Pattern($\alpha$, $S$, $Support_P$)
Parameters : $\alpha$ = length k frequent sequential pattern; $S$ = prefix border
position set of (k-1)-length sequential pattern; $Support_P$ = support count in
(k-1)-length sequential pattern's projected item-last-position list
Goal : Generate (k+1)-length frequent sequential pattern

Main():
1. Scan DB once to do:
    1.1 $P_s$ ← Create the position list representation of 1-length sequences
    1.2 $B_s$ ← Find the frequent 1-length sequences
    1.3 $L_s$ ← Obtain the item-last-position list of the 1-length sequences
    1.4 $Support_P|_{\langle\rangle}$ ← Find the support count of $\langle\rangle$-projected item-last-
                    position list
2. For each frequent sequence $\alpha_s$ in $B_s$
    2.1 Call Gen_Pattern ($\alpha_s$, 0, $Support_P|_{\langle\rangle}$)

Function Gen_Pattern($\alpha$, $S$, $Support_P$)
3. $S_\alpha$ ← Find the prefix border position set of $\alpha$ based on $S$
4. $Support_P|_\alpha$ ← Obtain the support count of $\alpha$-projected item-last-
                    position list, based on $S$, $S_\alpha$ and $Support_P$
5. $FreItem_{s,\alpha}$ ← Obtain the item list of $\alpha$ based on $Support_P|_\alpha$
6. For each item $\gamma_s$ in $FreItem_{s,\alpha}$
    6.1 Combine $\alpha$ and $\gamma_s$, results in $\theta$ and output
    6.2 Call Gen_Pattern ($\theta$, $S_\alpha$, $Support_P|_\alpha$)
```

Figure 4. PAID algorithm pseudo code

Table 4. Position List of DB

SID	*Item Positions*
10	$a : 7 \rightarrow 9 \rightarrow null$
	$b : 1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow null$
	$c : 4 \rightarrow null$
	$d : 2 \rightarrow 6 \rightarrow 10 \rightarrow null$
20	$a : 3 \rightarrow 4 \rightarrow 9 \rightarrow null$
	$b : 5 \rightarrow 7 \rightarrow 10 \rightarrow null$
	$c : 2 \rightarrow 6 \rightarrow null$
	$d : 1 \rightarrow 8 \rightarrow null$
30	$a : 2 \rightarrow 4 \rightarrow 7 \rightarrow 10 \rightarrow null$
	$b : null$
	$c : 1 \rightarrow 6 \rightarrow 9 \rightarrow null$
	$d : 3 \rightarrow 5 \rightarrow 8 \rightarrow null$

In function $Gen\_Pattern$, to find the prefix border position set of k-length α (Step 3), 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.) We look for the first position that is larger than the (k-1)-length prefix border position.

Step 4 and Step 5, shown in Fig. 4, are used to find the (k+1)-length frequent pattern based on the frequent k-length pattern and the 1-length candidate items in the projected DB (projected item-last-position list). These two steps are justified by Lemma 1, Lemma 2 and Lemma 3. Commonly, support counting is the most time consuming part in the entire mining process. Here, we test the candidate item in the projected DB (projected item-last-position list), just as PrefixSpan [11] does. The correctness of the strategy was discussed in [11]. The pseudo code of finding (k+1)-length sequential patterns is shown in Fig. 5.

Finding (k+1)-length frequent pattern. In the *item-last-position list*, i.e., Fig 2 (a), we look for the first element whose last position is larger than the prefix border position, as Step 4 and Step 5 in Fig. 5 do. Then, we discovery those items, which "disappear" after growing from (k-1)-length prefix frequent sequence to k-length prefix frequent sequence, and decrement these "disappeared" items support

```
Input : $S$ = prefix border position set of (k-1)-length frequent sequential pattern; $S_\alpha$ = prefix border position set of k-length frequent sequential pattern $\alpha$; $Support_P$ = support count in (k-1)-length sequential pattern's projected item-last-position list; $\varepsilon$ = user specified minimum support
Output : $FreItem_s$ = local frequent item list

1. For each sequence, F
2.     $S_F \leftarrow$ obtain prefix border position of F in $S$
3.     $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in $S_\alpha$
4.     M = Find the corresponding index for $S_F$
5.     N = Find the corresponding index for $S_{\alpha,F}$
6.     while ( M != Null && M < N )
7.         $Support_P$[M.item] - -;
8.         M++;
9. For each item $\beta$ in Suplist
10.     If ( $Support_P[\beta] \geq \varepsilon$)
11.         $FreItem_s$.insert($\beta$);
```

Figure 5. Finding (k+1)-length frequent patterns

from the support count of the projected item-last-position list, as shown in Step 6 to Step 8. Finally, we can get the frequent items in the projected item-last-position list (projected DB), as shown in Step 9 to Step 11. Obviously, in dense datasets, the size of "disappeared" projected item-last-position list should be much smaller than the projected DB, which is scanned by PrefixSpan algorithm. Moreover, we only pass and count once for each different item in the "disappeared" projected item-last-position list because, in *item-last-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, PAID will save much time because our search space is much smaller than the one used in PrefixSpan by making good use of the intermediate result (support value). The example has been described in Section 1.2.

I-Step of PAID. As Ayres et al. did in [3], the whole process of sequential pattern mining should includes two steps: a *sequence-extension step* (*S-Step*) and a *itemset-extension step* (*I-Step*). We have already described the *S-Step* of PAID. The *I-Step* is similar to the *S-Step*. One difference is that in *I-Step*, the basic unit is 2-length itemset extension sequence, i.e., (ab), (bc), instead of 1-length sequence, i.e., a, b, in S-Step. Another difference is that when patterns grow, *I-Step* should guarantee that the prefix sequences of the tested candidates are the same. We deal with it by joining the position list of each candidate IE item after current prefix position, which is similar to the method used in SPADE [18].

3 Performance Study

In this section, we describe our experimentation and evaluations conducted on both synthetic and real datasets by comparing four algorithms PrefixSpan, SPADE, LAPIN and PAID[2]. We performed the experiments using a 1.6GHz Intel Pentium(R)M PC with a 1G memory, running Microsoft Windows XP. All the four algorithms were written in C++. The output of the programs was turned off to make the comparison equitable. Detailed algorithm implementation of the four algorithms is described as follows:

[2]There are many algorithm implementations existing. We choose PrefixSpan and SPADE as our competitors because they are the state-of-the-art algorithms and have shown superior performance compared to others [11] [18].

Table 5. Parameters used in data generation

Symb.	Meaning
D	Number of customers in the dataset ($\times$1K)
C	Average number of transactions per customer
T	Average number of items per transaction
S	Average length of maximum sequences
I	Average length of transactions within maximum sequences
N	Number of different items in the dataset ($\times$1K)

1. *PrefixSpan. PrefixSpan* was tested with the implementation provided by the algorithm inventor [11][3].

2. *SPADE. SPADE* was tested with the implementation provided by the algorithm inventor [18][4].

3. *LAPIN. LAPIN* was tested with the implementation provided by the algorithm inventor [17] (*LAPIN_LCI* was tested here because it is the fastest one among the LAPIN family for large datasets).

4. *PAID. PAID* was implemented as described in this paper.

3.1 Experimental Results

For the datasets used in our performance study, we used two kinds of datasets: a group of synthetic datasets and two real datasets. It showed that PAID outperformed PrefixSpan and SPADE by up to more than an order of magnitude on these datasets and had a good scalability. Compared with LAPIN, PAID is faster and consumes much less memory than LAPIN.

Synthetic Data. The synthetic datasets were generated by an IBM data generator, as described in [2]. The meaning of the different parameters used to generate the datasets is shown in Table 5.

The first test of the four algorithms is on the dataset C100T20S10I10N1D5. The average number of items in a transaction is 20 and the average number of transactions in a sequence is set to 100. So it is a dataset with very long sequences. Fig. 6 (a) shows the distribution of frequent sequences of the dataset. The number of frequent sequences is in logarithm scale (i.e., $\log_{10}$L for sequence length L). Fig. 6 (b) shows the processing time of the four algorithms at different support thresholds. We can see that PAID is much more efficient than PrefixSpan, SPADE and LAPIN. When min_support = 0.95, PAID (runtime = 595 seconds) is more than an order of magnitude faster than PrefixSpan (runtime = 8161 seconds) and SPADE (runtime = 4980 seconds). The running time of LAPIN is 1013 seconds.

The second test is performed on the dataset C40T20S20I20N0.2D100, which is much larger than the first synthetic dataset since it contains 100k customer sequences. Because the number of items is 200 and the average length of sequences is 800 (i.e., 20$\times$ 40), it is denser than the first dataset. Fig. 7 (a) shows the distribution of frequent sequences of the dataset. Fig. 7 (b) shows the processing time of the four algorithms at different support thresholds. The result mirrors that of the first test closely.

[3]http://illimine.cs.uiuc.edu/

[4]http://www.cs.rpi.edu/ zaki/software/

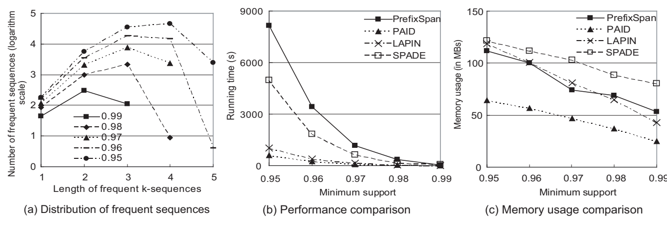


Figure 6. Dataset (C100T20S10I10N1D5)

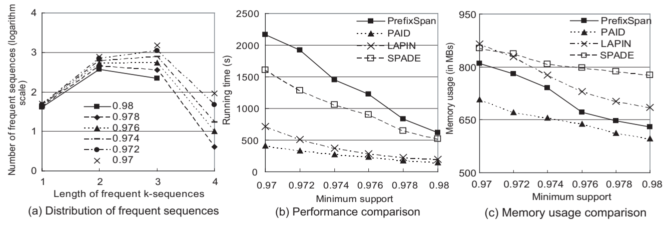


Figure 7. Dataset (C40T20S20I20N0.2D100)

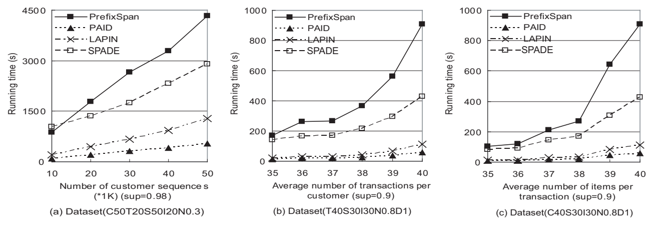


Figure 8. Scalability test

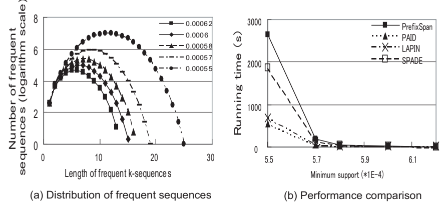


Figure 9. Dataset (Gazelle)

Memory Usage. We compare the memory usage among the four algorithms, using the two datasets shown above. Fig. 6 (c) shows the results for C100T20S10I10N1D5 dataset, from which we can see that PAID is efficient in memory usage. It consumes almost half memory of that used in PrefixSpan, SPADE and LAPIN. For example, at support 0.95, PrefixSpan consumes about 112 MB memory, SPADE consumes about 122 MB memory, LAPIN consumes about 119 MB memory, while PAID only uses about 64 MB memory. Fig. 7 (c) shows the memory usage for dataset C40T20S20I20N0.2D100. At support 0.97, PrefixSpan consumes about 809 MB memory, SPADE consumes about 853 MB memory, LAPIN consumes about 866 MB memory, while PAID uses 707 MB memory.

Scalability Test. We study how PAID performs with increasing number of customer sequences (D), average number of events per customer sequence (C) and average number of items per event (T), respectively. Note that for the left experiments, we only compare the performance of the four algorithms because of the limited space. The memory usage comparison mirrors the result in the above section closely.

Fig. 8 (a) shows how PAID scales up as the number of customer sequences is increased, from 10K to 50K, whose corresponding database size ranging from 40M to 200M. The experiment was performed on the C50T20S50I20N0.3 with minimum support is 0.98. It can be observed that PAID scales almost linearly and is much faster than the other three algorithms. For example, in Fig. 8 (a), when D = 50, PAID (runtime = 544 seconds) is about 8 times faster than PrefixSpan (runtime = 4329 seconds), five times faster than SPADE (runtime = 2903 seconds) and two times faster than LAPIN (runtime = 1292 seconds). Fig. 8 (b) and 8 (c) show the other two scalability experimental results. For both the graphs, we used S30I30N0.8D1. In Fig. 8 (b) we set T to 40, and varied C from 35 to 40, and Fig. 8 (c) we set C to 40, varied T from 35 to 40. It can be easily observed PAID scales linearly with the two varying parameters and is always faster than the other three algorithms.

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 datasets.

The first real dataset, Gazelle, was obtained from Blue Martini company, which was also used in KDD-Cup 2000. This dataset contains 59602 sequences (i.e., customers), 149639 sessions, and 497 distinct page views. The average sequence length is 2.5 and the maximum sequence length is 267. More detailed information about this dataset can be found in [12]. Fig. 9 (a) shows the distribution of frequent sequences of Gazelle dataset for different support thresholds. We can see that this dataset is a sparse dataset compared with Rat because, only when the support threshold is very low are there some long frequent sequences (i.e., when min_sup=0.00062, total frequent sequences number = 207168). Fig. 9 (b) shows the performance comparison between PrefixSpan, SPADE and PAID for Gazelle dataset. We can see that PAID is more efficient than PrefixSpan and SPADE. For example, at support 0.00055, PAID (runtime = 583 seconds) is near five times faster than PrefixSpan (runtime = 2642 seconds) and four times faster than SPADE (runtime = 1857 seconds). The running time of LAPIN is 677 seconds.

We have obtained the second dataset, MSNBC, from the UCI KDD Archive[5]. This dataset comes from Web server logs for msnbc.com and news-related portions of msn.com on Sep. 28, 1999. There are 989,818 users and only 17 distinct items, because these items are recorded at the level of URL category, not at page level, which greatly reduces the dimensionality. The average sequence length is 6 and the maximum sequence length is 14,795. Fig. 10 (a) shows the distribution of frequent sequences of Rat dataset for different support thresholds, from 0.00011 to 0.00015. When the support threshold is 0.00011, there are many frequent sequences (total number = 45,541,248). Fig. 10 (b) shows the performance comparison among PrefixSpan, SPADE and PAID for MSNBC dataset. The result mirrors that of the Gazelle dataset closely.

[5]http://kdd.ics.uci.edu/databases/msnbc/msnbc.html

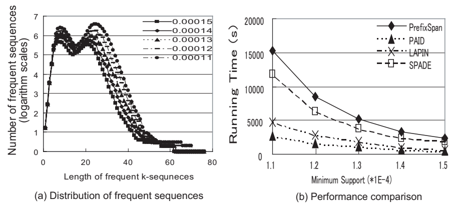


Figure 10. Dataset (MSNBC)

3.2 Analysis

With the above thorough performance study, we are convinced that PAID is much more effective than PrefixSpan, SPADE and LAPIN. There are several reasons:

- PAID reuses already found results (support value) of k-length frequent sequences to count the supports of (k+1)-length candidate sequences. Thus, PAID avoids to count from scratch, as done in the other algorithms.
- PAID applies the Last Position Induction strategy (LAPIN) [17], which scans only a small part of projected database. However, the other algorithms need to scan the whole (projected) database to count the candidate support.
- By using a vertical representation format of the original database, PAID can apply binary search in each iteration to find the corresponding frequent prefix sequence position. In contrast, the other algorithms apply sequential search, which is obviously much slower than PAID for long pattern large datasets.
- PAID consumes less and stable memory space because it does not need to construct the S-Matrix dynamically to store the candidate sequences support information, as done in PrefixSpan, or the bitmap representation of item-last-position list, as done in LAPIN (LAPIN_LCI).

4 Conclusions

In this work, we have proposed a new algorithm, PAID, for efficient sequential pattern mining. Our main idea is to make good use of the intermediate result of k-length sequence pattern, to discovery the (k+1)-length frequent pattern. By combining some other optimizations, PAID can reduce searching significantly and thus, to improve the performance. By thorough experiments and evaluations, we have demonstrated that PAID outperforms the previous works by up to more than an order of magnitude with low memory consuming. Our experimental results also show that PAID is very efficient at using synthetic data and real-world data. In the future, we will investigate some related problems, i.e., closed sequential pattern mining and pattern summarization.

References

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *VLDB*, pages 487-499, 1994.

[2] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *ICDE*, pages 3-14, 1995.

[3] J. Ayres, J. Gehrke, T. Yiu and J. Flannick. Sequential Pattern Mining using A Bitmap Representation. In *KDD*, pages 429-435, 2002.

[4] D. Chiu, Y. Wu and A.L.P. Chen. An Efficient Algorithm for Mining Frequent Sequences by a New Strategy without Support Counting. In *ICDE*, pages 375-386, 2004.

[5] M.N. Garofalakis, R. Rastogi and K. Shim. SPIRIT: Sequential PAttern Mining with Regular Expression Constraints. In *VLDB*, pages 223-234, 1999.

[6] J. Han, G. Dong and Y. Yin. Efficient Mining of Partial Periodic Patterns in Time Series Database. In *ICDE*, pages 106-115, 1999.

[7] H.C. Kum, J. Pei, W. Wang and D. Duncan. ApproxMAP: Approximate Mining of Consensus Sequential Patterns. In *SDM*, pages 311-315, 2003.

[8] C. Luo and S.M. Chung. Efficient Mining of Maximal Sequential Patterns Using Multiple Samples. In *SDM*, pages 64-72, 2005.

[9] H. Mannila, H. Toivonen and A.I. Verkamo. Discovering Frequent Episodes in Sequences. In *KDD*, pages 210-215, 1995.

[10] S. Parthasarathy, M.J. Zaki, M. Ogihara and S. Dwarkadas. Incremental and Interactive Sequence Mining. In *CIKM*, pages 251-258, 1999.

[11] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. Mining Sequential Patterns by Pattern-growth: The PrefixSpan Approach. *TKDE*, 16(11): pages 1424-1440, 2004.

[12] R. Kohavi, C. Brodley, B. Frasca, L. Mason and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. *SIGKDD Explorations*, (2)2: pages 86-98, 2000.

[13] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *EDBT*, pages 3-17, 1996.

[14] W.G. Teng, M.S. Chen and P.S. Yu. A Regression-Based Temporal Pattern Mining Scheme for Data Streams. In *VLDB*, pages 93-104, 2003.

[15] P. Tzvetkov, X. Yan and J. Han. TSP: Mining Top-k Closed Sequential Patterns. In *ICDM*, pages 347-358, 2003.

[16] X. Yan, J. Han and R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *SDM*, pages 166-177, 2003.

[17] 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/Document/LAPIN.pdf

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