有效率的以位元技術挖掘連續樣式的方法
An Efficient Bitmap-Based Approach to Mining Sequential Patterns for Large Databases *

張玉盈, 黃立文, 陳俊榮, 吳建輝
Ye-In Chang, Lee-Wen Huang, Jiun-Rung Chen, and Chien-Hui Wu
Dept. of Computer Science and Engineering
National Sun Yat-Sen University
{*E-mail: changyi@cse.nsysu.edu.tw*}

摘要

在一個交易資料庫中, 連續樣式是指顧客在一段時間內所買的物品存在一些關係。假如可以藉由連續樣式的挖掘來取得這些物品間的關係, 將可提供更好的行銷策略以吸引顧客的注意。在過去的研究中, GSP演算法產生較少的候選集, 但花了太多時間去計算候選集的個數。SPAM演算法使用位元運算去減少計算候選集個數的時間, 卻產生太多的候選集。在本論文中, 我們提出一個新的位元技術的演算法, 同時能產生較少的候選集, 並減少計算候選集個數的時間。實驗結果顯示, 我們的方法在執行時間上可以提供較好的效能。

關鍵詞: 位元技術挖掘, 資料探勘, 知識發現, 樣式分析, 連續樣式

Abstract

One of important research areas of data mining is *mining sequential patterns*. For a transaction database, sequential pattern means that there are some relations between the items bought by customers in a period of time. If we can find these relations by mining sequential patterns, we can provide better selling strategy to gain more customers' attentions. However, since the transaction database contains a lot of data, and it will be scanned during the mining process again and again, to improve the running efficiency is an important topic. In the GSP algorithm, it generates fewer number of candidates; however, it still spends too much time in counting candidates. In the SPAM algorithm, it uses bitwise operations to reduce the time for counting candidates; however, it generates too many candidates which will never become frequent itemsets. In this paper, we propose a new bitmap-based algorithm. We use the similar candidate generation method presented in the GSP algorithm to reduce the number of candidates and the similar counting method proposed in the SPAM algorithm to reduce the time of counting candidates. From our simulation results, based on the same bit representation for the transaction database, we show that our proposed algorithm could provide better performance than the SPAM algorithm in terms of the processing time, since our algorithm could generate fewer number of candidates than the SPAM algorithm.

Keywords: Bitmap-based mining, data mining, knowledge discovery, pattern analysis, sequential patterns.

1 Introduction

In recent years, with the computers and information industries growing more and more rapidly, varies data around us become more complexity and huge. As the amount of data grows, it becomes very difficult to determine the useful data. So, there comes a research named *data mining* which is used to help making decisions and information retrieval [18]. *Mining sequential patterns* is a very important topic for the behaviors in the real world. A sequential pattern indicates a sequence of transactions that usually occurred serially in time. For example, in a transactional database, one might purchase a TV and then purchase some speakers at some later time. After a period of time, he/she could possible buy a VCD player

*This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-93-2213-E-110-003.

and some VCDs. If the database contains a sufficient number of customers who have a purchasing sequence of TV, speaker, VCD player and VCD, then such a sequence is a sequential pattern. In this case, we can provide the speaker information to the customer after he/she has bought a TV.

The problem of mining sequential patterns was first introduced in [3]. The input data is a set of sequences, called *data-sequences*. Each data-sequence is a list of *transactions*, where each transaction is a sets of literals, called *items*. Each transaction is associated with a transaction-time. A *sequential pattern* consists of a list of sets of items. The problem is to find all sequential patterns with a user-specified minimum *support*, where the support of a sequential pattern is the percentage of data-sequences that contain the pattern [15]. Generally speaking, there are two differences between mining sequential patterns and mining association rules: candidate generation and counting, where mining association rules means that the presence of some items in a transaction will imply the presence of other items in the same transaction. The difference in the part of the candidate generation is shown in Figure 1. Since the transaction time is a factor of sequential mining, there are four candidates in sequential patterns, while there are only two candidates in association rules. Note that in mining association rules, the information about the customer ID is ignored. The other difference is the counting for candidates, which is shown in Figure 2. Since the same customer has two transactions with the same item, the count of the candidates of sequential patterns is 1, while the count of the candidate of association rules is 2. From the above observations, we can not solve the problems of sequential patterns by using the existing association rules algorithms directly.

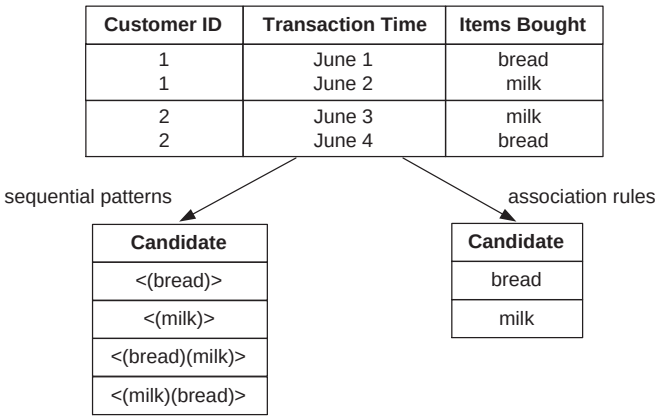


Figure 1: The difference of candidate generation between mining sequential patterns and mining association rules

The tasks of finding frequent sequential patterns in large databases will bring lots of benefits for many applications, such as DNA sequences, telecommunication alarms, customer relationship management [9, 10]. There are several Apriori-based algorithms, such as AprioriAll [3] and GSP [15], have been proposed. (Note that the Apriori algorithm is one of well-known algorithm for mining association rule.) However, their search space is extremely large. For example, with m attributes, there are $O(m^k)$ potentially frequent sequences of length k. Moreover, these algorithms are iterative in nature, which need high I/O cost. Pei *et al.* proposed another two algorithms [8] which could provide better performance than the GSP algorithm [14] by reducing the huge set of candidates and the multiple scans of database. In their algorithms, they use the projected database as their data structure which needs a lot of memory space and needs too much time in constructing those structures.

Basically, the above algorithms for the candidate generation apply a common property, "the subset sequences of a frequent candidate sequence are also frequent", which reduces the number of candidates resulting in reducing both the process-

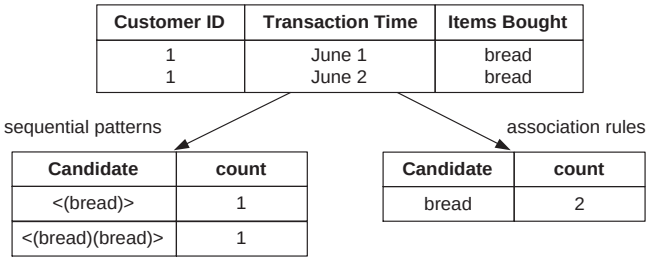


Figure 2: The difference of counting between mining sequential patterns and mining association rules

ing time and the storage space. This is one spirit of the Apriori algorithm [2]. Therefore, we need an algorithm that keeps the spirit of the Apriori algorithm but avoid the expensive cost of candidate generation and test.

On the other hand, we notice the constant execution time of all logical operations at the bit level (e.g., AND, OR, NOT, etc.). Additionally, we observe that sorting the data is not required when using bitmaps [6]. Besides, since the candidate generation and test is under bitwise operations, the bitmap data structure is simple, easy to implement, and providing fast processing time.

In [4], based on the bitmap representation, Ayres *et al.* proposed the SPAM algorithm to solve the problems of I/O cost and poor data structure that the Apriori-based or non-Apriori-based algorithms have met. Besides, it keeps the advantages of bitwise operations. The SPAM algorithm is simple, fast, and easy to implement. However, the process of the candidate generation based on the *Lexicographic Sequence Tree* is time and space consuming. Moreover, it generates too many candidates which will never become frequent itemsets.

In this paper, we propose a new bitmap-based algorithm to discover the frequent sequential patterns, which avoids the weakness of the SPAM algorithm but keeps its advantages, such as the speed of the bitwise operations. Moreover, we recall the spirit of Apriori-based algorithm, that is, to make full use of the previous generated frequent sequences. Hence, by modifying the way to generate candidates in the GSP algorithm and applying the bitwise operations in the SPAM algorithm, the proposed algorithm can mine sequential patterns efficiently. That is, we use the similar candidate generation method presented in the GSP algorithm to reduce the number of candidates and the similar counting method proposed in the SPAM algorithm to reduce the time of counting candidates. In the proposed algorithm, we classify the itemsets into two cases, simultaneous occurrence (noted as AB) and sequential occurrence (noted as A->B). In the case of simultaneous occurrence, the number of candidate is C_k^n based on the exhausted method. In order to prevent too many candidates generated, we make use of the property, "the subsets of a frequent itemset are also frequent", to reduce the number of candidates from C_k^n to C_k^y, $k \leq y < n$. In the case of sequential occurrence, the candidates are generated by using a special join operation which could combine, for example, A->B and B->C to A->B->C. Moreover, we have to consider two other cases: (1) combing A->B and A->C to A->BC; (2) combing A->C and B->C to AB->C. The method of counting candidates is similar to the SPAM algorithm (*i.e.*, bitwise operations). From our simulation results, we show that the proposed algorithm outperforms the SPAM algorithm in all datasets. The main reason is that our algorithm could generate fewer number of candidates than the SPAM algorithm, which reduces a lot of time of testing candidates.

The rest of the paper is organized as follows. Section 2 gives a survey of some algorithms for mining sequential patterns. In Section 3, we present a new bitmap-based algorithm for mining sequential patterns. In Section 4, we give a

comparison of the performance of the SPAM algorithm and our proposed algorithm. Section 5 gives a conclusion.

2 Background

In this section, we give a survey of some well-known data mining algorithms for sequential patterns problems.

2.1 Formal Problem Definitions

First, we give the formal definitions of the problem of mining sequential patterns which are proposed by Agrawal and Srikant [3, 15]. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of *items*. A subset $X \subseteq I$ is an *itemset* and $|X|$ is the *size* of X. A *sequence* $s = \langle s_1, s_2, ..., s_m \rangle$ is an ordered list of itemsets, where $s_i \subseteq I, i \in \{1, ..., m\}$. The size, m, of a sequence is the number of itemsets in the sequence. The length l of a sequences $s = \langle s_1, s_2, ..., s_m \rangle$ is defined as $l = \sum_{i=1}^{m} |s_i|$. A sequence with length l is called an l-sequence [4]. A sequence $s_a = \langle a_1, a_2, ..., a_n \rangle$ is a *subsequence* of another sequence $s_b = \langle b_1, b_2, ..., b_m \rangle$, if there exist integers $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, ..., a_n \subseteq b_{i_n}$. For example, the sequence $k = \langle (A)(BC) \rangle$ is a subsequence of $\langle (AB)(BCE)(D) \rangle$, since $(A) \subseteq (AB)$, $(BC) \subseteq (BCE)$. However, the sequence $\langle (ABC) \rangle$ is not a subsequence of k.

Let D be a set of sequences called *data-sequences*. Each data-sequence is a list of transactions, ordered by increasing transaction-time. A transaction T consists of a customer-id, a transaction time, and a set of items, such as $T = (CID, TID, I)$, where CID is a customer-id, TID is a transaction time, and I is a set of items. We assume that no data-sequence has more than one transaction with the same transaction-time. The *support* of a sequence s in D is defined as

$$support(s) = \frac{\#\ of\ the\ data-sequences\ that\ contain\ s}{\#\ of\ total\ data-sequences}$$

Table 1: The transaction database

CID	TID	Itemset
1	1	(AD)
1	2	(B)
1	5	(AF)
2	2	(BD)
2	6	(EF)
3	4	(AE)
3	5	(A)

Table 2: The data-sequences

CID	Sequence
1	$\langle (AD)(B)(AF) \rangle$
2	$\langle (BD)(EF) \rangle$
3	$\langle (AE)(A) \rangle$

For a user-specified min_sup, if $support(s) \geq min\_sup$, s is a frequent itemset.

Table 1 shows the transaction database which is sorted by CID and TID. Table 2 shows the corresponding data-sequences. From this table, the number of the data-sequences is three. Suppose we want to find the support of $s = \langle (A)(A) \rangle$. For a sequence, such as $\langle (A)(A) \rangle$, has the same meaning as the form of A->A. From Table 2, we know that s is a subsequence of the sequences of customer 1 and customer 3. Therefore, we have $support(s) = 66.7\%$. If $min\_sup = 50\%$, s is a frequent 2-itemset.

2.2 Related Work

Many algorithms have been proposed for mining sequential patterns [1, 3, 4, 5, 7, 8, 11, 12, 14, 13, 15, 16, 17, 19, 20, 21]. Ayres *et al.* proposed the $SPAM$ algorithm in which the data is recorded in the bit representation [4]. In this algorithm, they use the sequence-extended and itemset-extended techniques to build up the lexicographic sequence tree, and use the bitwise AND operation to generate the candidates. However, it spends a lot

of time to prune sequence-extended and itemset-extended sequences. Moreover, it costs lots of space to store the necessary information to follow the rule of the lexicographic sequence tree. The above algorithms assume that there is no time constraint for sequential patterns. The first two of the following algorithms introduce time constraints for mining sequential patterns.

Srikant and Agrawal proposed the *GSP* algorithm [15]. This algorithm adds taxonomies, sliding windows, and time constraints for the real world conditions. These constraints affect the counting of the candidates. In addition, the GSP algorithm uses a contiguous subsequence method to generate the candidates. As compared to AprioriAll, GSP is between 30% to 5 times faster than AprioriAll [15].

3 The Bitmap-Based Approach

Most of the proposed algorithms follow the *Apriori* principle to solve the sequential pattern mining problems. One advantage of the principle is that any subset of the frequent itemsets will be frequent. Thus, we can reduce a lot of steps to check out whether a candidate is frequent or not. However, if we use the string recognition to do this work, it will be too slow to find out the frequent itemsets. Therefore, the *SPAM* algorithm uses the bitmap representation to speed up the process [4]. Although it reduces the time spent on generating candidates, it generates too many candidates that are not frequent. In this Section, we present an algorithm based on the bitmap-based approach to reduce the number of candidates by using a new candidate generation method and adding some new bitwise operations for needs.

3.1 The Bitmap Representation

In this section, we give a simple example to illustrate how to denote sequential patterns by using the bitmap representation. The bitmap representation uses $True$ and $False$ to show the states of items. For example, Figure 3-(a) shows the original transactions. It means that customer 1 buys ACD, AB, and BCD at time 10, 15, and 20, respectively. We transform these transactions into the bitmap representation, as shown in Figure 3-(b). To simply our denotation, we use 1 and 0, instead of $True$ and $False$, to represent whether the item is bought or not. In Figure 3-(b), we can observe that when $TID = 10$, items A, C, and D are marked with 1, and item B is marked with 0, since 1 means that it really happens while 0 means that it does not.

There are two basic cases of sequential patterns. The first one is that two items bought in different time, *i.e.*, the *sequential* occurrence. It means that two items have a former and later relationship and the order of them is important. The second one is that two items bought at the same time, *i.e.*, the *simultaneous* occurrence.

Now, we describe the bitmap operations between these two basic cases of sequential patterns. Suppose we want to mine the relationship between items A and B. From Figure 3-(b), we know A = [110] and B = [011] with TID = 10, 15, and 20. For the first one, A->B (the sequential occurrence), we find out the first bit 1 that happens in A with its TID being the smallest one among those transactions for the same customer. We then change it to 0, and set the rest of bits to 1 no matter they are 1 or 0. We apply an AND operation on the result and B. Finally, we get the result of A->B as shown in Figure 4-(a). For the second one, AB (the simultaneous occurrence), we

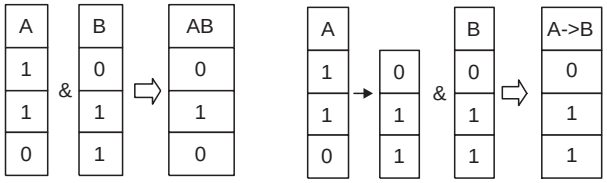
CID	TID	Itemsets
1	10	A C D
1	15	A B
1	20	B C D

(a)

CID	TID	A	B	C	D
1	10	1	0	1	1
1	15	1	1	0	0
1	20	0	1	1	1

(b)

Figure 3: An example for the bitmap representation: (a) the original transactions; (b) the corresponding bitmap representation.



(a) (b)

Figure 4: Two cases using bitwise operations: (a) A->B (sequential occurrence); (b) AB (simultaneous occurrence).

apply an AND operation on A and B directly. Thus, we can have the result of AB as shown in Figure 4-(b). These operations are proposed in the $SPAM$ algorithm [4].

3.2 Definitions

To efficiently mining sequential patterns based on the bitmap representation in our algorithm, we will apply one bit transformation technique, *S-step*, used in the SPAM algorithm [4], which is defined as follows:

- **S-step transformation:** If there is an itemset x and its corresponding bitmap representation is $[x_0, x_1, ..., x_{i-1}, x_i]$, $\exists k$ such that $x_k = True$ (1), and for all $i < k$ such that $x_i = False$ (0), we set $x_k = False$ (0) and $x_i = True$ (1), $\forall i > k$.

In addition to the above bit transformation, we define the following two operations:

	x = [0101]	y = [1100]
S-step transformation	[0011]	[0111]
Inner *AND* operation	[0]	[0]
Inner *OR* operation	[1]	[1]
Outer *AND* operation	[0100]	
Outer *OR* operation	[1101]	

Figure 5: An example for the S-step transformation, the inner AND/OR operation, and the outer AND/OR operation

- **Inner AND/OR operation:** If there is an itemset x and its corresponding bitmap representation is $[x_0, x_1, ..., x_{n-1}, x_n]$, an inner AND operation is x_0 AND x_1 AND ... AND x_{n-1} AND x_n, and an inner OR operation is x_0 OR x_1 OR ... OR x_{n-1} OR x_n.

- **Outer AND/OR operation:** If there are two itemsets x and y and their corresponding bitmap representations are $x = [x_0, x_1, ..., x_{n-1}, x_n]$, $y = [y_0, y_1, ..., y_{n-1}, y_n]$, respectively, an outer AND operation is $[x_0$ AND y_0, x_1 AND y_1, ..., x_{n-1} AND y_{n-1}, x_n AND $y_n]$ and an outer OR operation is $[x_0$ OR y_0, x_1 OR y_1, ..., x_{n-1} OR y_{n-1}, x_n OR $y_n]$

For example, given the itemsets $x = [0101]$ and $y = [1100]$, after performing S-step transformations, we have $x = [0011]$ and $y = [0111]$, as shown in Figure 5.

3.3 The Algorithm

In this section, we illustrate the proposed algorithm by using an example transaction database as shown in Figure 6. Figure 7 shows the corresponding bitmap representation. The procedure *main* is shown in Figure 8. Given $min\_sup = 50\%$, for this example, it means that an itemset is frequent

CID	TID	Itemsets
1	10	C D
	15	A B C
	20	A B F
	25	A D F
2	15	A B E F
	25	E
3	10	A B F
4	15	D G H
	20	B F
	30	A G

Figure 6: An example transaction database

CID	A	B	C	D	E	F	G	H
1	0	0	1	1	0	0	0	0
	1	1	1	0	0	0	0	0
	1	1	0	0	0	1	0	0
	1	0	0	1	0	1	0	0
2	1	1	0	0	1	1	0	0
	0	0	0	0	1	0	0	0
3	1	1	0	0	0	1	0	0
4	0	0	0	1	0	0	1	1
	0	1	0	0	0	1	0	0
	1	0	0	0	0	0	1	0

Figure 7: The bitmap representation Exist for the example database

if it exists in more than 2 ($= 4 * 50\%$) customer transactions. The whole algorithm is processed as follows. First, we find out L_1 by using procedure $Reduce\_BDB$.

For example, as shown in Figure 9-(a), for item $A = [0111]$ with $CID = 1$, we apply an inner OR operation on A. The result is 1, which means that the support of this item is 1. After summarizing the support of other items, we delete the items with their support less than min_sup. Finally, the remaining items are L_1, as shown in Figure 9-(b).

Next, to generate L_2, as described in the previous section, we have to consider two cases for the same CID: the itemsets with the same TID (denoted Case 1) and the itemsets with different $TIDs$ (denoted Case 2), for mining sequential patterns.

For Case 1, the itemsets with the same TID, we call procedure $L\_Same\_TID\_P(Y, n)$, as shown in Figure 10, where procedure $Combination$ is used to compute the combinations of the items in Y, and stores all the possible combinations in $COMB$. For each CID, according to $COMB$, we apply an outer AND operation on any two of

```
procedure main () {
  ReadFromSource(Exist);
  Reduce_BDB(Exist);
  /* Generating L1 */
  L_Same_TID_P(Y, n);
  /* Generating L2 with the same TID */
  L_Diff_TID1_P();
  /* Generating L2 with different TIDs */
  while (Ln ≠ ∅) then {
    MakeTable(L_Diff_TIDn−1);
    Y = Reduce_CSameTID(L_Same_TIDn−1, n);
    L_Same_TID_P(Y, n);
    L_Diff_TID_CaseA();
    L_Diff_TID_CaseB();
    L_Diff_TID_CaseC();
    n = n + 1; } }
```

Figure 8: Procedure *main*

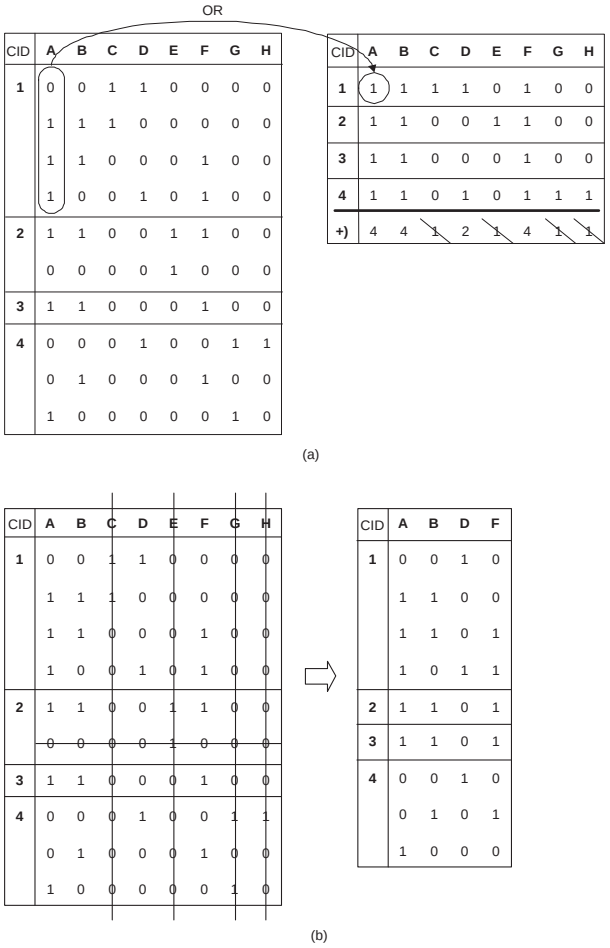


Figure 9: The generation of L_1: (a) the process of pruning items by using an OR operation; (b) the final bitmap representation of L_1.

procedure $L\_Same\_TID\_P$ (Y, n) **{**
foreach CID c **do {**
Combination(Y,n,c,COMB);
$Total\_COMB = Total\_COMB \bigcup COMB$;
foreach itemset $I \in COMB$ **do {**
$isTrueTrans$ = False;
foreach TID t **do {**
$isTrueItem$ = True;
foreach item $i \in I$ **do**
$isTrueItem = Exist[c, i, t]$ AND $isTureItem$;
$isTrueTrans = isTrueTrans$ OR $isTrueItem$;**}**
if $(isTrueTrans = True)$ **then**
$I.count = I.count + 1$; **} }**
foreach itemset $I \in Total\_COMB$ **do {**
if $I.count \geq min\_s$ **then**
$L\_Same\_TID_n = L\_Same\_TID_n \bigcup \{I\}$; **} }**

Figure 10: Procedure $L\_Same\_TID\_P$

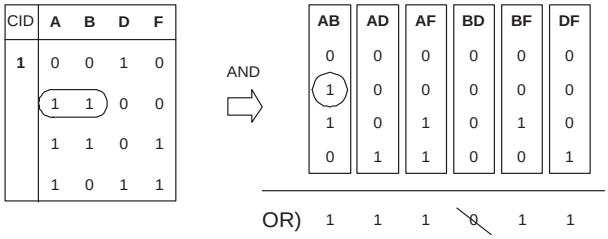


Figure 11: The generation of C_2 for $CID = 1$ and the same TID (Case 1)

items in $Exist$. Then, we apply an inner OR operation on each candidate to obtain their support. Finally, we summarize the support to find out the frequent itemset. For example, as shown in Figure 11, after we apply an outer AND operation on item $A = [0111]$ and item $B = [0110]$, we will have a new itemset $AB = [0110]$. Then, we apply an inner OR operation on itemset AB and the result is 1, which means that AB is in the transactions for $CID = 1$. The result of $C\_Same\_TID_2$ is shown in Figure 12. If the final support are less than *min_sup*, they can not be frequent and will be deleted. Finally, the remaining itemsets are $L\_Same\_TID_2$.

For Case 2, itemsets with different $TIDs$, we call procedure $L\_Diff\_TID1\_P$, as shown in Fig-

CID	C2
1	A B A F B F A D D F
2	A B A F B F
3	A B A F B F
4	BF

Figure 12: The result of C_2 for the same TID (Case 1)

procedure $L\_Diff\_TID1\_P()$ **{**
foreach CID c **do {**
$MPermutation(2,c,MPERM)$;
$Total\_MPERM = Total\_MPERM \bigcup MPERM$;
foreach itemset $I \in MPERM$ **do {**
$Transformation(I.former)$;
$isTrueTrans$ = False;
foreach TID t **do**
$isTrueTrans = isTrueTrans$ OR
($I.former[t]$ AND $I.later[t]$);
if ($isTrueTrans$ = True) **then**
$I.count = I.count + 1$; **}**
foreach itemset $I \in Total\_MPERM$ **do {**
if ($I.count \geq min\_s$) **then {**
$L\_Diff\_TID_2 = L\_Diff\_TID_2 \bigcup \{I\}$;
$CheckSet = CheckSet \bigcup \{I\}$; **} } }**

Figure 13: Procedure $L\_Diff\_TID1\_P$

ure 13, where procedure *MPermutation (i.e., Modified Permutation)* is used to compute the permutations of L_1 and stores all the possible permutations in $MPERM$. The difference between the modified permutation and the mathematical permutation is that the modified permutation has a self-permutation relationship. For example, if we have A and B, the mathematical permutation will have A->B and B->A. However, relationships A->A and B->B should be also considered in our algorithm, which are necessary for mining sequential patterns. Therefore, according to procedure $MPERM$, we change the bits of the former itemset by applying the S-step transformation. We apply an outer AND operation on the result and the later itemset. Then, we apply an inner OR operation on the result and the resulting bit is the support of the item.

For example, assume that B->A is one of the itemsets in $MPERM$, B = [0110] is the former itemset and A = [0111] is the later itemset. The process of computing the result of B->A is shown in Figure 14. After applying the S-step transformation on B, we have B = [0011]. The resulting bits mean that any itemset appearing with TID = 3 and 4 will have a former itemset B. Since the itemset B->A is composed of B and A, we should also examine whether A will satisfy this condition or not. Therefore, we apply an outer AND operation on B and A = [0111], and we have B->A = [0011], which means those cases that the later itemset A appears in the pattern B->A are TID = 3 and 4, respectively. Then, we apply an inner OR operation on B->A. Finally, the result is 1, which means that B->A exists in the transactions for CID = 1. The result of $C\_Diff\_TID_2$ is shown in Figure 15. If the results are less than min_sup, they can not be frequent and will be deleted. Finally, the remaining itemsets are $L\_Diff\_TID_2$. The final result of $L\_Same\_TID_2$ and $L\_Diff\_TID_2$ is shown in Figure 16-(a) while Figure 16-(b) shows the corresponding bitmap of L_2.

For the following while loop, as shown in Figure 8, we generate L_n based on L_{n-1}, $n \geq 3$, until $L_n = \emptyset$. In order to generate candidates C_n, $n \geq 3$, we need to create tables for frequent itemsets. We define two tables as $FTable$ (*i.e.*,

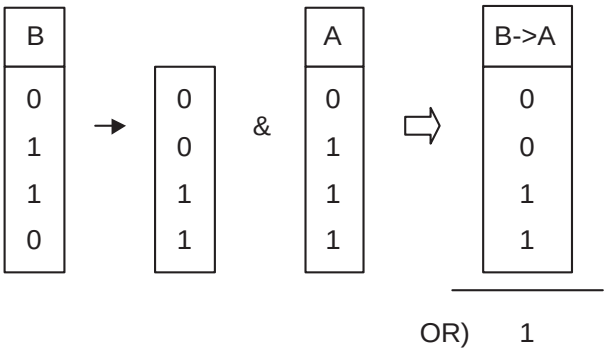


Figure 14: The generation of B->A for $CID = 1$ and different $TIDs$ (Case 2)

CID	C2
1	A -> A A -> B A -> D A -> F B -> A B -> B B -> D B -> F D -> A D -> B D -> D D -> F F -> D F -> F F -> A

CID	C2
2	
3	
4	B -> A D -> B D -> F D -> A F -> A

Figure 15: The result of C_2 for different $TIDs$ (Case 2)

L2		Support
$L\_Same\_TID_2$	A B A F B F	3 3 4
$L\_Diff\_TID_2$	B -> A D -> B D -> F D -> A F -> A	2 2 2 2 2

(a)

AB	AF	BF	B->A	D->B	D->F	D->A	F->A
0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0
1	1	1	1	1	1	1	0
0	1	0	1	0	1	1	1

(b)

Figure 16: The result of L_2: (a) the counts of L_2; (b) the corresponding bitmaps of L_2 for $CID = 1$.

Front	Rear
B	A
D	B
D	F
D	A
F	A

(a)

Front	Rear
B	A
D	B
D	F
D	A
F	A

(b)

Figure 17: Related tables of L_2: (a)$FTable$; (b)$RTable$

the Front Table) and $RTable$ (*i.e.*, the Rear Table). We split the itemset (a_1->a_2->... ->a_{n-1}->a_n) into two different sub-itemsets ((a_1->a_2->... ->a_{n-1}), (a_n)) and ((a_1), (a_2->... ->a_{n-1}->a_n)) and put them into the two tables. For example, if we have an itemset D->B->A->C, the itemset can be split by the first ->and the last ->. This will produce ((D->B->A), (C)) and ((D), (B->A->C)). Tables for this example are shown in Figure 17.

As described in the process of the generation of L_2, there are two cases for the generation of L_n, $n \geq 3$. For Case 1, itemsets with the same TID, in the first step, because any subset of a frequent itemset is also frequent, we can prune the useless items before calling procedure $L\_Same\_TID\_P$. The pruning procedure is shown in Figure 18. For example, as shown in Figure 19-(a), we have three $L\_Same\_TID_2$ itemsets, AB, AF, BF. They are used to generate $C\_Same\_TID_3$. When we want to find out the combinations of $C\_Same\_TID_3$, we do not need to generate ABD, ADF, BDF. They will never be the frequent itemsets, because item D is not include in the $L\_Same\_TID_2$ itemsets. Therefore, we do not directly compute the combinations of three items from L_1 (*i.e.*, A, B, D, and E), but compute the combinations of three items from A, B, and F. The number of candidates can be reduced from $C_3^4 = 4$ to $C_3^3 = 1$. First, we check AB and mark A and B with 1.

function *Reduce_C_Same_TID* ($L\_Same\_TID_{n-1}$, n)
{
  foreach itemset $I \in L\_Same\_TID_{n-1}$ **do {**
    foreach item $i \in$ I **do**
      $Y = Y \bigcup \{i\}$; **}**
  return (Y); **}**

Figure 18: function *Reduce_C_Same_TID*

Then, we check AF and mark A and F with 1. Third, we check BF and mark B and F with 1. The marking bit 1 means that the item exists in $L\_Same\_TID\_2$. After that, we apply an inner OR operation on these items. If the final result is 1, it means that this item is needed. In the second step, we use the same method as illustrated in the process of the generation of $L\_Same\_TID_2$. This is shown in Figure 19-(b). Finally, we obtain $C\_Same\_TID_3$ as shown in Figure 20.

For Case 2, itemsets with different $TIDs$, there are three subcases in this case of the candidate generation. We use the $FTable$ and $RTable$ to support them. The diagram for these sub-cases is shown in Figure 21.

For subcase A, and for the table shown in Figure 17, we find out the itemsets where $FT\_Front = RT\_Rear$. Then, we check if the itemset RT_Front->FT_Rear is frequent or not (*i.e.*, if it is in $CheckSet$). If this does not stand, it means that the subsequence RT_Front->FT_Rear of this itemset RT_Front->RT_Rear->FT_Rear is not frequent and we can drop this candidate. After the above process, we apply the S-step transformation to transform the itemset RT_Front->RT_Rear, which is from the $RTable$. Next, we apply an outer AND operation on it and the itemset FT_Front->FT_Rear, which is from $FTable$. Therefore, the candidate itemset RT_Front->RT_Rear->FT_Rear is generated. We apply an inner OR operation on this

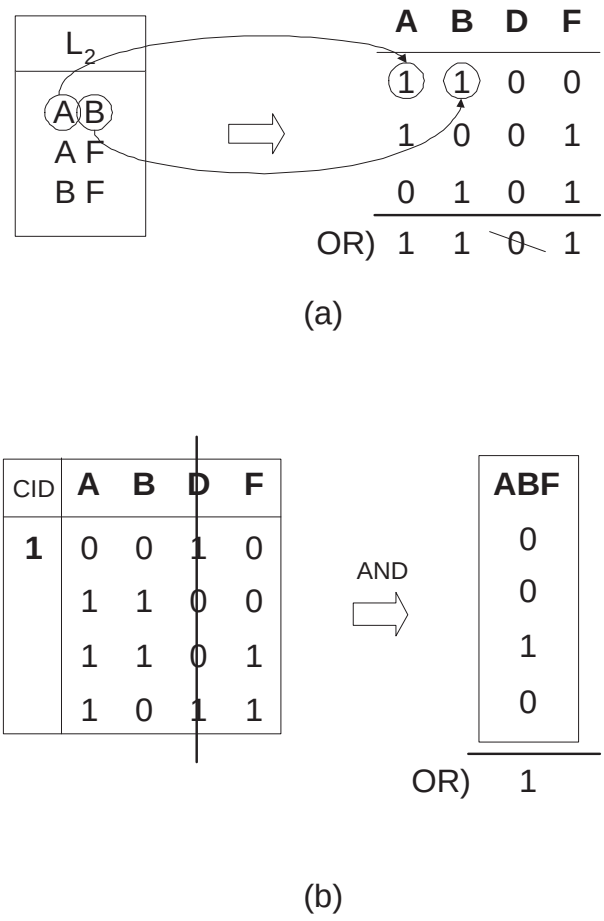


Figure 19: The generation of C_3: (a) the prepruning process; (b) the generation of C_3 using AND operation for $CID = 1$ and the same TID (Case 1).

CID	C3
1	A B F
2	A B F
3	A B F
4	A B F

Figure 20: The result of C_3 for the same TID (Case 1)

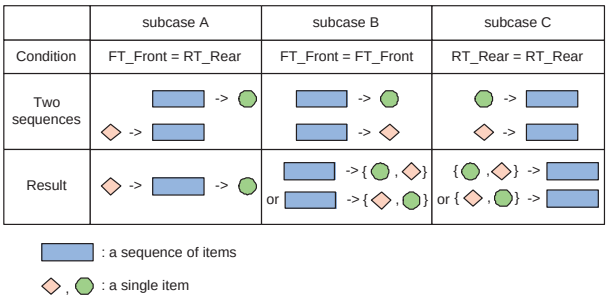


Figure 21: Three subcases for the case of different $TIDs$

candidate to obtain its support. The procedure is shown in Figure 22. For example, we find that the itemset B (RT_Rear) in $RTable$ is equal to the itemset B (FT_Front) in $FTable$ and the itemset D->A (RT_Front->FT_Rear) is in $CheckSet$. The process for this example is shown in Figure 23. Since that D->A has been mined and is known as frequent, we apply the S-step transformation to transform the itemset D->B = [0110] to become [0011]. Next, we apply an outer AND operation on the result and the itemset B->A = [0111]. Finally, the candidate D->B->A = [0011] is generated. After the inner OR operation, we obtain that the support of this candidate for $CID = 1$ is 1.

For subcase B, and for the table shown in Figure 17, we find out any two of the itemsets where $FT\_Front = FT\_Front$. We apply an outer AND operation on these itemsets to generate candidate FT_Front ->(FT_Rear_1 + FT_Rear_2). The pattern (FT_Rear_1+FT_Rear_2) means the synthesis of two itemsets FT_Rear_1 and FT_Rear_2, such as, $B + F = BF$ and $BD + BF = BDF$. Then, we apply an inner OR operation on the result. The procedure is shown in Figure 24. For example, we find that the itemset D is in FT_Front and D appears more than once and two of these itemsets are D->B = [0110] and D->F = [0011]. The process for this example is

```
procedure L_Diff_TID_CaseA() {
 foreach itemset FI ∈ FTable do {
  foreach itemset RI ∈ RTable do {
   if ((FI.front = RI.rear) and
        (not(Prune(RI.front->FI.rear)))) then {
    Candidate = RI.front->RI.rear->FI.rear;
    foreach CID c do {
     if (Candidate.length = n) do {
      Transformation(FI[c]);
      isTureTrans = False;
      foreach TID t do {
       Candidate[t] = Iset[t] AND RI[c,t];
       isTrueTrans =
            Candidate[t] OR isTrueTrans; }
      if (isTrueTrans = True) then
       Candidate.count = Candidate.count+1; } }
     if (Candidate.count ≥ min_s) then
       L_Diff_TID_n =
            L_Diff_TID_n ⋃ {Candidate}; } } } }
```

Figure 22: Procedure $L\_Diff\_TID\_CaseA$

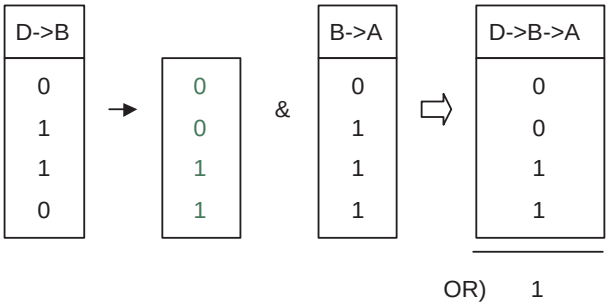


Figure 23: The generation of D->B->A (subcase A) for $CID = 1$ and different $TIDs$ (Case 2)

shown in Figure 25. We apply an outer AND operation on these two itemsets and a new candidate D->BF = [0010] will be generated. After the inner OR operation, the support of this candidate for $CID = 1$ is 1.

For subcase C, and for the table shown in Figure 17, we find out any two of the itemsets where $RT\_Rear = RT\_Rear$. We synthesize the itemsets RT_Front_1 and RT_Front_2 and generate the itemset ($RT\_Front_1 + RT\_Front_2$). We try to find whether this itemset exists in $L\_Same\_TID_n$ or not. If we can not find the

```
procedure L_Diff_TID_CaseB() {
/* FI_1 + FI_2 means the synthesis of two itemsets.*/
 foreach itemset FI_1 ∈ FTable do {
  foreach itemset FI_2 ∈ FTable do {
   if (FI_1.front = FI_2.front) then {
    Candidate = FI_1 + FI_2;
    if (Candidate.length = n) then {
     foreach CID c do {
      isTrueTrans = False;
      foreach TID t do {
       Candidate[t] = FI_1[t] AND FI_2[t];
       isTrueTrans =
          Candidate[t] OR isTrueTrans; };
      if (isTureTrans = True) then
       Candidate.count =
          Candidate.count + 1; };
     if (Candidate.count ≥ min_s) then {
      L_Diff_TID_n =
          L_Diff_TID_n ⋃ {Candidate};
      CheckSet =
          CheckSet ⋃ {Candidate}; } } } } };
```

Figure 24: Procedure $L\_Diff\_TID\_CaseB$

itemset, we do not need to generate this candidate; otherwise, we apply the S-step transformation on the itemset. Next, we apply an outer AND operation on it and the itemset RT_Front_1->RT_Rear. (We can use itemset RT_Front_2 ->RT_Rear instead of RT_Front_1 ->RT_Rear, because RT_Front_1 and RT_Front_2 lead to the same result RT_Rear. Since we want to generate the candidate (RT_Front_1 + RT_Front_2) ->RT_Rear, we obtain the same result no matter which itemset (RT_Front_1 or RT_Front_2) is used to do this operation.) Then, we apply an inner OR operation on the result. The procedure is shown in Figure 26.

For example, we find that the itemset A is in RT_Rear and A appears more than once and two of these itemsets are B->A = [0011] and F->A = [0001]. The process for this example is shown in Figure 27. We first synthesize B and F to BF. Since BF = [0010] is in $L\_Same\_TID_2$,

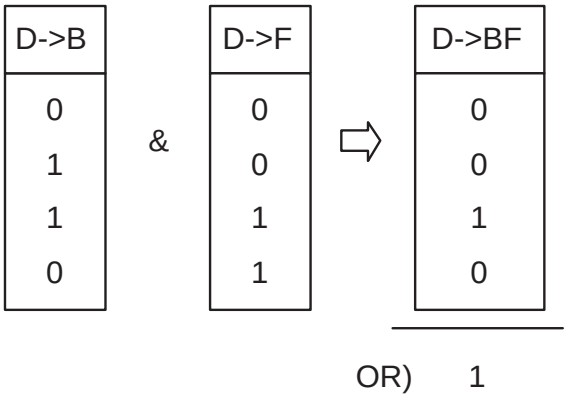


Figure 25: The generation of D->BF (subcase B) for $CID = 1$ and different $TIDs$

we apply the S-step transformation on the itemset. Next, we apply an outer AND operation on the result and B->A (or F->A). Finally, the candidate BF->A = [0001] is generated. After the inner OR operation is performed, the support of this candidate for $CID = 1$ is 1. Thus, L_3 and its count for $CID = 1$ is shown in Figure 28-(a), and Figure 28-(b) shows the corresponding bitmaps.

Up to this point, we have generated L_3 for this example. $FTable$ and $RTable$ tables for L_3 is shown in Figure 29. Similarly, we continue to process the steps inside the while loop to generate $FTable_n$ and $RTable_n$ tables for L_n until $L_n = \emptyset$.

4 Performance

In this Section, we study the performance of the proposed algorithm. As stated in [14], the performance of the PrefixSpan algorithm [14] is better than the GSP [15] and the FreeSpan [8] algorithms in terms of the processing time. On the other hand, as stated in [4], the performance of the SPAM algorithm [4] is better than the PrefixSpan and the SPADE [20] algorithms in terms of the processing time. Therefore, in this paper, we only make a comparison of our proposed algorithm with the SPAM algorithm.

```
procedure L_Diff_TID_CaseC() {
/* RI_1 + RI_2 means the synthesis of two itemsets.*/
  foreach itemset RI_1 ∈ RTable do {
    foreach itemset RI_2 ∈ RTable do {
      if (RI_1.rear = RI_2.rear) then {
        Candidate = RI_1 + RI_2;
        if (Candidate.length = n) then {
         if (Candidate.head ∈ L_Same_TID_{n-1}) then{
           foreach CID c do {
            Transformation(Candidate.head);
            isTrueTrans = False;
            foreach TID t do {
              Candidate[t] =
                    Candidate.head[t] AND RI_1[t];
              isTrueTrans =
                    Candidate[t] OR isTrueTrans; };
            if (isTrueTrans = True) then
              Candidate.count =
                 Candidate.count + 1; };
           if (Candidate.count ≥ min_s) then {
            L_Diff_TID_n =
               L_Diff_TID_n ⋃ {Candidate};
            CheckSet =
               CheckSet ⋃ {Candidate};} } } } } };
```

Figure 26: Procedure $L\_Diff\_TID\_CaseC$

L3		Support
L_Same_TID $_3$	A B F	4
L_Diff_TID $_3$	B F -> A D -> B F D -> B -> A D -> F -> A	2 2 2 2

(a)

ABF	BF->A	D->BF	BF->A	D->B->A	D->F->A
0	0	0	0	0	0
0	0	0	0	0	0
1	0	1	0	1	0
0	1	0	1	1	1

(b)

Figure 28: The result of L_3: (a) the counts of L_3; (b) the corresponding bitmap of L_3 for $CID = 1$.

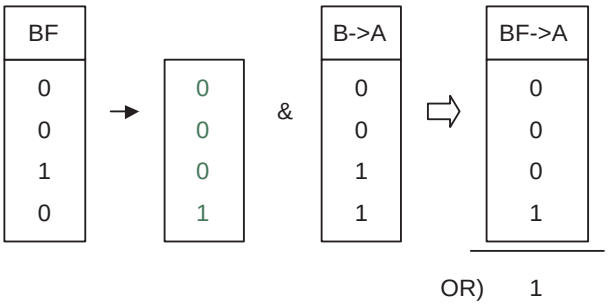


Figure 27: The generation of BF->A (subcase C) for $CID = 1$ and different $TIDs$

Front	Rear
D->B	A
D->F	A
D	BF
BF	A

(a)

Front	Rear
D	B->A
D	F->A
D	BF
BF	A

(b)

Figure 29: Related tables of L_3: (a) $FTable$; (b) $RTable$

4.1 Generation of Synthetic Data

We generate the synthetic datasets using the same data model in [3], which also has been used in [4, 8, 11, 14, 15]. The synthetic data is to simulate the behaviors of customers buying patterns in the retail market. The parameters used in the generation program of the synthetic data are shown in Table 3.

Table 3: Parameters

Parameter	Meaning
D	Number of customers (= size of the database)
C	Average number of transactions per customer
T	Average number of items per transaction
S	Average length of maximal potentially frequent sequences
I	Average size of Itemsets in maximal potentially frequent sequences
N_S	Number of maximal potentially frequent sequences
N_I	Number of maximal potentially frequent itemsets
N	Number of items

First, we determine the number of transactions for the customer, and the average size of each transaction for this customer. The number of transactions is decided from a Poisson distribution with mean μ equal to C, and the average size of the transaction is decided from a Poisson distribution with μ equal to T.

Then, we assign items to the transactions of the customer. Each customer is assigned with a series of potentially frequent sequences. If the frequent sequence on hand does not fit in the customer-sequence, *i.e.*, the size of this customer-sequence exceeding the predefined size of the customer-sequence, the itemset is still added into the customer-sequence, anyway, in half of the cases, and the itemset is moved to the next customer-sequence in the rest of the cases.

Frequent sequences are chosen from a table $\mathcal{T}$ of maximal potentially frequent sequences. There are N_S sequences in $\mathcal{T}$. A sequence in $\mathcal{T}$ is generated by first picking the number of itemsets in the sequence from a Poisson distribution with mean μ equal to S. Itemsets in the first sequence are chosen randomly from a table of maximal potentially frequent itemsets. The table of maximal potentially frequent itemsets is similar to the table of maximal potentially frequent sequences. Since the sequences often have common items, *i.e.*, popular products being bought together usual, some fraction of itemsets in subsequent sequences are chosen from the previous sequence generated. We use an exponentially distributed random variable with mean equal to the *correlation level* to decide this fraction for each itemset. The remaining itemsets are picked at random. In our experiments, the correlation level was set to 0.25.

Each sequence in $\mathcal{T}$ has a weight which determines the probability that this sequence will be picked. This weight is picked from an exponential distribution with unit mean. The next sequence to be added into a customer sequence is chosen from $\mathcal{T}$ according to their weights. The larger the weight of one sequence is, the higher probability of this sequence being picked is.

Since all the items in a frequent sequence are not always bought together, we assign each sequence in $\mathcal{T}$ a *corruption level* c. When adding a sequence to a transaction, we keep dropping an item from the sequence as long as a uniformly distributed random number between 0 and 1 is less than c. The corruption level for a sequence is fixed and is obtained from a normal distribution with mean 0.75 and variance 0.1. Each transaction is

stored in a file system with the form of <*customer identifier, transaction identifier, item*>.

4.2 Simulation Results

In our simulation result, the performance measures which we concern about are the CPU execution time of the computation and the sensitivity to parameters. First, we make a comparison of the execution time for six synthetic data sets(six cases). These data sets are used to simulate the sales volume of the stores within one month. The first two cases are small-sized shops, the second two cases are medium-sized stores, and the third two cases are large-sized stores. Then, we make a comparison of the execution time under the setting of different parameters. For the comparison of the first performance measure, N_S, N_I, S, I were set to 5000, 25000, 4, and 1.25, respectively. For the comparison of the second performance measure, N_S, N_I, S, I were set to 5000, 25000, 10, and 10, respectively.

4.2.1 Time Scalability

The first performance measure of our experiments was to evaluate the execution time of these algorithms to mining various synthetic data sets. These data sets used in the data generator are shown in Table 4. The comparisons of execution time under different minimum supports are shown in Figure 30.

From these figures, we show that the proposed algorithm always requires shorter execution time than the SPAM algorithm no matter what size the data set is or what value the minimum support is. The reason is that these two algorithms both use the bitwise operations to count the support. However, our algorithm generates fewer number of candidates than the SPAM algorithm. Therefore, our algorithm needs less time to check whether one

Table 4: The synthetic data sets for three different kinds of stores

Case	Name	D	C	T	N
1	D1.5C10T3N0.5	1.5	10	3	0.5
2	D4C10T3N0.5	4	10	3	0.5
3	D25C6T7N5	25	6	7	5
4	D30C7T5N10	30	7	5	10
5	D150C3T20N10	150	3	20	10
6	D180C3T15N20	180	3	15	20

Case	Size	Meaning
1	473KB	small-sized store
2	1.35MB	small-sized store
3	10.3MB	medium-sized store
4	12.4MB	medium-sized store
5	24.2MB	large-sized store
6	27.0MB	large-sized store

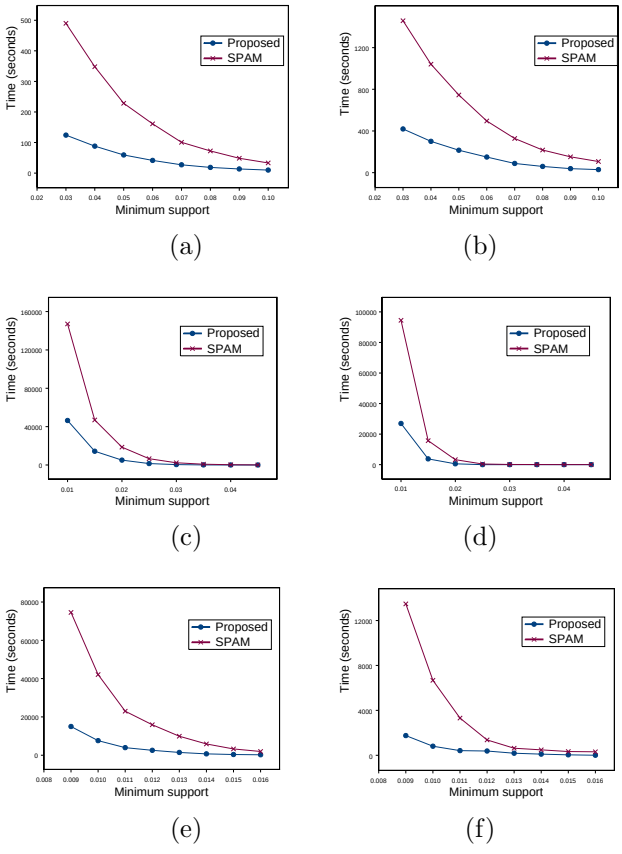


Figure 30: The comparisons of the execution time under different minimum supports: (a) Case 1: D1.5C10T3N0.5; (b) Case 2: D4C10T3N0.5; (c) Case 3: D25C6T7N5; (d) Case 4: D30C7T5N10; (e) Case 5: D150C3T20N10; (f) Case 6: D180C3T15N20.

Table 5: The synthetic data sets with various parameters

Case	Name
A	D?C10T10N10
B	D15C?T10N10
C	D15C10T?N10
D	D15C10T10N?

candidate is frequent or not than the SPAM algorithm. Generally speaking, in both of algorithms, the higher the *min_sup* is, the shorter the execution time is. This is because when the *min_sup* gets higher, the number of frequent itemsets generated becomes fewer. It leads to fewer number of candidates.

4.2.2 Sensitivity to Parameters

We studied the sensitivity of the SPAM algorithm and our algorithm to the change of some parameters. The data sets(cases A, B, C, D) are shown in Table 5, and *min_sup* is set to 0.04. The results are shown in Figure 31.

From Figure 31-(a)(case A), we observe that when the number of customers increases, the execution time increases slowly. The reason is that, when the number of customers increases, the number of candidates also increases slowly. However, in Figures 31-(b) and 31-(c), the execution time increases rapidly, when the number of transactions and the number of items per customer increase. The reason is that, in both cases(cases B and C) if the sequence of one customer is long, the number of candidates will increase rapidly. Therefore, they have to spend a lot of time to count the candidates. From Figure 31-(d), we observe that when the number of items increases, the execution time decreases. The reason is that since there are too many items, the count for each candidate is distributed too and will not create too many large itemsets. Therefore, the number of candidates decreases and this leads to short execution time.

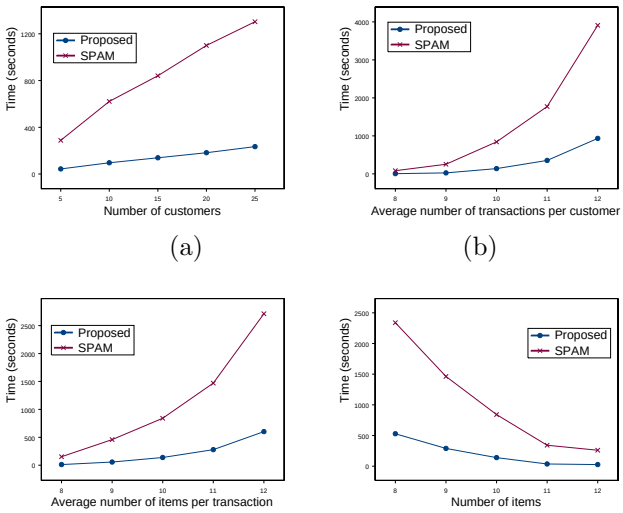


Figure 31: The comparisons of the execution time under different number of customers: (a) Case A: D?C10T10N10; (b) Case B: D15C?T10N10; (c) Case C: D15C10T?N10; (d) Case D: D15C10T10N?.

From the above comparisons, we show that our algorithm is always faster than the SPAM algorithm. Note that the key issues of effecting performance of mining sequential patterns are candidate generation and counting. Our algorithm and the SPAM algorithm both use the bitwise operation for the counting part, so the candidate generation part becomes the main factor that affects the performance of mining. Since the process of the candidate generation of our algorithm is more efficient than the SPAM algorithm and our algorithm generates fewer candidates than the SPAM algorithm, our algorithm could provide better performance than the SPAM algorithm.

5 Conclusion

In this paper, we have proposed a bitmap-based algorithm for mining sequential patterns. Basically, there are two differences between mining association rules and sequential patterns: candidate generation and counting. The situations

are more complex in mining sequential patterns than in mining association rules. That is why we could not apply the technique of mining association rules directly to the problem of mining sequential patterns. Therefore, the algorithms for mining sequential patterns must be redesign. The most time-consuming task in the process of mining sequential patterns is to count the candidates. We find that using the bitwise operations can reduce a lot of time when counting. Therefore, we proposed a new bitmap-based algorithm for mining sequential patterns efficiently. The basic idea is to use the similar candidate generation method proposed in the GSP algorithm to reduce the number of candidates and the similar counting method proposed in the SPAM algorithm to reduce the time of counting candidates. We have made a comparison of execution time between the proposed algorithm and the SPAM algorithm by simulation. We have conducted several experiments using different synthetic datasets, including the datasets for different kinds of stores and the datasets for different parameters of the synthetic data. The simulation results have shown that our algorithm could provide better performance than the SPAM algorithm in terms of the processing time no matter whether the size of the datasets is large or small.

References

[1] R. Agrawal and J. C. Shafer, "Parallel Mining of Association Rules," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 962–969, Dec. 1996.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pp. 487–499, 1994.

[3] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. of the 11th Int. Conf. on Data Engineering*, pp. 3–14, 1995.

[4] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential Pattern Mining Using a Bitmap Representation," *Proc. of the 8th Int. Conf. on Knowledge Discovery and Data Mining*, pp. 429–435, 2002.

[5] Y. L. Chen, S. S. Chen, and P. Y. Hsu, "Mining Hybrid Sequential Patterns and Sequential Rules," *Information Systems*, Vol. 27, No. 5, pp. 345–362, 2002.

[6] G. Gardarin, P. Pucheral, and F. Wu, "Bitmap Based Algorithms for Mining Association Rules," *Proc. of the 14th Bases de Donnes Avances*, pp. 157–176, 1998.

[7] M. Garofalakis, R. Rastogi, and K. Shim, "Mining Sequential Patterns with Regular Expression Constrains," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 14, No. 3, pp. 530–552, May/June 2002.

[8] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. C. Hsu, "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining," *Proc. of the 6th Int. Conf. on Knowledge Discovery and Data Mining*, pp. 335–359, 2000.

[9] M. Y. Lin and S. Y. Lee, "Incremental Update on Sequential Patterns in Large Databases by Implicit Merging and Efficient Counting," *Information Systems*, Vol. 29, No. 5, pp. 385–404, July 2004.

[10] M. Y. Lin and S. Y. Lee, "Fast Discovery of Sequential Patterns through Memory Indexing and Database Partitioning," *Journal of Information Science and Engineering*, Vol. 21, No. 1, pp. 109–128, Jan. 2005.

[11] M. Y. Lin, S. Y. Lee, and S. S. Wang, "DELISP: Efficient Discovery of Generalized Sequential Patterns by Delimited Pattern-Growth Technology," *Proc. of the 6th Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pp. 198–209, 2002.

[12] F. Masseglia, P. Poncelet, and M. Teisseire, "Incremental Mining of Sequential Patterns in Large Databases," *Data and Knowledge Engineering*, Vol. 46, No. 1, pp. 97–121, July 2003.

[13] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, "Mining Sequential Patterns by Prefix-Projected Pattern Growth: The PrefixSpan Approach," *IEEE Tran. on Knowledge and Data Engineering*, Vol. 16, No. 11, pp. 1424–1440, Jan. 2004.

[14] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, "Prefixspan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," *Proc. of the Int. Conf. on Data Engineering*, pp. 215–224, April 2001.

[15] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proc. of the 5th Int. Conf. on Extending Database Technology*, pp. 3–17, 1996.

[16] C. Y. Wang, T. P. Hong, and S. S. Tseng, "Maintenance of Discovered Sequential Patterns for Record Modification," *Proc. of the Int. Computer Symp. Workshop on Artificial Intelligence*, pp. 1682–1687, 2002.

[17] W. Wang, J. Yang, and P. S. Yu, "Mining Patterns in Long Sequential Data with Noise," *ACM SIGKDD Explorations Newsletter*, Vol. 2, No. 2, pp. 28–33, Dec. 2000.

[18] J. M. Wei, W. G. Yi, and M. Y. Wang, "Novel Measurement for Mining Effective Association Rules," *Knowledge-Based Systems*, Vol. 19, No. 8, pp. 739–743, Dec. 2006.

[19] J. Yang, W. Wang, P. S. Yu, and J. Han, "Mining Long Sequential Patterns in a Noisy Environment," *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 406–417, 2002.

[20] M. J. Zaki, "Efficient Enumeration of Frequent Sequences," *Proc. of the 7th Int. Conf. on Information and Knowledge Management*, pp. 68–75, 1998.

[21] M. J. Zaki, "Parallel Sequence Mining on Shared-Memory Machines," *Journal of Parallel and Distributed Computing*, Vol. 61, No. 3, pp. 401–426, March 2001.