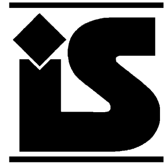
PERGAMON

Information Systems 27 (2002) 345–362



www.elsevier.com/locate/infosys

Mining hybrid sequential patterns and sequential rules

Yen-Liang Chen[a,*], Shih-Sheng Chen[a], Ping-Yu Hsu[b]

[a] *Department of Information Management, National Central University, Chung-Li, Taiwan 320, Republic of China*

[b] *Department of Business Administration, National Central University, Chung-Li, Taiwan 320, Republic of China*

Received 10 January 2001; received in revised form 29 June 2001; accepted 18 December 2001

Abstract

The problem addressed in this paper is to discover the frequently occurred sequential patterns from databases. Basically, the existing studies on finding sequential patterns can be roughly classified into two main categories. In the first category, the discovered patterns are continuous patterns, where all the elements in the pattern appear in consecutive positions in transactions. The second category is to mine discontinuous patterns, where the adjacent elements in the pattern need not appear consecutively in transactions. Although there are many researches on finding either kind of patterns, no previous researches can find both of them. Neither can they find the discontinuous patterns formed of several continuous sub-patterns. Therefore, we define a new kind of patterns, called hybrid pattern, which is the combination of continuous patterns and discontinuous patterns. In this paper, two algorithms are developed to mine hybrid patterns, where the first algorithm is easy but slow while the second complicated but much faster than the first one. Finally, the simulation result shows that our second algorithm is as fast as the currently best algorithm for mining sequential patterns. © 2002 Elsevier Science Ltd. All rights reserved.

Keywords: Data mining; Pattern; Association rule; Sequential data

1. Introduction

Data mining is to extract implicit, previously unknown and potentially useful information from database [1]. Many approaches have been proposed to extract information. One of the most important one is mining association rules. This approach was first introduced in [2], and can be stated as follows:

Given a database of sales transaction, an association rule is an implication of the form $X \rightarrow Y$, where X and Y are subset of items and $X \cap Y = \emptyset$. This rule indicates that if we bought X then it is likely that we will also purchase Y. To find the association rules from database, we firstly need to calculate the supports of itemsets X and $X \cup Y$, where the support of X is the number of transactions in database containing X. Next, we must determine if the itemsets $X \cup Y$ and X are frequent, where an itemset is frequent if its support is greater than or equal to the user-specified minimum support (called *minsup*.) If they are frequent, we then compute the confidence of the rule by formula $support(X \cup Y)/support(X)$. Finally, the rule $X \rightarrow Y$ holds if its confidence is greater than or equal to the user-specified minimum confidence (called *minconf*.)

Many of previous researches in association rules assume that the items bought in a transaction are unordered [2–7]. In other words, the items in a

*Corresponding author. Tel.: +886-3-4267266; fax: +886-3-4254604.

E-mail address: ylchen@im.mgt.ncu.edu.tw (Y.-L. Chen).

0306-4379/02/$ - see front matter © 2002 Elsevier Science Ltd. All rights reserved.
PII: S 0 3 0 6 - 4 3 7 9 (0 2) 0 0 0 0 8 - X

transaction can be denoted by a set of items. Although this assumption seems reasonable, we may encounter situations where order is important. For example, Agrawal and Srikant [8] point out that not only the items bought but also the transaction times are needed to mine customer behaviors. Another example [9–12] involves mining traversal patterns in web surfing. Still many other examples exist, such as shopping sequences in malls, traveling sequences in a tour, plan failure predictions [13] and the genetic sequences of gene [14]. All these problems need to deal with ordered data, and many researches have been done [15,16,8–10,17–20,12,21,22,14] to discover meaningful patterns from them.

In general, finding patterns from sequential data in the literature can be classified into three main categories. The first category is finding similar patterns, where a pattern is similar to another pattern if the distance or correlation between them is within a user-defined threshold [15–17,23]. The second category is finding periodic patterns, which are to discover the cyclic patterns in time-stamped databases [18,20]. Finally, the last one is finding frequent patterns, which are to enumerate those patterns with supports exceeding some given threshold [8,9,19,24,11,12,21,22,13]. Among these three categories, finding frequent patterns is possibly the most popular and has been extensively studied because of its numerous applications in the areas such as web surfing, plan execution, event sequence and shopping order analysis. It can be further divided into sequential frequent pattern and non-sequential frequent pattern. The former [8,9,19,11,12,21,14,13] are linear subsequences occurred frequently in databases, while the latter [25,24,22] are non-linear subsequences such as partial-order-graphs, tree structures or graph structures.

In essence, previous researches use similar methods as those of mining association rules to find sequential pattern. The patterns are discovered from database by identifying those subsequences with supports no less than the given threshold. The found patterns indicate what are the frequently occurred subsequences that need our attention. For example, a user may visit site A, then site B and finally site C in order. And if many users have the same behavior, then the subsequence $\langle$ABC$\rangle$ forms a meaningful pattern.

Basically, the sequential patterns found by the previous researches can be classified into two major categories: continuous patterns and discontinuous patterns. In finding continuous patterns [15,16,9,17,23,14], we say a pattern $A = \langle a_1, a_2, \ldots, a_n \rangle$ occurs in a transaction of sequential data $B = (b_1, b_2, \ldots, b_m)$ if there exists integer i such that $a_1 = b_i, a_2 = b_{i+1}, \ldots, a_n = b_{i+n-1}$. In other words, all the elements in the pattern appear in consecutive positions of transactions. For example, the continuous pattern $\langle$AB$\rangle$ occurs in an ordered transaction of (A, B, K, F, P) but does not in (K, Q, A, D, B). In practice, difficulties may arise for continuous patterns. Let us use the following navigation sequences to explain:

- Sequence 1: A, B, Y, K, F.
- Sequence 2: A, B, K, F, P.
- Sequence 3: C, A, B, C, R, K, F.
- Sequence 4: K, Q, A, D, B.
- Sequence 5: F, G, B, M.

Suppose the minimum support threshold is 3. Then an interesting navigation pattern is first to visit sites A and B, and then visit sites K and F. But, between the visiting of sites A and B and that of K and F, there are a variable number of intermediate sites. This example indicates that a meaningful pattern may consist of several continuous sub-patterns that are not adjacent. If the traditional algorithm for finding continuous patterns is applied, we only find patterns $\langle$AB$\rangle$ and $\langle$KF$\rangle$ but without finding $\langle$AB*KF$\rangle$, where "*" means a variable number of intermediate elements.

The second category finds discontinuous patterns [8,11,12,21,26,13], where we say a discontinuous pattern $A = (a_1, a_2, \ldots, a_n)$ occurs in a transaction of ordered data $B = (b_1, b_2, \ldots, b_m)$ if there exists integer $i_1 < i_2 < \cdots < i_n$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots, a_n = b_{i_n}$. For example, $\langle$A*B*K*F$\rangle$, $\langle$B*K*F$\rangle$, $\langle$B*F$\rangle$ and $\langle$K$\rangle$ are some discontinuous patterns occurred in the above example.

Note that the continuous patterns such as $\langle$AB$\rangle$ and $\langle$KF$\rangle$ cannot be found by the algorithms for mining discontinuous patterns.

Similarly, neither can the discontinuous patterns such as ⟨B*K⟩ and ⟨A*B*K*F⟩ be found by the algorithms for continuous patterns. Most interestingly, neither types of algorithms can discover the hybrid patterns such as ⟨ABK*F⟩, ⟨AB*KF⟩ and ⟨AB*K*F⟩. Without these hybrid patterns, many important traversal relations will be lost. For example, in analyzing the customer footsteps in an on-line store, we may be interested in finding the patterns like

⟨referrer-address entry-page*product-page⟩,
⟨referrer-address entry-page*product-page*basket-page⟩ and
⟨referrer-address entry-page*product-page*basket-page purchasing-action⟩,

from which we can analyze the look-to-click rates, click-to-basket rates and basket-to-buy rates [27] for different area visitors starting at different entry pages. Besides, the hybrid pattern ⟨referrer-address entry-page* product-page basket-page⟩ indicates how many click-throughs are directly converted to basket placement. As another example, the hybrid pattern ⟨product-page details-page* purchasing-action⟩ tells us how many visitors who continuously watch the product-page and the related details-page will finally buy the product.

Therefore, not knowing the hybrid patterns means that some implicit, previously unknown and potentially useful knowledge are still hidden in the database, and this means a failure of the mining task. That motivates us to develop a method to discover hybrid patterns from database.

In this paper, we use a variable character "*" to denote a sequence of unknown items. The number of items represented by a "*" can be as small as zero or as large as any number allowed by database systems. In finding patterns from sequences, the pattern may contain many variable characters "*". Therefore, the hybrid patterns would look like as ⟨AB*KF⟩, ⟨AB*C*KF⟩ or ⟨A*BCD⟩.

From the discovered hybrid patterns, we can generate sequential rules. The generated rules are different from the previous ones in two ways.

(1) Not only do we have the traditional rule that reasons forwardly such as $X \rightarrow Y$, but also we can find the backward rules such as $X \leftarrow Y$, which means that if we currently visit Y then it is possible that we have visited X before.
(2) We divide the rules into two categories where the first is direct rule and the second is indirect rule. A direct rule of the form $X \rightarrow Y$ means that we visit Y immediately after we visited X. An indirect rule of the form $X \rightarrow^{*} Y$ means that Y is eventually visited after X, but the detail visiting order is unspecified.

In Section 2, we present an algorithm for finding hybrid patterns. In Section 3, the algorithm is further improved by using complicated data structures to reduce the number of phases of the algorithm and the number of scans through database. Next, we discuss how to generate sequential rules from hybrid patterns in Section 4. Finally, the evaluation of the algorithms' performances is done in Section 5 and the conclusion is given in Section 6.

2. The proposed algorithm

Let $I = \{i_1, i_2, \ldots, i_m\}$ denote all items in database D. Each transaction $T = \langle e_1, e_2, \ldots, e_n \rangle$ is a sequence of items, where $e_i \in I$ is the ith item. Let "*" denote a subsequence of any length, including zero length. A "*" can be replaced by any sequence. For example, a pattern of ⟨A*B⟩ can denote ⟨AB⟩, ⟨ACB⟩, ⟨ADDCB⟩ and so on. We are going to find patterns $X = \langle x_1, x_2, \ldots, x_n \rangle$ from databases. Basically, patterns are sequences of items satisfying the following constraints:

(1) $x_i \in I \cup \{\text{“}*\text{”}\}$.
(2) $x_1 \in I$ and $x_n \in I$.
(3) For $1 < i < n$, if $x_i =$ "*" then $x_{i+1} \in I$ and $x_{i-1} \in I$.

According to this definition, ⟨ABC⟩, ⟨A*BC⟩ and ⟨A*B*C⟩ are patterns, but ⟨*ABC⟩, ⟨*AB**C*⟩ and ⟨A*B**C⟩ are not patterns. If the following conditions are satisfied then we say that a transaction T contains the pattern ($X \subset T$):

(1) For each known item x in X, there is a corresponding matched item x in T.

(2) If x and y are two known items in X and x precedes y in X, then the corresponding item matched with x must also precede the corresponding item matched with y in T.
(3) If there is no interleaving "*" between items x and y in X, then the corresponding positions in T must be continuous.

Example 1. If $X = \langle A^*BC \rangle$ and $T = \langle BACABCC \rangle$ then we have two matches satisfying $X \subset T$. In the first match, the first, second and third known items in X match the second, fifth and sixth items in T, respectively. As to the second match, we map the first, second and third known items in X to the fourth, fifth and sixth items in T, respectively. But, when we set $X = \langle A^*BAC \rangle$ then $X \not\subset T$.

Since a pattern X may occur several times in a transaction T, we need to determine the number of matches of $X \subset T$. This counting is not necessary for the previous researches on transaction databases, because they view a transaction as a set of items and hence they only need to determine if a pattern exists in a transaction. On the contrary, because we view a transaction as a sequence with repeated items, we need to compute the supports of patterns according to their total occurrences. The following function is designed to determine the number of occurrences of X in T, where $X(i)$ is the ith known item in X, $T(j)$ is the jth item in T and $number(X,1,T,1)$ is the main program to activate the computation.

Example 2. Suppose we have $X = \langle AB^*B \rangle$ and $T = \langle BABAABCB \rangle$. By running the function $number(X,1,T,1)$, we will call the functions $number(X,2,T,3)$, $number(X,2,T,5)$, $number(X,2,T,6)$, because $X(1)$ matches $T(p)$ when $p = 2$, 4 and 5. In executing $number(X,2,T,3)$, we will activate $number(X,3,T,4)$ because of $X(2) = T(3)$. After that, the execution of $number(X,3,T,4)$ will increase the value of *num* by 2, since $X(3)$ matches $T(p)$ when $p = 6$ and $p = 8$. As to the execution of $number(X,2,T,5)$, we just return to the calling program since $X(2)$ is not equal to $T(5)$. Finally, the execution of $number(X,2,T,6)$ will activate the function $number(X,3,T,7)$, which will increase the value of *num* by 1 when $p = 8$. Thus, the final value of *num* is 3.

For a pattern X, the number of known elements in X is called the fixed length of X. Similarly, we define the variable length of pattern X as the number of "*" in X. For example, we have the fixed length as 3 and variable length as 1 if $X = \langle A^*BC \rangle$, and fixed length as 4 and variable length as 2 if $X = \langle A^*BB^*C \rangle$. Throughout the paper we use $X_{k,r}$ to denote a pattern X with fixed length k and variable length r. For example, $\langle A^*BC \rangle$ and $\langle A^*BB^*C \rangle$ can be represented by $X_{3,1}$ and $X_{4,2}$, respectively.

By definition, if X is a pattern, then it will have no leading "*" and trailing "*". But, if we write it as $/X$, it means that the leading "*" is added. Similarly, $X/$ means that the trailing "*" is added,

```
Function number(X, i, T, i)
/* num denotes the number of matches of X ⊂ T */
/* n denotes the number of elements in T */
/* we assume that there is an interleaving "*" between X(0) and X(1) */
If j = n + 1 then return
If i = 1 and j = 1 then num = 0
If there is no interleaving "*" between X(i − 1) and X(i) then
    If X(i) = T(j) and X(i) is the last item in X then num + +
    Else if X(i) = T(j) then number(X, i + 1, T, j + 1)
Else if there is an interleaving "*" between X(i − 1) and X(i) then
    For p = j to n
        If X(i) = T(p) and X(i) is the last item in X then num + +
        Else if X(i) = T(p) then number(X, i + 1, T, p + 1)
    Endfor
Return
```

Transaction	Item
001	AECAG
002	CGBAGD
003	KBGAC
004	BIGHG
005	BGACAFJ

Fig. 1. A sample database.

and $/X/$ for both. For example, if $X = \langle A^*BB^*C \rangle$, then $/X = \langle {}^*A^*BB^*C \rangle$, $X/ = \langle A^*BB^*C^* \rangle$ and $/X/ = \langle {}^*A^*BB^*C^* \rangle$.

In database D, the total number of occurrences of pattern X is called the support of X, which is denoted by *support*(X). Let *minsup* be the user-specified minimum threshold for support. A pattern X is frequent if it satisfies $support(X) \geqslant minsup$.

Let $L_{k,r}$ denote the set of all frequent patterns with fixed length k and variable length r. Similarly, we define $C_{k,r}$ as the set of all candidates patterns with fixed length k and variable length r. Note that, a candidate pattern in $C_{k,r}$ may not be a frequent pattern in $L_{k,r}$, but every frequent pattern in $L_{k,r}$ must exist in $C_{k,r}$. Therefore, $C_{k,r}$ is a superset of $L_{k,r}$.

Our algorithm proceeds in phases, where the kth phase is to find all frequent patterns of fixed length k from the candidate patterns of fixed length k. In the first phase, we need to find $C_{1,0}$, i.e., all candidate patterns with only one known element and zero "*". For the database in Fig. 1, the corresponding $C_{1,0}$ is shown in Fig. 2. By scanning the database sequentially and determining how many occurrences of those patterns, we can get the supports of all patterns in $C_{1,0}$. If the minimum support threshold is 2, then we can get the resulting frequent patterns $L_{1,0}$ as shown in Fig. 2.

The kth phase can be further divided into k sub-phases. The first sub-phase of the kth phase is to produce $L_{k,k-1}$. To this end, we derive $C_{k,k-1}$ by $L_{k-1,k-2}$ join $L_{k-1,k-2}$. For example, $C_{2,1}$ in Fig. 3 was derived by $L_{1,0}$ join $L_{1,0}$ and $C_{3,2}$ in Fig. 4 by $L_{2,1}$ join $L_{2,1}$. (For the details of join operation, please see the function *GetJoin*, which will be introduced later.) Having obtained $C_{k,k-1}$, we then derive $L_{k,k-1}$ by computing their supports, which can be done by scanning the database and trimming patterns with insufficient supports.

The jth sub-phase of the kth phase, where $2 \leqslant j \leqslant k$, contains the following steps. In this sub-phase, we need to produce $L_{k,k-j}$ from $C_{k,k-j}$. To get $C_{k,k-j}$, we can produce it from $L_{k,k-j+1}$ by removing an internal variable character "*". For example, by removing an internal variable character "*" from $\langle A^*C^*A \rangle$ in $L_{3,2}$, we derive candidate patterns $\langle AC^*A \rangle$ and $\langle A^*CA \rangle$ in $C_{3,1}$. Having obtained $C_{k,k-j}$, we then derive $L_{k,k-j}$ by computing their supports and pruning

$C_{1,0}$	
Candidate	support
A	6
B	4
C	4
D	1
E	1
F	1
G	7
H	1
I	1
J	1
K	1

→

$L_{1,0}$	
Frequent	Support
A	6
B	4
C	4
G	7

Fig. 2. The first phase.

those patterns without enough supports. The above steps can be summarized as the following algorithm:

Algorithm GFP1
1 Generate $C_{1,0}$
2 Determine $L_{1,0}$ by scanning the database
3 For ($k = 2; L_{k-1,k-2} \neq \emptyset; k++$) do
4 $C_{k,k-1} = GetJoin(L_{k-1,k-2}, L_{k-1,k-2})$
5 Prune $C_{k,k-1}$
6 Determine $L_{k,k-1}$ by scanning the database
7 For ($j = 2; j = k; j++$) do
8 $C_{k,k-j} = GetNextCandidate(L_{k,k-j+1})$
9 Prune $C_{k,k-j}$
10 Determine $L_{k,k-j}$ by scanning the database

In the above algorithm, some steps need to be further explained. These steps include the function *GetJoin* in step 4, GetNextCandidate in step 8 and Pruning method in steps 5 and 9. In the following, we explain each of them in order.

2.1. Function GetJoin

In algorithm GFP1, function *GetJoin* is used to generate $C_{k,\,k-1}$ from the frequent pattern set $L_{k-1,\,k-2}$. To see how this is done, let us consider a frequent pattern X in $L_{k,\,k-1}$. Since X is in $L_{k,\,k-1}$, it can be represented as follows:

$X = \langle x_1 * x_2 * x_3 * \cdots * x_{k-2} * x_{k-1} * x_k \rangle.$

In everywhere X has occurred, so did the following two patterns:

$X(1, k-1) = \langle x_1 * x_2 * x_3 * \cdots * x_{k-2} * x_{k-1} \rangle,$

$X(2, k) = \langle x_2 * x_3 * \cdots * x_{k-2} * x_{k-1} * x_k \rangle,$

where $X(i,j)$ denotes the pattern formed by the subsequence of X from the ith known item to the jth known item. Therefore, the necessary condition of X in $L_{k,\,k-1}$ is that $X(1, k-1)$ and $X(2, k)$ must be also frequent pattern, i.e., belong to the set $L_{k-1,\,k-2}$. In a word, by joining the patterns in $L_{k-1,\,k-2}$, we can find a set of candidate patterns which contains all frequent patterns in $L_{k,\,k-1}$. The following is the formal definition of function *GetJoin*:

Function *GetJoin*($L_{k-1,\,k-2}, L_{k-1,\,k-2}$)
insert into $C_{k,\,k-1}$
select $p/, q(k-1, k-1)$
from $L_{k-1,\,k-2}$ p, $L_{k-1,\,k-2}$ q
where $p(2, k-1) = q(1, k-2)$

Example 3. Suppose we want to generate $C_{4,3}$ in Fig. 6 from $L_{3,2}$ in Fig. 4. When $p = \langle A^*C^*A \rangle$ and $q = \langle C^*A^*G \rangle$, we have $p(2,3) = q(1,2) = \langle C^*A \rangle$; hence, $p/ = \langle A^*C^*A^* \rangle$ and $q(3,3) = \langle G \rangle$ and the generated pattern is $\langle A^*C^*A^*G \rangle$.

2.2. Function GetNextCandidates

In Algorithm GFP1, function *GetNextCandidate* is used to generate $C_{k,\,k-j}$ from the frequent pattern set $L_{k,\,k-j+1}$. To see how this is done, let X denote a frequent pattern in $L_{k,\,k-j}$. Then X will have k–j internal "*", k known items and k-1 interleaving positions between known elements. Since $k-j < k-1$, there must exist two adjacent known elements x_i and x_{i+1} without interleaving "*". Hence we can represent X as follows:

$X = \langle x_1 \ldots x_i x_{i+1} \ldots \ldots x_k \rangle.$

If we insert a variable character "*" into the position between x_i and x_{i+1}, then we get a new pattern as follows:

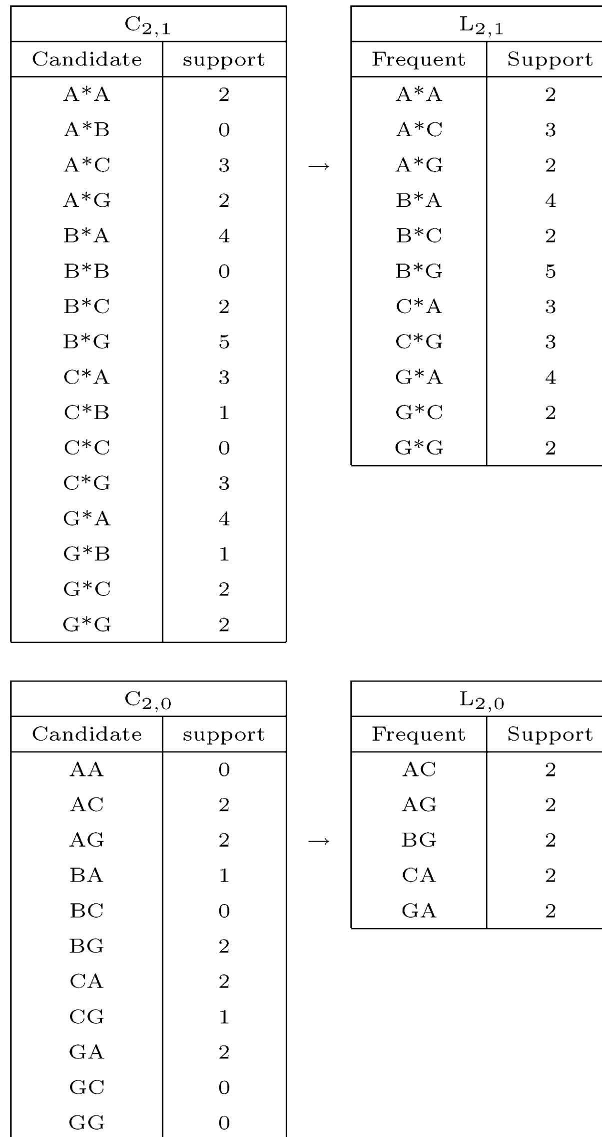
$X' = \langle x_1 \ldots x_i * x_{i+1} \ldots \ldots * x_k \rangle.$

Obviously, wherever contain X also contain X'. In other words, the support of X' will be not less than that of X. Hence, the necessary condition of X in $L_{k,\,k-j}$ is that X' must be in $L_{k,\,k-j+1}$. From this observation, we can find the candidate set $C_{k,\,k-j}$ by removing a "*" from all the frequent patterns in $L_{k,\,k-j+1}$. The following is the formal definition of function *GetNextCandidates*:

Function *GetNextCandidates*($L_{k,\,k-j+1}$)
insert into $C_{k,\,k-j}$
select $\langle x_1 \ldots x_i x_{i+1} \ldots \ldots x_k \rangle$
from $L_{k,\,k-j+1} X' = \langle x_1 \ldots x_i * x_{i+1} \ldots \ldots x_k \rangle$
where "*" exists between x_i and x_{i+1}

2.3. Prune $C_{k,\,k-j}$

In fact, the size of $C_{k,\,k-j}$ can be further reduced. Let us consider pattern $\langle A^*BCD \rangle$. Obviously, the support of this pattern must be no more



$C_{2,1}$	
Candidate	support
A*A	2
A*B	0
A*C	3
A*G	2
B*A	4
B*B	0
B*C	2
B*G	5
C*A	3
C*B	1
C*C	0
C*G	3
G*A	4
G*B	1
G*C	2
G*G	2

→

$L_{2,1}$	
Frequent	Support
A*A	2
A*C	3
A*G	2
B*A	4
B*C	2
B*G	5
C*A	3
C*G	3
G*A	4
G*C	2
G*G	2

$C_{2,0}$	
Candidate	support
AA	0
AC	2
AG	2
BA	1
BC	0
BG	2
CA	2
CG	1
GA	2
GC	0
GG	0

→

$L_{2,0}$	
Frequent	Support
AC	2
AG	2
BG	2
CA	2
GA	2

Fig. 3. The second phase.

than its sub-patterns such as $\langle A^*BC \rangle$, $\langle A^*B^*D \rangle$, $\langle A^*CD \rangle$ and $\langle BCD \rangle$. Hence the necessary condition of $X \in C_{k,k-j}$ to be frequent is that its (k-1)-subpattern Y must also be frequent. According to this observation, the set $C_{k,k-j}$ can be further pruned by the following procedure:

For patterns $X \in C_{k,k-j}$ do
if any (k-1)-subpattern Y of X is not frequent
then delete X from $C_{k,k-j}$

To find (k-1)-subpattern Y of X, we need to choose one known item in X and replace it with "*". After the replacement, if the replaced "*" has some adjacent "*"s, then we further combine them as a single "*". For example, if $X = \langle A^*BCD \rangle$, then by replacing A we get $Y = \langle BCD \rangle$, by B we get $Y = \langle A^*CD \rangle$, by C we get $Y = \langle A^*B^*D \rangle$ and D we get $Y = \langle A^*BC \rangle$. Since the replaced "*" may have zero, one or two adjacent "*"s, the (k-1)-subpattern Y of X may have $k-j+1$ or

$C_{3,2}$	
Candidate	support
A*A*A	0
A*A*C	0
A*A*G	1
A*C*A	2
A*C*G	1
A*G*A	0
A*G*C	0
A*G*G	0
B*A*A	1
B*A*C	2
B*A*G	1
B*C*A	1
B*C*G	0
B*G*A	3
B*G*C	2
B*G*G	1
C*A*A	0
C*A*C	0
C*A*G	2
C*G*A	1
C*G*C	0
C*G*G	1
G*A*A	1
G*A*C	2
G*A*G	1
G*C*A	1
G*C*G	0
G*G*A	0
G*G*C	0
G*G*G	0

→

$L_{3,2}$	
Frequent	Support
A*C*A	2
B*A*C	2
B*G*A	3
B*G*C	2
C*A*G	2
G*A*C	2

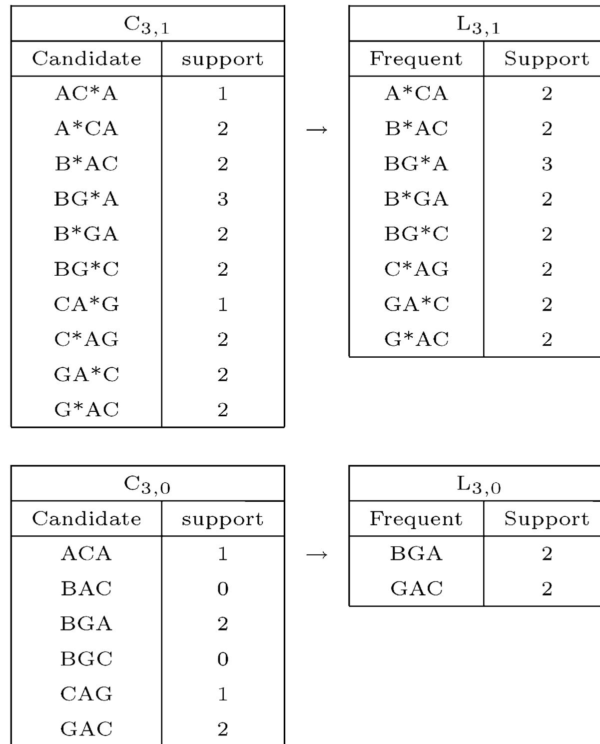
Fig. 4. The first sub-phase in the third phase.

$k-j$ or $k-j-1$ internal "*"s. This means that we can determine if Y is frequent or not by examining $L_{k-1,k-j+1}$, $L_{k-1,k-j}$ and $L_{k-1,k-j-1}$. Finally, we use an example to illustrate the whole algorithm.

Example 4. Fig. 1 shows the database, where we have five transactions and 11 items. Suppose we have *minsup* $= 2$. In the first phase, we first generate $C_{1,0}$, and the result is shown in Fig. 2. Then, we scan the database to determine the support of each candidate pattern in $C_{1,0}$, and we get $L_{1,0}$ by removing those candidate patterns without enough supports. The results are shown in Fig. 2.

In the second phase, the following two sub-phases are done:

(1) Derive $C_{2,1}$ by *GetJoin*($L_{1,0}, L_{1,0}$). Prune $C_{2,1}$ by checking if its $\langle 1 \rangle$-subpatterns are



$C_{3,1}$	
Candidate	support
AC*A	1
A*CA	2
B*AC	2
BG*A	3
B*GA	2
BG*C	2
CA*G	1
C*AG	2
GA*C	2
G*AC	2

→

$L_{3,1}$	
Frequent	Support
A*CA	2
B*AC	2
BG*A	3
B*GA	2
BG*C	2
C*AG	2
GA*C	2
G*AC	2

$C_{3,0}$	
Candidate	support
ACA	1
BAC	0
BGA	2
BGC	0
CAG	1
GAC	2

→

$L_{3,0}$	
Frequent	Support
BGA	2
GAC	2

Fig. 5. The second and third sub-phases in the third phase.

frequent. Compute their supports and determine $L_{2,1}$.

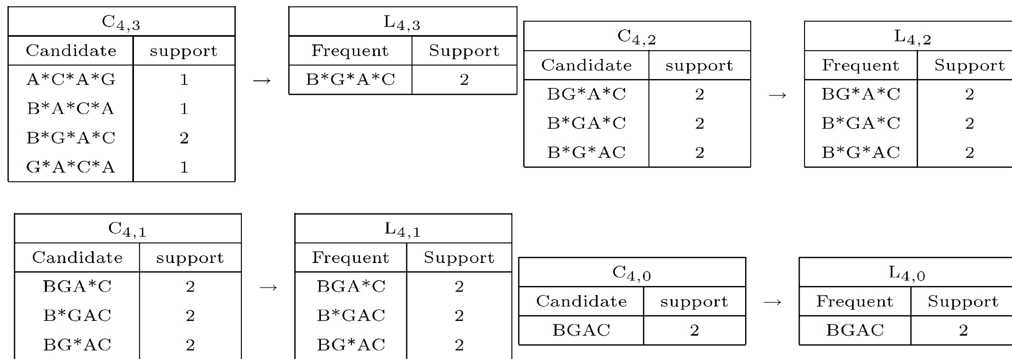
(2) Generate $C_{2,0}$ by function *GetNextCandidates*($L_{2,1}$). Prune $C_{2,0}$ by checking if its $\langle 1 \rangle$-subpatterns are frequent. Compute their supports and determine $L_{2,0}$.

The other phases can be done similarly, and its results are shown in Figs. 3–6.

3. An improved algorithm

Although the algorithm in Section 2 is correct, it is possible to further improve its performance. First, note that the function *number*$(X, 1, T, 1)$ is used to determine the number of occurrences of $X \subset T$. Thus, if there are n different patterns X in the candidate pattern set, say $C_{k,r}$, then each transaction needs to execute the function n times. This redundancy motivates us to design a tree structure representing all the patterns in $C_{k,r}$ so that we can examine all the patterns at a time. Next, we also notice that the algorithm requires k sub-phases in the kth phase in order to find the frequent patterns from the candidate sets $C_{k,k-1}, C_{k,k-2}, \ldots, C_{k,0}$. For each sub-phase, we need to scan the database one time. Therefore, if the maximum fixed length of the candidates is m, then we need to scan the database $1 + 2 + 3 + \cdots + m = m(m+1)/2$ times. To reduce this heavy I/O cost, our approach is to integrate all the sub-phases in the same phase into a single phase. This way, the number of times of database scans can be reduced from $m(m+1)/2$ to m.

In the following, we describe how we finish the above two improvements. First, each phase of the algorithm, say the kth phase, needs to construct a candidate tree to represent all the patterns appeared in the candidate sets $C_{k,k-1}, C_{k,k-2}, \ldots, C_{k,0}$. Initially, the candidate tree is only a root without any other nodes. For each



$C_{4,3}$	
Candidate	support
A*C*A*G	1
B*A*C*A	1
B*G*A*C	2
G*A*C*A	1

→

$L_{4,3}$	
Frequent	Support
B*G*A*C	2

$C_{4,2}$	
Candidate	support
BG*A*C	2
B*GA*C	2
B*G*AC	2

→

$L_{4,2}$	
Frequent	Support
BG*A*C	2
B*GA*C	2
B*G*AC	2

$C_{4,1}$	
Candidate	support
BGA*C	2
B*GAC	2
BG*AC	2

→

$L_{4,1}$	
Frequent	Support
BGA*C	2
B*GAC	2
BG*AC	2

$C_{4,0}$	
Candidate	support
BGAC	2

→

$L_{4,0}$	
Frequent	Support
BGAC	2

Fig. 6. The fourth phase.

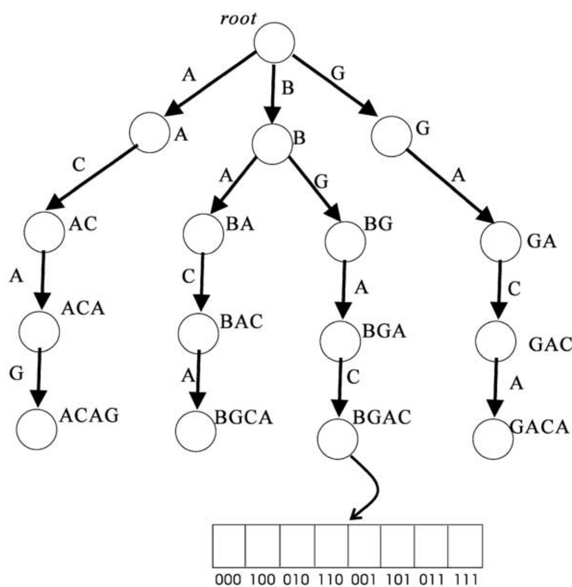


Fig. 7. The candidate tree constructed from $C_{4,3}$.

candidate pattern in $C_{k,k-1}$, we insert it into the candidate tree. Note that, after all patterns have been inserted, all the leaves have the same depth and the path from the root down to the leaf node corresponds to a pattern. For example, if we construct the candidate tree for $C_{4,3}$ in Fig. 6, then the result is shown as in Fig. 7.

Here, the readers should note this tree is suitable for representing not only $C_{4,3}$ but also the supersets of $C_{4,2}$, $C_{4,1}$ and $C_{4,0}$. For example, leaf node BGAC can be viewed in eight different ways as BGAC, B*GAC, BG*AC, B*G*AC, BGA*C, B*GA*C, BG*A*C, B*G*A*C. To make the difference clear, we attach eight fields with each leaf node in Fig. 7, where each field is used to store the support of a different pattern.

Generally speaking, for a candidate tree constructed from $C_{k,k-1}$, each leaf node, say $x_1x_2x_3\ldots x_k$, will have 2^{k-1} fields to keep the supports for 2^{k-1} different candidate patterns. Let these fields be numbered as $0, 1, 2, \ldots, 2^{k-1}-1$. To arrange the storing positions for these different patterns, we use the following rule. Set $b_i = 1$ if the considered pattern has a "*" between x_i and x_{i+1}. Otherwise, $b_i = 0$. Then the field with index s is used to store the pattern where $\sum_{i=1}^{k-1} b_i 2^{i-1} = s$. According to this rule, the leaf node BGAC will store the eight different patterns in this sequence BGAC, B*GAC, BG*AC, B*G*AC, BGA*C, B*GA*C, BG*A*C and B*G*A*C.

In our new algorithm, all the transactions will traverse the candidate tree one by one. In such a traversal, we will remember the positions in the transaction that match the arcs in the path. For example, assume that we traverse the tree by transaction 5 = (BGACAFJ). After we reaching the leaf node, say leaf node BGAC, the position matched with the first arc is 1, matched with the second is 2, with the third is 3 and the last is 4. From these matched positions, we can infer the possible patterns. If two adjacent positions are continuous, then we may have either one "*" interleaved between them or none. On the other hand, if they are not continuous, then there must

have a "*" between them. For example, if the matched positions are 1, 2, 3, 4, then all eight patterns occur in the transaction, and the indexes b_1, b_2 and b_3 can all be set as either 1 or 0. As another example, if the transaction data is (BFGKACD), then the matched positions are 1, 3, 5 and 6. In this case, B*G*AC and B*G*A*C are possible patterns; so, b_1 and b_2 can only be 1 but b_3 can be either 1 or 0.

In the process of traversing the candidate tree, if we have reached a leaf node, we need to find all the matched patterns and increase the supports of all the corresponding fields. This means that the computation time for each reached leaf of depth k is $O(2^{k-1})$. Since the reaching of a leaf node may occur many times in scanning the database, this cost is too high to tolerate. Here, we provide a way to reduce the computation time for each reached leaf from $O(2^{k-1})$ to $O(1)$. In the following, we explain our idea:

For a pattern X, we say Y is a generalization of X, if X and Y are the same patterns except that Y has more "*"s than X. For example, BG*A*C is a generalization of BGA*C and BGAC. In Fig. 8, we show the possible generalizations for every pattern in $C_{k,k-1}$. For example, the pattern BGA*C (indexed as 001) has generalized patterns as B*GA*C (101), BG*A*C (011) and B*G*A*C (111). Similarly, the pattern BG*AC (indexed as 010) has generalized patterns as BG*A*C (011), B*G*AC (110) and B*G*A*C (111). In traversing the candidate tree in Fig. 7, if we get a series of matched positions, say 1, 3, 5 and 6, then the possible indexes are "110" and "111" and the found patterns are B*G*AC and B*G*A*C. In these two patterns, B*G*AC is more specific than B*G*A*C, and let us call it the most specific pattern. As a similar example, if the transaction is (ABKFKGAC), then the series of matched positions is 2,6,7,8, and so the found patterns are B*GAC, B*G*AC, B*GA*C and B*G*A*C, where B*GAC is the most specific pattern. Originally, when we have reached a leaf in the candidate tree, we need to increase the supports of all found patterns. That is why we need $O(2^{k-1})$ computation for every reaching. To reduce this cost, rather than all found patterns, we only increase the support of the most specific pattern. Say, if the series of matched positions is 2, 6, 7, 8, then we only increase the field of index "100". Similarly, if the series of positions is 1, 3, 5, 6, then we add 1 into the field with index "110". This way, the computation cost for each reaching is reduced to $O(1)$, because we need only one addition.

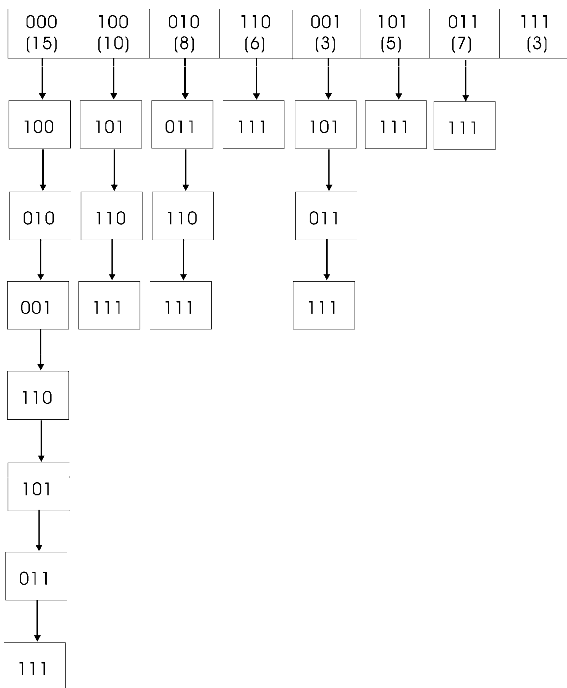


Fig. 8. The compensation list structure from $C_{4,3}$.

Unfortunately, a problem arises immediately is how we can ensure that the supports of all the patterns are correct, because we omit many additions that we should do. To compensate those missing computations, we choose to redo it again after all transactions have finished the traversals of the candidate tree. Let us consider Fig. 8 again, where the numbers inside the brackets denote the supports without compensation. Since general patterns occur in those places where specific patterns occur, we can compensate the missing counts by adding back the supports to the generalized patterns. Therefore, the support of pattern BGA*C (indexed as 001) should be added into those of B*GA*C (101), BG*A*C (011) and B*G*A*C (111). Similarly, the support of BG*AC (indexed as 010) need be added into those of BG*A*C (011), B*G*AC (110) and B*G*A*C (111).

In Fig. 9,we show the supports of all the patterns after compensation.

Note that the compensation list structure like Fig. 8 can be applied to all the patterns of fixed length 4, because they all have the same generalization structure. From this list, we know in what order the compensation can be executed and into which fields we should add the supports. The advantage of using this list is that we can save many comparison operations. Without the list, each time when we process a candidate in $C_{k,k-1}$, comparisons are required to determine the next field to execute the compensation and into which fields our supports should be added. If there have totally n candidates in $C_{k,k-1}$, then all these comparisons are repeated n times. Since all candidates in $C_{k,k-1}$ have the same generalization structure, it is possible to spare all these comparisons if we store the comparison procedure as a list structure. By using the list, we are free from the comparisons, and the additions can be done directly by following the list.

000	100	010	110	001	101	011	111
15	25	23	39	18	33	33	57

Fig. 9. The supports of all the patterns after compensation.

In the above paragraphs, we do not yet give the details of how a transaction traverses the candidate tree. The following is the step-by-step procedure, where we can start the traversal by calling *traverse*$(CT, root, T, 1, \emptyset)$.

```
Function traverse(CT, u, T, j, positions)
/*CT denotes the constructed candidate tree */
/*u is the node in CT that we are going to match with T(j)*/
/*positions stores the list of matched positions from root to node u */
/* item(u, v) denote the item associated with arc (u, v) */
If u is a leaf node
    Then add 1 into the field of the most specific pattern and return
Else if u is not a leaf and T(j) is the end of transaction
    Then return
Else if u is not a leaf and T(j) is the not the end of transaction
    Then traverse(CT, u, T, j + 1, positions)
        For each child v of u
            If item(u, v) = T(j)
                then append j to the end of positions
                    traverse(CT, v, T, j + 1, positions)
Return
```

Example 5. Let us use the candidate tree of Fig. 7 and transaction $T = $ (BGACAC) as an example to illustrate the execution of the above function. We activate the computation by calling *traverse*$(CT, root, T, 1, \emptyset)$. In that function, we will call another two functions *traverse*$(CT, root, T, 2, \emptyset)$, and *traverse*$(CT$, node B, $T, 2, (1))$. In processing *traverse*$(CT, root, T, 2, \emptyset)$, another two functions *traverse*$(CT, root, T, 3, \emptyset)$, *traverse*$(CT$, node G, $T, 3, (2))$ will be called. Similarly, the execution of function *traverse*$(CT$, node B, $T, 2, (1))$ will initiate another two functions *traverse*$(CT$, node B,$T, 3, (1))$ and *traverse*$(CT$, node BG,$T, 3, (1, 2))$. Repeatedly doing this way, we will find the most specific patterns in the following functions:

traverse(CT, node BGAC, T, 5, (1,2,3,4)) and the pattern BGAC
traverse(CT, node GACA, T, 6, (2,3,4,5)) and the pattern GACA
traverse(CT, node BGAC, T, 7, (1,2,3,6)) and the pattern BGA*C
traverse(CT, node BGAC, T, 7, (1,2,5,6)) and the patter BG*AC

Finally, by integrating all the techniques proposed in this section into Algorithm GFP1, the final algorithm will become as follows:

Algorithm GFP2
1 Generate $C_{1,0}$
2 Determine $L_{1,0}$ by scanning the database
3 For ($k = 2; L_{k-1,k-2} \neq \emptyset; k++$) do
4 $C_{k,k-1} = GetJoin(L_{k-1,k-2}, L_{k-1,k-2})$
5 Prune $C_{k,k-1}$
6 Construct the candidate tree from $C_{k,k-1}$
7 For every transaction
8 traverse the candidate tree
9 add the supports only to the fields of the most specific patterns
10 Endfor
11 Build the compensation-working list
12 For each leaf node in the candidate tree
13 Compensate the supports of the fields by following the working list
14 Endfor
15 Determine $L_{k,k-1}$ by removing those patterns with insufficient supports

4. Sequential rules

From the found hybrid patterns, the sequential rules can be generated. Our rules are different from the previous researches' in two ways.

(1) Besides the forward rules that reason forwardly such as $X \to Y$, we also have the backward rules such as $X \leftarrow Y$.
(2) We divide the rules into two categories where the first is direct rule and the second indirect.

Totally, we have the following four kinds of rules:

(1) Direct-and-forward: A direct-and-forward rule of the form $X \to Y$ holds if (a) the support of pattern $Z = X + Y$ is no less than *minsup*; and (b) the confidence of the rule is no less than the user-specified minimum confidence. Here, the confidence of the rule is defined as follows: *confidence*$(X \to Y) =$ *support*$(Z)/$*support*(X).

For example, if our rule is $\langle$AB$\rangle \to \langle$CD*G$\rangle$, then we have *confidence*$(X \to Y) =$ *support*$(\langle$ABCD*G$\rangle)/$ *support*$(\langle$AB$\rangle)$.

(2) Direct-and-backward: A direct-and-backward rule of the form $X \leftarrow Y$ holds if (a) the support of pattern $Z = X + Y$ is no less than *minsup*; and (b) the confidence of the rule is no less than the user-specified minimum confidence. Here, the confidence of the rule is defined as follows:

confidence$(X \leftarrow Y) =$ *support*$(Z)/$*support*(Y). For example, if our rule is $\langle$AB$\rangle \leftarrow \langle$CD*G$\rangle$, then we have *confidence*$(X \leftarrow Y) =$ *support*$(\langle$ABCD*G$\rangle)/$ *support*$(\langle$CD*G$\rangle)$.

(3) Indirect-and-forward: An indirect-and-forward rule of the form $X \to^* Y$ holds if (a) the support of pattern $Z = X + /Y$ is no less than *minsup*; and (b) the confidence of the rule is no less than the user-specified minimum confidence. Here, the confidence of the rule is defined as follows:

confidence$(X \to^* Y) =$ *support*$(Z)/$*support*(X). For example, if our rule is $\langle$AB$\rangle \to^* \langle$CD*G$\rangle$ then we have *confidence*$(X \to^* Y) =$ *support*$(\langle$AB*CD*G$\rangle)/$ *support*$(\langle$AB$\rangle)$.

(4) Indirect-and-backward: An indirect-and-backward rule of the form $X \leftarrow^* Y$ holds if

(a) the support of pattern $Z = X/+Y$ is no less than *minsup*; and (b) the confidence of the rule is no less than the user-specified minimum confidence. Here, the confidence of the rule is defined as follows:

$confidence(X \leftarrow^{*} Y) = support(Z)/support(X)$.

For example, if our rule is $\langle AB \rangle \leftarrow^{*} \langle CD^{*}G \rangle$ then we have

$confidence(X \leftarrow^{*} Y) = support(\langle AB^{*}CD^{*}G \rangle)/support(\langle AB \rangle)$.

5. Experimental results

The experiment contains three parts. Firstly, we compare the performance between the algorithms proposed in Sections 2 and 3, GFP1 and GFP2, and we find that GFP2 is much faster than GFP1. Secondly, we compare GFP2 with the algorithm for mining continuous patterns. Since there were no previous researches dedicated for studying its performance, we adopt the most popular method, a modified Apriori algorithm, as the comparison target. (Although these past researches used continuous patterns, their goals are not to develop efficient methods for finding them [15,16,9,17,23,14].) The simulation result shows that GFP2 greatly outperforms the modified Apriori algorithm. Thirdly, we compare GFP2 with the algorithms for mining discontinuous patterns. The targets of our comparison include PrefixSpan algorithm [21] and WAP-tree algorithm [12], because PrefixSpan and WAP-tree are independently declared as the fastest algorithms for discontinuous patterns. The simulation result shows that GFP2 is faster than the PrefixSpan algorithm, but is only as fast as the WAP-tree algorithm. Although our algorithm did not surpass the WAP-tree algorithm, the patterns we can find are much more abundant than those of the WAP-tree algorithm. Not only can we find discontinuous patterns, as the WAP-tree algorithm does, but also we can find continuous patterns as well as some interesting patterns that are neither discontinuous nor continuous.

The first part of simulation is done on a PC with Pentium-II 266 processor and 384M main memory under the NT4.0 operating system and use experiment data from the Access log file in the library Web server. A record in the log file is composed of an IP address and a web page. In the log file, the IP sequentially visits many web pages and we combine these continuous records with the same IP into a single record. This way, a record is a transaction with many items (web pages). We use the real-life dataset and vary different parameters to analyze the algorithms' performances and the numbers of generated patterns.

Tables 1 and 2 compare the running times of the two algorithms. Table 1 shows the execution times of GFP1 and GFP2 when the minimum support is fixed as five transactions but the number of transactions is varied from 500 transactions to 4000 transactions. Table 2 shows the execution

Table 1
Execution times vs. numbers of transactions for the two algorithms

Number of transactions (s)	500	1000	1500	2000	2500	3000	3500	4000
Execution time of GFP1	829	850	947	1017	2372	2340	14375	33328
Execution time of GFP2	1	1	1	1	2	2	2	3

Table 2
Execution times vs. supports for the two algorithms

Minimum support transactions (s)	15	12	9	6	3
Execution time of GFP1	912	923	938	981	5118
Execution time of GFP2	1	1	1	1	3

Table 3
Parameter settings

$\lvert D\rvert$	Number of transactons
$\lvert T\rvert$	Average size of the transactions
$\lvert I\rvert$	Average size of the maximal potentially frequent continuous patterns
$\lvert L\rvert$	Number of maximal potentially frequent patterns
N	Number of items

times of GFP1 and GFP2 as the minimum support is decreased from 15 to 3 and the number of transactions is set as 2000. As expected, GFP2 outperforms GFP1 in all cases.

Secondly, we compare GFP2 with the modified Apriori algorithm, which is for finding continuous patterns. Since the real-life dataset on our hand is not large enough, we generate the synthetic datasets by applying the algorithm in [4] and all the experiments are performed a PC with Pentium-III 933 processor and 1024M main memory under the Window 2000 operating system. However, the content of our transactions is little different from theirs in that we allow a transaction to have repeated items. Similar as that in [4], the parameters need to be set are shown in Table 3, where we set $N = 10000$ and $\lvert L\rvert = 5000$. In this experiments, three datasets are used for comparisons: T25I2D10K, T25I2D30K and T25I2D50K. The results in Table 4 show that GFP2 greatly outperforms the modified Apriori algorithm. In Table 4, we also note that the number of generated hybrid patterns is obviously greater than that of continuous patterns. In fact, the numbers of generated patterns could have an even bigger contrast, if the minimum support is changed to a smaller value. However, we did not do this for saving the endless time to run the modified Apriori algorithm.

Finally, we make a comparison between GFP2, PrefixSpan and WAP-tree algorithms. Table 5 summarizes the dataset parameter settings. Figs. 10 and 11 show the execution times of 3 algorithms for different sizes of database, where $\lvert I\rvert$ is set as either 2 or 4 and the minimum support is set as 0.01. Similarly, Figs. 14 and 15 show the execution times for different minimum supports, where $\lvert I\rvert$ is set as either 2 or 4 and the number of transactions is set as 500K. From the result, we see

Table 4
The comparison between GFP2 and the modified Apriori algorithm

Minimum support (0.015)	T25.I2.D10K	T25.I2.D30K	T25.I2.D50K
Apriori-like (sec)	3022	8930	16544
Number of frequent sequences	193	182	212
GFP2 (sec)	8	17	33
Number of frequent sequences	329	204	476

Table 5
Parameter settings

Name	$\lvert T\rvert$	$\lvert I\rvert$	$\lvert D\rvert$
T10.I2.D100K	10	2	100K
T10.I2.D200K	10	2	200K
T10.I2.D300K	10	2	300K
T10.I2.D400K	10	2	400K
T10.I2.D500K	10	2	500K
T10.I4.D100K	10	4	100K
T10.I4.D200K	10	4	200K
T10.I4.D300K	10	4	300K
T10.I4.D400K	10	4	400K
T10.I4.D500K	10	4	500K

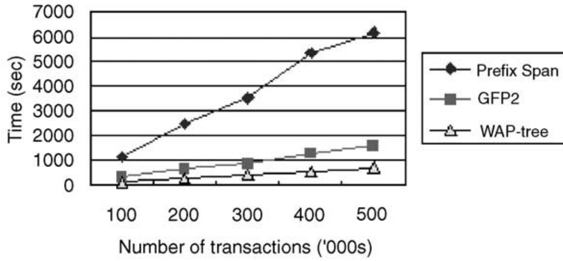


Fig. 10. Execution times vs. numbers of transactions for I2.

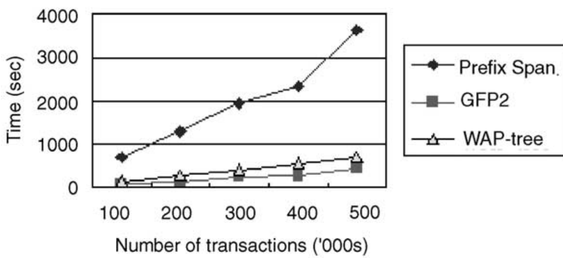


Fig. 11. Execution times vs. numbers of transactions for I4.

that the running time of 3 algorithms grow linearly when the number of transactions is increased, and the performances of GFP2 and WAP-tree are better than that of PrefixSpan. Among GFP2 and WAP-tree, GFP2 is the winner when the transaction size is bigger ($|I| = 4$). The possible reason for this phenomenon may be due to WAP-tree algorithm requires a lot of recursions to recursively produce other sub-WAP-trees. Therefore, when the recursion depth is deeper, its performance deteriorates more quickly than GFP2.

Besides, Figs. 12 and 13 show the numbers of generated hybrid patterns and discontinuous patterns for different database sizes. From the figures, we see that the number of patterns is not related with the database size. Further, Figs. 16 and 17 show that the number of generated hybrid patterns is only slightly larger than that of discontinuous patterns when the minimum support is large. The reason for this phenomenon is because when the minimum support threshold is high we only generate those patterns that are short, i.e., containing only one item. But for a smaller minimum support threshold, the result

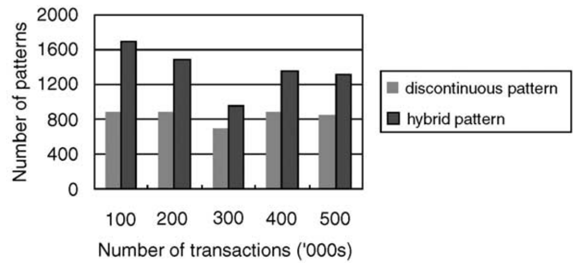


Fig. 12. Numbers of frequent patterns vs. numbers of transactions for Fig. 10.

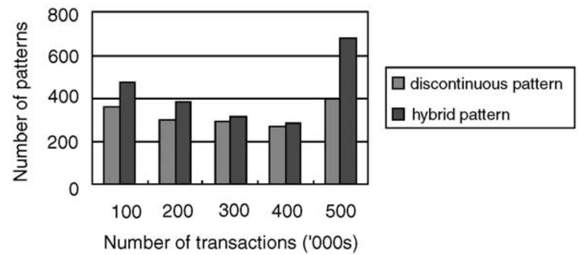


Fig. 13. Numbers of frequent patterns vs. numbers of transactions for Fig. 11.

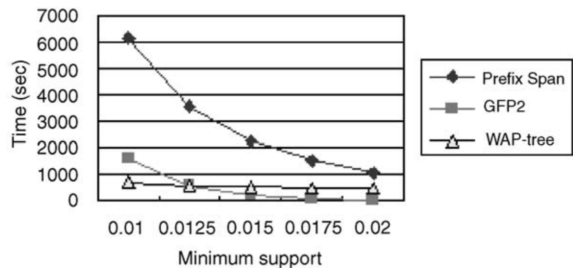


Fig. 14. Execution times vs. minimum supports for I2.

shows that the number of hybrid patterns is much more than that of discontinuous patterns.

6. Conclusion

The problem studied in this paper is to find hybrid patterns from sequential data. In this paper, two algorithms are developed, where the

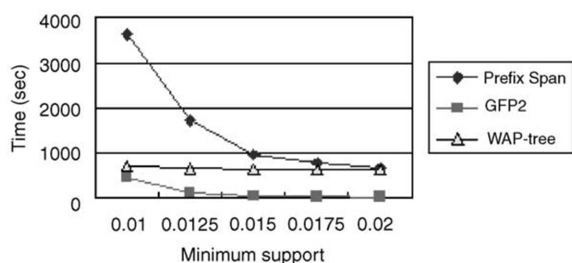


Fig. 15. Execution times vs. minimum supports for I4.

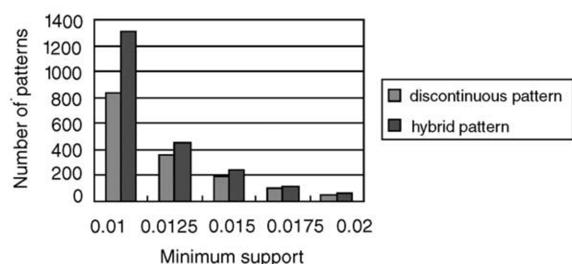


Fig. 16. Numbers of frequent patterns vs. minimum supports for I2.

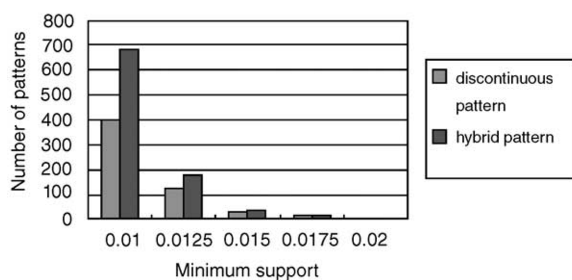


Fig. 17. Numbers of frequent patterns vs. minimum supports for I4.

first algorithm is easy but slow while the second complicated but much faster. From the experiments, it is shown that the second algorithm GFP2 is as fast as the currently best algorithm for mining sequential patterns. Although our algorithm only runs the same speed as the WAP-tree algorithm, the patterns we can find are much more abundant than those of the WAP-tree algorithm. Not only can we find discontinuous patterns, as the WAP-tree algorithm does, but also we can find continuous patterns as well as some interesting patterns that are neither discontinuous nor continuous.

This paper has some possible extensions. For example, note that generalization of patterns is very important for practical purposes. Without generalization, the generated patterns may be too detailed. Therefore, by including concept hierarchies, we may produce patterns or rules that are more abstract and meaningful. Next, another possible extension is to prune less interesting patterns that are trivial or implied by other patterns. Without removing uninteresting patterns, we may be overwhelmed by a great number of patterns. Further, we may ask for that the generated patterns must comply with some given meta-forms. This will help us to avoid lots of patterns that do not fit our expectations or applications. In addition, we can associate a quantity with each item in the sequence. This quantity may denote the staying time in a web site. And the problem now becomes to find quantitative patterns from sequential data. Finally, we can extend the transaction structure such that a transaction is formed of a list of itemsets rather than a list of items.

References

[1] M.-S. Chen, J. Han, P.S. Yu, Data mining: an overview from a database perspective, IEEE Trans. Knowledge Data Eng. 8 (6) (1996) 866–883.

[2] R. Agrawal, T. Imielinski, A. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, ACM Press, New York, 1993, pp. 207–216.

[3] R. Agrawal, J.C. Shafer, Parallel mining of association rules: design, implementation, and experience, IEEE Trans. Knowledge Data Eng. 8 (6) (1996) 962–969.

[4] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, Morgan Kaufmann, Los Altos, CA, 1994, pp. 478–499.

[5] J. Han, Y. Fu, Mining multiple-level association rules in large databases, IEEE Trans. Knowledge Data Eng. 11 (5) (1999) 798–805.

[6] J.S. Park, M.-S. Chen, P.S. Yu, An effective hash based algorithm for mining association rules, in: Proceedings of

the ACM SIGMOD International Conference on Management of Data, San Jose, California, ACM Press, New York, 1995, pp. 175–186.
[7] R. Srikant, R. Agrawal. Mining generalized association rules, in: Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Morgan Kaufmann, Los Altos, CA, 1995, pp. 407–419.
[8] R. Agrawal, R. Srikant, Mining sequential patterns, Proceedings of the 7th International Conference on Data Engineering, Taipei, Taiwan, IEEE Computer Society, 1995, pp. 3–14.
[9] M.-S. Chen, J.S. Park, P.S. Yu, Efficient data mining for path traversal patterns, IEEE Trans. Knowledge Data Eng. 10 (2) (1998) 209–221.
[10] R. Cooley, B. Mobasher, J. Srivastava, Data preparation for mining world wide web browsing patterns, Knowledge and Information Systems 1 (1) (1999) 5–32.
[11] B. Mobasher, N. Jain, E. Han, J. Srivastava, Web mining: pattern discovery from world wide web transactions, Technical Report TR96-050, Department of Computer Science, University of Minnesota, 1996.
[12] J. Pei, J. Han, B. Mortazavi-Asl, H. Zhu, Mining access pattern efficiently from web logs, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Kyoto, Japan, Springer, Berlin, 2000, pp. 396–407.
[13] M.J. Zaki, N. Lesh, M. Ogihara, PlanMine: sequence mining for plan failures, in: Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining, New York City, New York, AIAA Press, New York, 1998, pp. 369–374.
[14] J.T. Wang, G.W. Chirn, T.G. Marr, B. Shapiro, D. Shasha, K. Zhang, Combinatorial pattern discovery for scientific data: some preliminary results, in: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, ACM Press, New York, 1994, pp. 115–125.
[15] R. Agrawal, C. Faloutsos, A. Swami, Efficient similarity search in sequence databases, in: Conference on Foundations of Data Organization and Algorithms, Chicago, Illinois, Springer, Berlin, 1993, pp. 69–84.
[16] R. Agrawal, K. Lin, H.S. Sawhney, K. Shim, Fast similarity search in the presence of noise, scaling, and translation in time-series databases, in: Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, Morgan Kaufmann, Los Altos, CA, 1995, pp. 490–501.
[17] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases, in: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, ACM Press, New York, 1994, pp. 419–429.
[18] J. Han, G. Dong, Y. Yin, Efficient mining of partial periodic patterns in time series database, in: Proceedings of the 15th International Conference on Data Engineering, Sydney, Australia, 1999, pp. 106–115.
[19] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, ACM Press, New York, 2000, pp. 1–12.
[20] B. Ozden, S. Ramaswamy, A. Silberschatz, Cyclic association rules, in: Proceedings of the 14th International Conference on Data Engineering, Orlando, Florida, IEEE Computer Society, 1998, pp. 412–421.
[21] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu, PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth, in: Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany, IEEE Computer Society, 2001, pp. 215–224.
[22] M.S. Tsechansky, N. Pliskin, G. Rabinowitz, A. Porath, Mining relational patterns from multiple relational tables, Decision Supports Systems 27 (1999) 177–195.
[23] C. Li, P.S. Yu, V. Castelli, Hierarchyscan: a hierarchical similarity search algorithm for databases of long sequences, in: Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, IEEE Computer Society, 1996, pp. 546–553.
[24] H. Mannila, H. Toivonen, A.I. Verkamo, Discovery of frequent episodes in event sequences, Data Mining Knowledge Discovery 1 (3) (1997) 259–289.
[25] D.J. Cook, L.B. Holder, Graph-based data mining, IEEE Intell. Systems 15 (2) (2000) 32–41.
[26] M.J. Zaki, Efficient enumeration of frequent sequences, in: Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, Bethesda, Maryland, ACM Press, New York, 1998, pp. 68–75.
[27] S. Gomory, R. Hoch, L. Lee, M. Podlaseck, E. Schonberg, Analysis and visualization on metrics for on-line merchandising, in: International WEBKDD'99Workshop, San Diego, California, USA, 1999, pp. 126–141.