Proceedings of the Third International Conference on Machine Learning and Cybernetics, Shanghai, 26-29 August 2004

AN ALGORITHM FOR MINING GENERALIZED SEQUENTIAL PATTERNS

JIA-DONG REN[1], YIN-BO CHENG[2], LIANG-LIANG YANG[2]

[1]Modern Education and Technology Center, Yanshan University, Qinhuangdao 066004, China
[2] College of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China
E-MAIL: jdren@ysu.edu.cn, cyb@ysu.edu.cn, yll263@163.com

Abstract:

Sequential pattern mining is an important data mining problem with broad applications. Algorithm GSP discovers generalized sequential patterns. However, GSP still encounters problems when a sequence database is large and/or when sequential patterns to be mined are long. Algorithm PrefixSpan mines complete sequential patterns faster than GSP but it cannot mine generalized sequential patterns with time constraints, time windows and/or taxonomy.

In this paper, a new enhanced method based on PrefixSpan, is proposed, called EPSpan, which absorbs the spirit of PrefixSpan and extends PrefixSpan towards mining generalized sequential patterns.

Keywords:

Sequential pattern mining; Generalized sequential patterns

1. Introduction

The sequential pattern mining problem was first introduced by Agrawal and Srikant in [1], which discovers frequent subsequences as patterns in a sequence database. An example of such a pattern is that customers typically rent "Star Wars", then "Empire Strikes Back" and then "Return of the Jedi".

Many studies and improvement methods have contributed to sequential pattern mining to make it more efficient, e.g., GSP discovers these generalized sequential patterns [2]. WINEPI finds all episodes from a given class of episodes that are frequent enough and MINEPI discovers frequent episodes in sequential data [3]. FP-growth mines the complete set of frequent patterns by pattern fragment growth [4]. FreeSpan integrates the mining of frequent sequences with that of frequent patterns and uses projected sequence databases to confine the search and the growth of subsequence fragments[5]. PrefixSpan explores prefix-projection in sequential pattern mining[6].

Almost all of the previously proposed methods for mining sequential patterns and other time-related frequent patterns are Apriori-like, i.e., based on the Apriori property proposed in association mining [7], which states the fact that any super-pattern of a nonfrequent pattern cannot be frequent.

Agrawal and Srikant generalized the problem definition to incorporate time constraints, sliding time windows, and taxonomies in sequential patterns and presented a new algorithm GSP (Generalized Sequential Patterns) in [2]. GSP is a typical Apriori-like method. The Apriori-like sequential pattern mining methods bear three nontrivial, inherent costs which are independent of detailed implementation techniques: ① Potential huge set of candidate sequences; ② Multiple scans of databases; ③ Difficulties at mining long sequential patterns.

Algorithm PrefixSpan is a kind of pattern-growth method. Its general idea is to examine only the prefix subsequences and project only the prefix subsequences and project only their corresponding postfix subsequences into projected databases. PrefixSpan mines complete sequential patterns faster than GSP but it cannot mine sequential patterns with time constraints, time windows and/or taxonomy.

In this paper, we propose a new enhanced method based on PrefixSpan, called EPSpan, which absorbs the spirit of PrefixSpan and extends PrefixSpan towards mining generalized sequential patterns. First EPSpan find all largest-sequences which each data-sequence support. Then EPSpan prunes repetitious sequences that are generated from a data-sequence. So we need not consider the time or other time-related factors in the subsequent phase. Finally, we find the complete set of sequential patterns by that all largest-sequences are passed to PrefixSpan as input parameters.

The rest of this paper is organized as follows. Section 2 states the problem of mining generalized sequential patterns and PrefixSpan. The algorithm EPSpan is described in Section 3. Section 4 describes the experiment results. Finally, Section 5 concludes the paper with future work.

0-7803-8403-2/04/$20.00 ©2004 IEEE

2. Problem definitions and PrefixSpan

2.1. Mining generalized sequential patterns

In this section, we define the problem of generalized sequential pattern mining.

Let I=$\{i_1, i_2, \cdots, i_n\}$ be set of all items. An itemset is a non-empty of items. A sequence is an ordered list of itemsets. A sequence s is denoted by $\langle s_1 s_2 \cdots s_l \rangle$, where s_j is an itemset, i.e., $s_j \subseteq I$ for $1 \leqslant j \leqslant l$. s_j is also called an element of the sequence, and denoted as $(x_1 x_2 \cdots x_m)$, where $x_k \in I$ for $1 \leqslant k \leqslant m$. For brevity, the brackets are omitted if an element has only one item. That is, element (x) is written as x. An item can occur at most once in an element of a sequence, but can occur multiple times in different elements of a sequence. The number of instances of items in a sequence is called the length of the sequence. A sequence with length l is called an l-sequence. A sequence $a = \langle a_1 a_2 \cdots a_n \rangle$ is called a subsequence of $\beta = \langle b_1 b_2 \cdots b_m \rangle$ and β a super sequence of a, denoted as $a \subseteq \beta$, if there exist integers $1 \leqslant j_1 < j_2 < \cdots < j_n \leqslant m$ such that $a_1 \in b_{j1}$, $a_2 \in b_{j2}$, $\cdots$, $a_n \in b_{jn}$.

A sequence database S is a set of tuples $\langle sid, s \rangle$, where sid is a sequence_id and s is a sequence. A tuple $\langle sid, s \rangle$ is said to contain a sequence a, if a is a subsequence of s, i.e., $a \subseteq s$.

The support for a sequence is defined as the fraction of total data-sequences that "contain" this sequence. A data-sequence contains a sequence s if s is a subsequence of the data-sequence [1]. Starting with the definition, taxonomies, sliding windows, and time constraints are added [2]:

- **Plus taxonomies:** A transaction T contains an item $x \in I$ if x is in T or x is an ancestor of some item in T. A transaction T contains an itemset $y \in I$ if T contains every item in y. A data-sequence $d = \langle d_1 d_2 \cdots d_m \rangle$ contains a sequence $s = \langle s_1 s_2 \cdots s_n \rangle$ if there exist integers $i_1 < i_2 < \cdots < i_n$ such that s_1 is contained in d_{i1}, s_2 is contained in $d_{i2}, \cdots$, s_n is contained in d_{in}. If there is no taxonomy, this degenerates into a simple subsequence test.
- **Plus sliding windows:** The sliding window generalization relaxes the definition of when a data-sequence contributes to the support of a sequence by allowing a set of transactions to contain an element of a sequence, as long as the difference in transaction-times between the transactions in the set is less than the user-specified window-size. Formally, a data-sequence d= $\langle d_1 d_2 \cdots d_m \rangle$ contains a sequence $s = \langle s_1 s_2 \cdots s_n \rangle$ if there exist integers $l_1 \leqslant u_1 < l_2 \leqslant u_2 \cdots < l_n \leqslant u_n$ such that

  1. s_i is contained in $\bigcup_{k=li}^{ui} d_k, 1 \leqslant i \leqslant n$, and
  2. transaction-time(d_{ui}) − transaction-time(d_{li}) ⩽ window-size, $1 \leqslant i \leqslant n$

- **plus time constraints:** Time constraints restrict the time gap between sets of transactions that contain consecutive elements of the sequence. Given user-specified window-size, max-gap and min-gap, a data-sequence $d = \langle d_1 d_2 \cdots d_m \rangle$ contains a sequence $s = \langle s_1 s_2 \cdots s_n \rangle$ if there exist integers $l_1 \leqslant u_1 < l_2 \leqslant u_2 \cdots < l_n \leqslant u_n$ such that

  1. s_i is contained in $\bigcup_{k=li}^{ui} d_k, 1 \leqslant i \leqslant n$,
  2. transaction-time(d_{ui}) − transaction-time(d_{li}) ⩽ window-size, $1 \leqslant i \leqslant n$
  3. transaction-time(d_{li}) − transaction-time(d_{ui-1}) > min-gap, $2 \leqslant i \leqslant n$, and
  4. transaction-time(d_{ui}) − transaction-time(d_{li-1}) ⩽ max-gap, $2 \leqslant i \leqslant n$

The problem of mining generalized sequential patterns can be stated as follows: Given a database D of data-sequences, a taxonomy T, user-specified min-gap and max-gap time constraints, and a user-specified sliding-window size, the problem of mining sequential patterns is to find all sequences whose support is greater than the user-specified minimum support. Each such sequence represents a sequential pattern, also called a frequent sequence.

2.2. PrefixSpan

PrefixSpan is a novel, scalable, and efficient at mining of sequential patterns. Its major idea is that, instead of projecting sequence databases by considering all the possible occurrences of frequent subsequences, the projection is based only on frequent prefixes because any frequent subsequence can always be found by growing a frequent prefix.

More details about algorithm PrefixSpan are given in [6]. Here is an example of PrefixSpan. For the sequence database S in Table 1 with min-sup = 2, sequential patterns in S can be mined by a prefix-projection method in the following steps:

Table 1. A sequence database

Sequence_id	Sequence
10	$\langle a(abc)(ac)d(cf) \rangle$
20	$\langle (ad)c(bc)(ae) \rangle$
30	$\langle (ef)(abc)(df)cb \rangle$
40	$\langle eg(af)abc \rangle$

Step 1: Find length-*l* sequential patterns. Scan once to find all frequent items in sequences. Each of these frequent items is a length-*1* sequential pattern. They are ⟨*a*⟩ :4, ⟨*b*⟩ :4, ⟨*c*⟩ :4, ⟨*d*⟩ :3, ⟨*e*⟩ :3, ⟨*f*⟩ :3, where ⟨pattern⟩ :count represents the pattern and its associated support count.

Step 2:Divide search space. The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones having prefix ⟨*a*⟩ ;⋯; and (6) the ones having prefix ⟨*f*⟩ .

Step 3: Find subsets of sequential patterns. The subsets of sequential patterns can be mined by constructing corresponding projected databases and mine each recursively. The mining process is explained as follows.

First, let us find sequential patterns having prefix ⟨*a*⟩ . Only the sequences containing ⟨*a*⟩ should be collected. Moreover, in a sequence containing ⟨ *a* ⟩ only the subsequence prefixed with the first occurrence of ⟨*a*⟩ should be considered. For example, in sequence ⟨(*cf*)(*ab*)(*df*)*cb*⟩ , only the subsequence ⟨(*_b*)(*df*)*cb*⟩ should be considered for mining sequential patterns having prefix ⟨*a*⟩ . Notice that (*_b*) means that the last element in the prefix, which is *a*, together with *b*, form one element. As another example, only the subsequence ⟨ (*abc*)(*ac*)*d*(*cf*) ⟩ of sequence ⟨*a*(*abc*)(*ac*)*d*(*cf*)⟩ should be considered.

Sequences in *S* containing ⟨*a*⟩ are projected with respect to ⟨*a*⟩ to form the ⟨*a*⟩ -projected database, which consists of four postfix sequences: ⟨ (*abc*)(*ac*)*d*(*cf*) ⟩ , ⟨(*_d*)*c*(*bc*)*d*(*ae*)⟩, ⟨(*_b*)(*df*)*cb*⟩⟩, ⟨(*_f*)*cbc*⟩. By scanning ⟨*a*⟩ -projected database once, all the length-2 sequential patterns having prefix ⟨a⟩ can be found. They are: ⟨*aa*⟩ :2, ⟨*ab*⟩ :3, ⟨(*ab*)⟩ :2, ⟨(*ac*)⟩ :4, ⟨*ad*⟩ :2, and ⟨*af*⟩ :2.

Recursively, all sequential having patterns prefix ⟨*a*⟩ can be partitioned into 6 subsets: (1) those having prefix ⟨*aa*⟩ , (2) those having prefix ⟨*ab*⟩ ,⋯, (6) those having prefix ⟨*af*⟩ . These subsets can be mined by constructing respective projected databases and mining each recursively as follows:

The ⟨*aa*⟩ -projected database consists of only one non-empty (postfix) subsequences having prefix ⟨*aa*⟩ : ⟨(*_bc*)(*ac*)*d*(*cf*)⟩ . Since there is no hope to generate any frequent subsequence from a single sequence, the processing of ⟨*aa*⟩ -projected database terminates.

The ⟨*ab*⟩ -projected database consists of three postfix sequences: ⟨(*_c*)(*ac*)*d*(*cf*)⟩ ⟨(*_c*)*ae*⟩ and ⟨*c*⟩ . Recursively mining ⟨*ab*⟩ -projected database returns four sequential patterns: ⟨(*_c*)⟩ , ⟨(*_c*)*a*⟩ , ⟨*a*⟩ and ⟨*c*⟩ (i.e. ⟨(*abc*)⟩ , ⟨(*abc*)*a*⟩ , ⟨*aba*⟩ and ⟨*abc*⟩).

⟨ (*ab*) ⟩ projected database contains only two sequences: ⟨(*_c*)(*ac*)*d*(*cf*)⟩ and ⟨(*df*)*cb*⟩⟩ , which leads to the finding of the following sequential patterns having prefix ⟨(*ab*)⟩ : ⟨*c*⟩ , ⟨*d*⟩ , ⟨*f*⟩ and ⟨*dc*⟩ .

The ⟨*ac*⟩ -, ⟨*ad*⟩ -, ⟨*af*⟩ - projected databases can be constructed and recursively mined similarly.

Similarly, we can find sequential patterns having prefix and ⟨*b*⟩ ⟨*c*⟩ ⟨*d*⟩ ⟨*e*⟩ and ⟨*f*⟩ , respectively, by constructing ⟨*b*⟩ -, ⟨*c*⟩ -, ⟨*d*⟩ -, ⟨*e*⟩ - and ⟨*f*⟩ -projected databases and mining them respectively.

3. EPSpan: efficient mining of generalized sequential patterns

We propose a new enhanced method based on PrefixSpan, called EPSpan, which is efficient at mining of generalized sequential patterns in large databases.

Taxonomies can be handled by adding all the ancestors of each item to the corresponding itemset. The basic approach and two optimizations were introduced in [2].

The method below eliminates time constraints and sliding windows. Given a sequence $s = \langle s_1\ s_2 \cdots s_n \rangle$ and a subsequence c , a taxonomy T, user-specified min-gap and max-gap time constraints, and a user-specified sliding-window size, c is a largest-sequence of s if s contains c and but s doesn't contain any supersequence of c. So if all largest-sequences supported by each data-sequence are found, time constraints and sliding windows are eliminated. This method reduces times of scanning original databases and leads to efficient mining.

We have the algorithm of finding largest-sequences as follows. Because of sliding windows and time constraints, start-time and end-time, which represent the first and last transaction-time of the itemset respectively, are added to each itemset in the algorithm [8].

Algorithm FindLargestSeq
Input: A data-sequence (which itemsets are sorted by transaction-time) $s = \langle s_1 s_2 \cdots s_n \rangle$; max-gap; min-gap; window-size
Output: largest-sequences
Method:
pSeq: a data-sequence
pSeqs: a set of data-sequence
nextSeq: the transaction that the next sequence begins at
n: length of the data-sequence
TS(item): a function that returns transaction-time of item
i, j: integer
begin
nextSeq = 1, *unfinished*=TRUE, *pSeqs* = ⟨⟩
while (*unfinished*) **do**

```
j=1, i=nextSeq, pSeq = ⟨⟩
unfinished=FALSE
while (i≤n) do
    if ISNULL(pSeq(j)) then
      pSeq(j).s = TS(s_i)
      pSeq(j).t = TS(s_i)
      pSeq(j).itemset = s_i
    else if TS(s_i) - pSeq(j).s ≤ window-size and
    (j=1or TS(s_i) - pSeq(j-1).s ≤max-gap) then
        pSeq(j).t = TS(s_i)
        pSeq(j).itemset= pSeq(j).itemset ∪ s_i
        if unfinished=FALSE then
            nextSeq = i
            unfinished=TRUE
        end if
    else if TS(s_i) - pSeq(j).s≤max-gap then
          if TS(s_i) - pSeq(j).t > min-gap then
            j=j+1
            pSeq(j).s = TS(s_i)
            pSeq(j).t = TS(s_i)
            pSeq(j).itemset = s_i
          else if unfinished=FALSE then
            nextSeq = i
            unfinished=TRUE
        end if
    else if unfinished=FALSE then
            nextSeq = i
            unfinished=TRUE
            break
        end if
    end if
    i=i+1
  end do
  pSeqs = pSeqs ∪ pSeq
 end do
end
```

Table 2. A sequence database

Transaction-Time	Items
10	1, 2
25	4, 6
50	3
55	2, 4
70	6

Example Consider the data-sequence shown in Table 2 and the case when max-gap is 20, min-gap is 10, and window-size is 5. We first find (1,2) at transaction-time 10, and then find (4,6) at transaction-time 25. The max-gap and min-gap between (1,2) and (4,6) are satisfied, we now move forward to (3). Since the gap between (4,6) and (3) is more than max-gap, we get the first largest-sequence ⟨(1,2), (4,6)⟩ . The new search begins at transaction-time 50. Since the gap between (3) and (2,4) is not more than window-size, (3) and (2,4) are put together to a new element (3,2,4), and the new search will begin at transaction-time 55. The start-time of (3,2,4) is transaction-time 50 and the end-time of (3,2,4) is transaction-time 55. Then the next element (6) is found. The max-gap and min-gap between (3,2,4) and (6) are satisfied, so we get ⟨(3,2,4),(6)⟩ . ⟨(3,2,4),(6)⟩ is the second largest-sequence. The new search begins at transaction-time 55. The max-gap and the min-gap between (2,4) and the next element (6) are satisfied. So ⟨(2,4), (6)⟩ is the third largest-sequence. The search is done.

If max-gap is 20, min-gap is 10, and window-size is 0, three largest-sequences are found. They are ⟨(1,2), (4,6)⟩ ⟨(3), (6)⟩ ⟨(2,4), (6)⟩ .

There probably exists repetitious sequences in largest-sequences that are generated from a data-sequence. EPSpan prunes repetitious sequences by checking them if they are contiguous subsequences of other sequences. Contiguous subsequence is defined as follows: given a sequence s= ⟨s_1 $s_2 \cdots s_n$⟩ and a subsequence c, c is a contiguous subsequence of s if any of the following conditions hold [2]:

1. c is derived from s by dropping an item from either s_1 or s_n.
2. c is derived from s by dropping an item from an element s_i which has at least 2 items.
3. c is a contiguous subsequence of c', and c' is a contiguous subsequence of s.

We first sort largest-sequences that are generated from each data-sequence by length descending order respectively. The reason for this is that the longer a sequence is, the more likely it is to be a supersequence of other sequences. Then a sequence is pruned if it is a contiguous subsequence of another sequence before it.

Actually, any contiguous subsequence of a frequent sequence is also frequent. So if all largest-sequences of data-sequences in sequence database are found, the complete set of sequential patterns can be found by that all largest-sequences are passed to PrefixSpan as input parameters.

Based on the above description, we have the algorithm EPSpan as follows.

Algorithm EPSpan
Input: A sequence database S, min-sup, max-gap, min-gap, window-size
Output: The complete set of sequential patterns
Method

begin
  $S' = \langle\rangle$
  for each data-sequence d **in** S
    FindLargestSeq(d, max-gap, min-gap, window-size)
    PruneRepetitiousSeq(pSeqs)
    $S' = S' \cup$ pSeqs
  next
  PrefixSpan($\langle\rangle$,0, S')
end

The correctness of algorithm EPSpan is based on that any contiguous subsequence of a frequent sequence is also frequent and the largest-sequences are largest sequences that contain all sequential patterns to be mined.

4. Experiment Results

The experiments are performed on a 550MHz Pentium-III PC with 256 megabytes main memory, running Microsoft Windows 2000 Server. All the methods are implemented using Microsoft Visual C++6.0. We compare EPSpan's performance with GSP.

The synthetic dataset we used was generated by the method described in [8]. To see the effect of the sliding window and time constraints on performance, we ran EPSpan on the synthetic dataset with window-size=10, min-gap=5 and max-gap=20. The experimental results of scalability with support threshold are shown as Figure 1.

The results show that EPSpan is more efficient than GSP, especially when the support threshold is low.

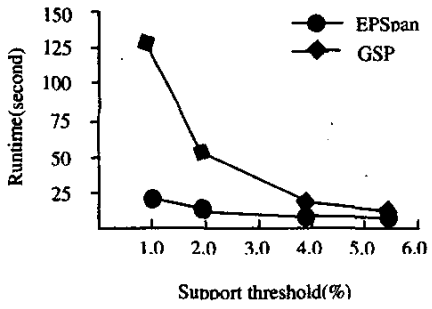

Figure 1. Performance Comparison

5. Conclusions

In this paper, we propose a new enhanced method based on PrefixSpan, called EPSpan. EPSpan mines generalized sequential patterns with time constraints, time windows and/or taxonomy efficiently. Its general idea is to find all largest-sequences which each data-sequence support and eliminate sliding windows, time constraints.

It is interesting to mine sequential patterns with other more complicated constraints and improve the performance of sequential patterns mining.

References

[1] R. Agrawal and R. Srikant. "Mining sequential patterns". In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[2] R. Srikant and R. Agrawal. "Mining sequential patterns: Generalizations and performance improvements". In *Proc. 5th Int. Conf. Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, Mar. 1996.

[3] H.Mannila, H. Toivonen, and A. I. Verkamo. "Discovery of frequent episodes in event sequences". Data Mining and Knowledge Discovery, 1:259-289,1997

[4] J. Han, J. Pei, and Y. Yin. "Mining frequent patterns without candidate generation". In Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data(SIGMOD'00), pages 1-12, Dallas, TX, May 2000.

[5] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, & M.-C. Hsu(2000). "FreeSpan: Frequent pattern-projected sequential pattern mining". Proceedings of 2000 International Conference on Knowledge Discovery and Data Mining, pp. 355–359.

[6] J. Pei,, J. Han, H. Pinto, Q. Chen, U. Dayal, & Hsu, M.-C. (2001). "PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth". Proceedings of 2001 International Conference on Data Engineering, pp. 215–224.

[7] R.Agrawa and R. Srikant. "Fast algorithms for mining association rules". In Proc. 1994 Int. Conf. Very large Data Bases(VLDB'94), pages 487-499, Santiago, Chile, Sept. 1994.

[8] Ming-rong Deng, Fu-gen Ye, et al. "Efficient algorithm for mining generalized sequential patterns". Journal of Zhejiang University(Science Edition), Vol 29 No.4, July 2002,p415-422.