The PSP Approach for Mining Sequential Patterns

F. Masseglia[2], F. Cathala[1,4], and P. Poncelet[1,3]

[1] LIM ESA CNRS 6077, Case 901, 163 Avenue de Luminy, 13288 Marseille Cedex 9, France E-mail: {poncelet,cathala}@lim.univ-mrs.fr
[2] LIRMM UMR CNRS 5506, 161, Rue Ada, 34392 Montpellier Cedex 5, France, E-mail: massegli@lirmm.fr
[3] IUT d'Aix-en-Provence
[4] Cemagref, division Aix-en-Provence, France

Abstract. In this paper, we present an approach, called PSP, for mining sequential patterns embedded in a database. Close to the problem of discovering association rules, mining sequential patterns requires handling time constraints. Originally introduced in [3], the issue is addressed by the GSP approach [10]. Our proposal resumes the general principles of GSP but it makes use of a different intermediary data structure which is proved to be more efficient than in GSP.

1 Introduction

Motivated by decision support problems, data mining, also known as knowledge discovery in databases, has been extensively addressed in the few past years (e.g. [5]). Among takled issues, the problem of mining association rules, initially introduced in [1], has recently received a great deal of attention [1, 2, 4, 5, 9, 11]. The problem of mining association rules is often referred to as the "market-basket" problem, because purchase transaction data collected by retail stores offers a typical application groundwork for discovering knowledge.
The concept of sequential pattern is introduced to capture typical behaviours over time, i.e. behaviours sufficiently repeated by individuals to be relevant for the decision maker [3].The GSP algorithm, proposed in [10], is intended for mining Generalized Sequential Pattern. It extends previous proposal by handling time constraints and taxonomies (*is-a* hierarchies).
In this paper we present an approach for discovering sequential patterns. It is widely inspired from the GSP algorithm, but it introduces some improvements which makes it possible to perform retrieval optimizations.
This paper is organized as follows. In section 2, the problem is stated and illustrated. An outline of GSP is given in section 3. Our proposal is detailed in section 4, and compared with GSP.

2 Mining maximal sequential pattern

This section widely resumes the formal description of the "market-basket" problem, introduced in [10].

First of all, we assume that we are given a database D of customers' transactions, each of which having the following characteristics: sequence-id or customer-id, transaction-time and the items involved in the transaction. Such a database is called a base of data sequences (Cf. Fig. 1).

Definition 1 Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals called *items*. An *itemset* is a non-empty set of items. A sequence s is a set of itemsets ordered according to their time-stamp. It is denoted by $< s_1 s_2 ... s_n >$ where s_j is an itemset. A *k-sequence* is a sequence of k-items (or of length k). A sequence $< s_1 s_2 ... s_n >$ is a sub-sequence of another sequence $< s'_1 s'_2 ... s'_m >$ if there exist integers $i_1 < i_2 < ... < i_n$ such that $s_1 \subseteq s'_{i_1}, s_2 \subseteq s'_{i_2}, ... s_n \subseteq s'_{i_n}$.

For aiding efficiently decision making, the aim is discarding non typical behaviours according to user's viewpoint. Performing such a task requires providing data sub-sequence s in the DB with a support value ($supp(s)$) giving its number of actual occurrences in the DB. In order to decide whether a sequence is frequent or not, a minimum support value (σ) is specified by user, and the sequence s is said frequent if the condition $supp(s) \geq \sigma$ holds.
From the problem statement presented so far, discovering sequential patterns resembles closely to mining association rules. However, elements of handled sequences are itemsets and not items, and a main difference is introduced with time concerns.
The user can decide that it does not matter if items were purchased separately as long as their occurrences enfold within a given time window, thus itemsets in the data sequence d could be grouped together with respect to the sliding window. Moreover when exhibiting from d, sub-sequences possibly matching with the supposed pattern, non adjacent itemsets in d could be picked up successively. Minimum and maximum time gaps are introduced to constrain such a construction. Window size and time constraints as well as the minimum support condition are parametrized by user as defined in [10].

Example 1 Let us consider the base D given in figure 1, reporting facts about a population merely reduced to four customers. Let us assume that the minimum support value is 50%, thus to be considered as frequent a sequence must be observed for at least two customers. The only frequent sequences, embedded in the DB are the following: $<$ (20) (90) $>$, $<$ (30) (90) $>$ and $<$ (30) (40, 70) $>$. By introducing a sliding window of 2 days, a new frequent sequence $<$ (20 30) (90) $>$ is discovered because it matches with the first transaction of C_4 while being detected for C_1, within a couple of transactions respecting the window size.

3 Related Work

This section is devoted for setting the groundwork of our proposal and not offering an overview of the whole domain. The interested reader could refer to [1, 2, 4, 5, 11] in which approaches for discovering association rules are presented and compared. In a different context, the issue of exhibiting sequences is addressed

Customer	Time	Items	Customer	Time	Items
C_1	01/01/1998	20,60	C_3	05/01/1998	30,50,70
C_1	02/02/1998	20	C_3	12/02/1998	10,20
C_1	04/02/1998	30			
C_1	18/02/1998	80,90			
C_2	11/01/1998	10	C_4	06/02/1998	20,30
C_2	12/01/1998	30	C_4	07/02/1998	40,70
C_2	29/01/1998	40,60,70	C_4	08/02/1998	90

Fig. 1. A data-sequence database example

in [3, 6, 10]. Since it is the basis of our approach, particular emphasis is placed on the GSP approach.
Basically, exhibiting frequent sequences requires firstly retrieving all data sequences satisfying the specified time constraints. These sequences are considered as candidates for being patterns. The support of candidate sequences is then computed by browsing the DB. Sequences for which the minimum support condition does not hold are discarded. The result is the set of frequent sequences.
For building up candidate and frequent sequences, the GSP algorithm performs several iterative steps such as the k^{th} step handles sets of k-sequences which could be candidate (the set is noted C_k) or frequent (in L_k). The latter set, called seed set, is used by the following step which, in turn, results in a new seed set encompassing longer sequences, and so on.
The first step aims to compute the support of each item in the database. When completed, frequent items (i.e. satisfying the minimum support) are discovered. They are considered as frequent 1-sequences (sequences having a single itemset, itself being a singleton). This initial seed set is the starting point of the second step. The set of candidate 2-sequences is built according to the following assumption: candidate 2-sequences could be any couple of frequent items, embedded in the same transaction or not. From this point, any step k is given a seed set of frequent (k-1)-sequences and it operates by performing the two following sub-steps:

- The first sub-step (join phase) addresses candidate generation. The main idea is to retrieve, among sequences in L_{k-1}, couples of sequences (s, s') such that discarding the first element of the former and the last element of the latter results in two sequences fully matching. When such a condition holds for a couple (s, s'), a new candidate sequence is built by appending the last item of s' to s.
- The second sub-step is called the prune phase. Its objective is yielding the set of frequent k-sequences L_k. L_k is achieved by discarding from C_k, sequences not satisfying the minimum support. For yielding such a result, it is necessary to count the number of actual occurrences matching with any possible candidate sequence.

Candidate sequences are organized within a *hash-tree* data-structure which can be accessed efficiently. These sequences are stored in the leaves of the tree while intermediary nodes contain hashtables. Each data-sequence d is hashed to find the candidates contained in d. When browsing a data sequence, time constraints must be managed. It is performed by navigating through the tree in a downward or upward way resulting in a set of possible candidates. For each candidate, GSP checks whether it is contained in the data-sequence. Because of the sliding window, minimum and maximum time gaps, it is necessary to switch during examination between forward and backward phases. Forward phases are performed for dealing progressively items. Let us notice that during this operation the min-gap condition applies in order to skip itemsets too close from their precedent. And while selecting items, sliding window is used for resizing transaction cutting. Backward phases are required as soon as the max-gap condition no longer holds.

4 The PSP approach

Our approach fully resumes the fundamental principles of GSP. Its originality is to use a different hierarchical structure than in GSP for organizing candidate sequences, in order to improve efficiency of retrievals.
The general algorithm [7] is similar than in GSP. At each step k, the DB is browsed for counting the support of current candidates (procedure CANDIDATE-VERIFICATION). Then the frequent sequence set L_k can be built. From this set, new candidates are exhibited for being dealt at the next step (procedure CANDIDATE-GENERATION). The algorithm stops when the longest frequent sequences, embedded in the DB are discovered thus the candidate generation procedure yields an empty set of new candidates. Support is a function giving for each candidate its counting value stored in the tree structure.
The tree structure, managed by the algorithms, is a *prefix-tree* close to the structure used in [8]. At the k^{th} step, the tree has a depth of k. It captures all the candidate k-sequences in the following way. Any branch, from the root to a leaf stands for a candidate sequence, and considering a single branch, each node at depth l ($k \geq l$) captures the l^{th} item of the sequence. Furthermore, along with an item, a terminal node provides the support of the sequence from the root to the considered leaf (included). Transaction cutting is captured by using labelled edges. More precisely, let us consider two nodes, one being the child of the other. If the items embodied in the nodes originally occurred during different transactions, the edge linking the nodes is labelled with a '-' otherwise it is labelled with a '+' (dashed link in figure 2).

Example 2 Let us assume that we are given the following set of frequent 2-sequences : $L_2 = \{<(10)\ (30)>,\ <(10)\ (40)>,\ <(30)\ (20)>,\ <(30\ 40)>,\ <(40\ 10)>\}$. It is organized according to our tree structure as depicted in figure 2. Each terminal node contains an item and a counting value. If we consider the node having the item **40**, its associated value 2 means that two occurrences of the sequence $\{<(10)\ (40)>\}$ have been detected so far.

Proposition 1 *Our structure requires less memory than the tree used in the GSP approach.*

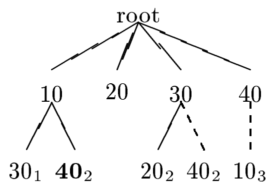

Fig. 2. Tree data structure

In GSP, due to the adopted tree structure, all candidates sequences are preserved and fully stored in the leaves. We argue that our prefix-tree structure is less costly from a memory viewpoint because it organizes candidates according to their common elements. In fact, initial sub-sequences common to several candidates are stored only once.

Let us now detail how candidates and data-sequences are compared through the CANDIDATE-VERIFICATION algorithm. The data sequence is progressively browsed starting with its first item. Its time-stamp is preserved in the variable l_a. Then successive items in d are examined and the variable u_a is used for giving the time-stamp of the current item. Of course if u_a - l_a = 0, the couple of underlying items (and all possible items between them) appears in a single transaction. When u_a becomes different from l_a, this means that the new selected item belongs to a different transaction. However, we cannot consider that performed so far the algorithm has detected the first itemset of d because of the sliding window. Thus the examination must be continued until the selected item is too far from the very first item of d. The condition u_a - $l_a \geq ws$ does no longer hold. At this point, we are provided with a set of items (I_p). For each frequent item in I_p (it matches with a node at depth 1) the function FINDSEQUENCE is executed in order to retrieve all candidates supported by the first extracted itemset. The described process is then performed for exhibiting the second possible itemset. l_a is set to the time-stamp of the first itemset encountered and once again u_a is progressively incremented all along the examination. The process is repeated until the last itemset of the sequence has been dealt.

CANDIDATE VERIFICATION ALGORITHM

input: T the tree containing all candidate and frequent sequences, a data-sequence d and its sequence identifier *idseq*. The step k of the General Algorithm.

output: T the set of all candidate sequences contained in d.

$l_a = FirstItemSet(d).time();$

while $(l_a <= LastItemSet(d).time())$ **do**

  $u_a = l_a;$

  while $((u_a - l_a) < ws)$ **do**

    $I_p = \{i_p \in d / i_p.time() \in [l_a, u_a]\};$

    for each $i_p \in I_p$ **do**

      if $(i_p \in root.Children)$ **then**

```
      depth = 0;
      FindSequence(l_a, u_a, root.Children(i_p), i_p, d, idseq, depth);
    u_a = (u_a.succ()).time() ;
  l_a = (l_a.succ()).time();
```

The function FindSequence is successively called by the previous algorithm for retrieving candidate sequences firstly beginning with a sub-set of the first item of d, then with the second, and so on. When a leaf is reached, the examined sub-sequence supports the candidate and its counting value must be incremented.

Find Sequence Algorithm
input: Two integers l_a, u_a standing for the itemset size, N, a node of T, i the item in d, the depth of the go down on the tree ($depth$).
output: T updated with respect to constraint times.

```
if (leaf(N) and depth = k) then
  if (idseq ≠ N.idlast) then
    N.idlast = idseq; N.cpt++;
else
  /* same transaction */
  I_p = {i_p ∈ d/i_p follows i and i_p.time() ∈ [l_a, u_a]};
  for each i_p ∈ I_p do
    if (i_p ∈ N.Same) then
      FindSequence(l_a, u_a, N.Same(i_p), i_p, d, idseq, depth + 1);
  /* other transaction */
  l_b = (u_a.succ()).time(); /*mingap constraint*/
  while ((l_b − u_a) < mingap) do l_b = (l_b.succ()).time();
  while (l_b ≠ LastItem(d).time()) do
    u_b = l_b;
    while ((u_b − l_b) < ws and (u_b − l_a) < maxgap) do
      I_p = {i_p ∈ d/i_p.time() ∈ [l_b, u_b]};
      for each i_p ∈ I_p do
       if (i_p ∈ N.Other) then
        FindSequence(l_b, u_b, N.Other(i_p), i_p, d, idseq, depth + 1);
      u_b = (u_b.succ()).time();
    l_b = (l_b.succ()).time();
```

When all the candidates to be examined are dealt, the tree is pruned in order to minimize required memory space. All leaves not satisfying the minimum support are removed. When such deletions complete, the tree no longer captures candidate sequences but instead frequent sequences.

Theorem 1 *For all data-sequence d and for all candidate sequence c in the tree T, if c is a sub-sequence of d then c.support will be incremented by* Candidate-Verification *and for all candidate sequence c' in T, if c'.support is incremented by* Candidate-Verification *then c' is a sub-sequence of d.*

Due to space limitation, we do not provide the proof of the theorems which could be found in [7].

The algorithm of candidate generation (defined in [7]) builds, step by step, the tree structure. At the beginning of step 2, the tree has a depth of 1. All nodes at depth 1 (frequent items) are provided with children supposed to capture all frequent items. This means that for each node, the created children are a copy of its brothers. When the k^{th} step of the general algorithm is performed, the candidate generation operates on the tree of depth k and yields the tree of depth $k+1$. For each leaf in the tree, we must compute all its possible continuations of a single item. Exactly like at step 2, only frequent items can be valid continuations. Thus only items captured by nodes at depth 1 are considered. Moreover we refine this set of possible itemsets by discarding those which are not captured by a brother of the dealt leaf. The basic idea under such a selection is the following. Let us consider a frequent k-sequence s and assume that s extended with a frequent item i is still frequent. In such a case, $s' = < s_1 s_2 ... s_{k-1}\ i >$ must necessarily be exibited during the candidate verification phase. Thus s' is a frequent k-sequence and its only difference with s is its terminal item. Associated leaves, by construction of the tree, are brothers.

Theorem 2 *Given a database D, for each sequence of length k, the structures used in GSP and in our approach capture the very same set of candidate sequences.*

Experiments

The proposed approach is implemented on an Ultra Sparc with 256 MB main memory. For experimentation, we generate synthetic customer transactions using the data generation program of [2][1]. Due to space limitation, we do not provide detailed results which could be found in [7]. Figure 3 gives the execution times of our algorithm applied to two DB examples.

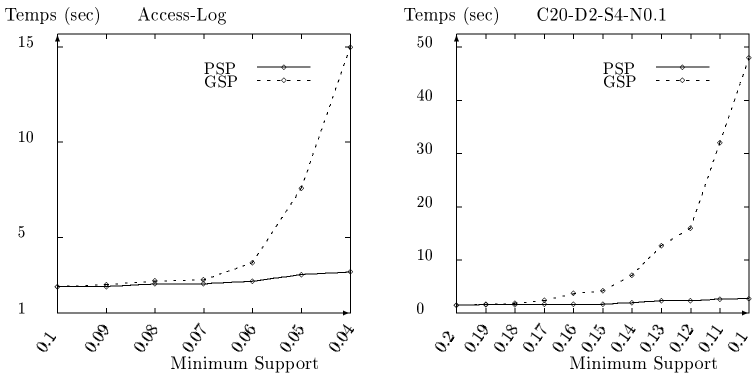

Fig. 3. Execution times

[1] Available at the following URL (*http://www.almaden.ibm.com/cs/quest*).

Discussion
Although our approach resumes GSP principles, we believe that the proposed prefix-structure is more efficient than the tree structure used in GSP.
Before explaining why, let us have a comparative illustration.

Example 3 Figure 4 depicts the data structures used in our approach (left tree) and in GSP (right tree), managed at the very same step of the general algorithm. More precisely, from the frequent 2-sequences given in example 2, candidate 3-sequences are obtained. Thus we have $C_3 =< (10)\ (40\ 10) > < (10)\ (30)\ (20) >$ $< (10)\ (30\ 40) > < (40\ 10)\ (30) > < (40\ 10)\ (40) >$. As stated in proposition 1, the number of items stored in our structure is significantly reduced compared with GSP.

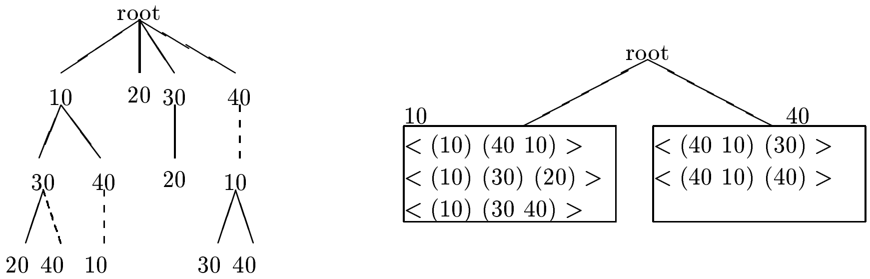

Fig. 4. Illustration of the prefix-tree and hash-tree structures

During the candidate verification phase in GSP, a navigation is performed through the tree until reaching a leaf storing several candidates. Then the algorithm operates a costly backtracking for examining each sequence stored in the leaf. In our approach, retrieving candidates means a mere navigation through the tree. Once a leaf is reached, the single operation to be performed is incrementing the support value.
In the tree structure of GSP, sequences grouped in terminal nodes share a common initial sub-sequence. Nevertheless, this feature is not used for optimizing retrievals. In fact, during the candidate verification phase, the GSP algorithm examines each sequence stored in the leaf from its first item to the last. In our approach, we make advantage of the proposed structure: all terminal nodes (at depth k) which are brothers stand for continuations of a common (k-1)-sequence. Thus it is costly and not necessary to examine this common sequence for all k-sequences extending it.
Moreover, the advantage of our tree-structure is increased by applying the following ideas. Let us imagine that a frequent k-sequence is extended to capture several (k+1)-candidates. Once the latter are proved to be unfrequent, they are of course pruned from the tree and the k-sequence is provided with a mark.

This mark avoids to attempt building possible continuations of the considered sequence during further steps. The mark is also used in order to avoid testing j-sequences $(2 < j < k)$.
Furthermore, at each step when a candidate k-sequence c is proved to be frequent, its possible sub-sequences of length l $(2 < l < k)$ ending with the k-1^{th} item of c are examined. For each of which matching with a candidate l-sequence, the considered l-sequence is pruned from the tree. In fact, such sub-sequences are no longer relevant since longer sequences continuing them are discovered. Applying this principle reduces the number of stored candidates.

5 Conclusion

The presented approach addresses the problem of mining sequential patterns within a DB of behavioural facts. We adopt the general principles defined by the GSP algorithm but propose a different data structure for storing candidate and frequent sequences. The proposed algorithms for handling this structure take advantages of its semantics for avoiding useless and costly operations when verifying candidates. Furthermore, the presented tree structure is proved to require less memory than in GSP for storing candidate sequences.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proc. of the SIGMOD'93*, Washington, 1993.
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Generalized Association Rules. In *Proc. of the VLDB'94*, Santiago, Chile, September 1994.
3. R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proc. of the ICDE'95*, Tapei, Taiwan, March 1995.
4. S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. of the SIGMOD'97*.
5. U.M. Fayad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1996.
6. H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery*, 1(3), 1997.
7. F. Masseglia. Le pré-calcul appliqué à l'extraction de motifs séquentiels en data mining. Technical report, LIRMM, France, June 1998.
8. A. Mueller. Fast Sequential and Parallel Algorithms for Association Rules Mining: A Comparison. Technical Report CS-TR-3515, Univ. Maryland-College, 1995.
9. A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proc. of the VLDB'95*, Zurich, 1995.
10. R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the EDBT'96*, Avignon, France, Sept 1996.
11. H. Toivonen. Sampling Large Databases for Association Rules. In *Proc. of the VLDB'96*, September 1996.