Pre-Processing Time Constraints for Efficiently Mining Generalized Sequential Patterns

Florent Masseglia
INRIA Sophia Antipolis
AxIS Research Team
2004 rte des lucioles
06902 Sophia Antipolis, France
Florent.Masseglia@inria.fr

Pascal Poncelet
Laboratoire LGI2P
École des Mines d'Alès
Parc Sc. Georges Besse
30035 Nîmes, France
Pascal.Poncelet@ema.fr

Maguelonne Teisseire
LIRMM UMR CNRS 5506
161 Rue Ada
34392 Montpellier, France
Teisseire@lirmm.fr

Abstract

In this paper we consider the problem of discovering sequential patterns by handling time constraints. While sequential patterns could be seen as temporal relationships between facts embedded in the database, generalized sequential patterns aim at providing the end user with a more flexible handling of the transactions embedded in the database. We propose a new efficient algorithm, called GTC (Graph for Time Constraints) for mining such patterns in very large databases. It is based on the idea that handling time constraints in the earlier stage of the algorithm can be highly beneficial since it minimizes computational costs by preprocessing data sequences. Our test shows that the proposed algorithm performs significantly faster than a state-of-the-art sequence mining algorithm.

1. Introduction

Although sequential patterns are of great practical importance (e.g. alarms in telecommunications networks, identifying plan failures, analysis of Web access databases, etc.), in the literature, they have received relatively little attention [1, 6, 4, 5, 2]. They could be seen as temporal relationships between facts embedded in the database. A sequential pattern could be "5% of customers bought 'Foundation', then 'Foundation and Empire', and then 'Second Foundation"'. The problem of discovering sequential patterns in databases, introduced in [1], is to find all patterns verifying user-specified minimum support, where the support of a sequential pattern is the percentage of data-sequences that contain the pattern. Such patterns are called *frequent patterns*. In [6], the definition of the problem is extended by handling time constraints and taxonomies (*is-a* hierarchies) and a new algorithm, called GSP was proposed. In this paper, we propose a new efficient algorithm, called GTC (Graph for Time Constraints) for mining generalized sequential patterns in large databases. GTC minimizes computational costs by using a data-sequence preprocessing operation that takes time constraints into account. The main new feature of GTC is that time constraints are handled prior to and separately from the counting step of a data sequence.

The rest of this paper is organized as follows. In Section 2, the problem of mining generalized sequential patterns is stated and illustrated. A review of related work is presented in Section 3. The reasons for our choices are discussed in Section 4. The GTC algorithm for efficiently discovering all generalized sequential patterns is given in Section 5. Section 6 presents the detailed experiments using both synthetic and real datasets, and performance results obtained are interpreted. Section 7 concludes the paper.

2. From Sequential Patterns to Generalized Sequential Patterns

First of all, we assume that we are given a database DB of customers' transactions, each of which has 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. Figure 4).

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 $s_1 \subseteq s'_{i_1}, s_2 \subseteq s'_{i_2}, ... s_n \subseteq s'_{i_n}$.*

Example 1 *Let us consider that a given customer purchased items* $10, 20, 30, 40, 50$*, according to the following sequence:* $s =<$ *(10) (20 30) (40) (50)*$>$*. This means that, apart from* 20 *and* 30 *which were purchased together, i.e. during a common transaction, the items in the sequence were bought separately.*
The sequence $s' =<$ *(20) (50)* $>$ *is a sub-sequence of* s *because (20)* $\subseteq$ *(20 30) and (50)*$\subseteq$ *(50). However* $<$ *(20) (30)* $>$ *is not a sub-sequence of* s *since items were not bought during the same transaction* □

In order to efficiently aid decision making, the aim is to discard non-typical behaviors according to the user's viewpoint. Performing such a task requires providing any data sub-sequence s in the DB with a support value ($supp(s)$) giving its number of actual occurrences in the DB[1]. This means comparing any sub-sequence with the whole DB. In order to decide whether a sequence is frequent or not, a minimum support value ($minSupp$) is specified by the user, and the sequence s is said to be frequent if the condition $supp(s) \geq minSupp$ holds.

This definition of sequence is not appropriate for many applications, since time constraints are not handled. In order to improve the definition, *generalized sequential patterns* are introduced [6].

When verifying whether a sequence is included in another one, transaction cutting enforces a strong constraint since only pairs of itemsets are compared. The notion of the sized sliding window enables that constraint to be relaxed. More precisely, 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, when browsing the DB in order to compare a sequence s, assumed to be a pattern, with all data sequences d in DB, itemsets in d could be grouped together with respect to the sliding window. Thus transaction cutting in d could be resized when verifying if d matches with s.
Moreover when exhibiting from the data sequence d, subsequences possibly matching with the assumed pattern, non-adjacent itemsets in d could be picked up successively. Minimum and maximum time gaps are introduced to constrain such a construction. In fact, to be successively picked up, two itemsets must occur neither too close nor too far apart in time. More precisely, the difference between their time-stamps must fit in the range $[min\text{-}gap, max\text{-}gap]$. Window size and time constraints as well as the minimum support condition are parameterized by the user. Mining sequences with time constraints allows a more flexible handling of the transactions.

We now define frequent sequences when handling time constraints:

Definition 2 *Given a user-specified minimum time gap (minGap), maximum time gap (maxGap) and a time window size (windowSize), a data-sequence* $d =< d_1...d_m >$ *is said to* $support$ *a sequence* $s =< s_1...s_n >$ *if there exist integers* $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ *such that: (i)* s_i *is contained in* $\cup_{k=l_i}^{u_i} d_k$*,* $1 \leq i \leq n$*; (ii) transaction-time* (d_{u_i}) *- transaction-time* (d_{l_i}) $\leq ws$*,* $1 \leq i \leq n$*; (iii) transaction-time* (d_{l_i}) *- transaction-time* ($d_{u_{i-1}}$) $> minGap$*,* $2 \leq i \leq n$*; (iv) transaction-time* (d_{u_i}) *- transaction-time* ($d_{l_{i-1}}$) $\leq maxGap$*,* $2 \leq i \leq n$*. The* $support$ *of* s*,* $supp(s)$*, is the fraction of all sub-sequences in* DB *supporting* s*. When* $supp(s) \geq minSupp$ *holds, being given a* $minimum\ support$ *value* $minSupp$*, the sequence* s *is called frequent.*

Example 2 *As an illustration for the time constraints, let us consider the following data-sequence describing the purchased items for a customer:*

Date	Items
01/01/2000	10
02/02/2000	20, 30
03/02/2000	40
04/02/2000	50, 60
05/02/2000	70

In other words, the data-sequence d *could be considered as follows:*

$$d =< (10)^1\ (20\ 30)^2\ (40)^3\ (50\ 60)^4\ (70)^5 >$$

where each itemset is stamped by its access day. For instance, $(50\ 60)^4$ *means that the items 50 and 60 were purchased together at time 4.*

Let us now consider a candidate sequence c=$<$ *(10 20 30 40) (50 60 70)* $>$ *and time constraints specified such as* $windowSize$*=3,* $minGap$*=0 and* $maxGap$*=5. The candidate sequence* c *is considered as included in the data-sequence* d *for the following two reasons:*

1. *the windowSize parameter makes it possible to gather together on the one hand the itemsets* $(10)\ (20\ 30)$ *and* (40)*, and on the other hand the itemsets* $(50\ 60)$ *and* (70) *in order to obtain the itemsets* $(10\ 20\ 30\ 40)$ *and* $(50\ 60\ 70)$*,*
2. *the minGap constraint between the itemsets* (40) *and* $(50\ 60)$ *holds.*

Considering the integers l_i *and* u_i *in the Definition 2, we have* $l_1 = 1,\ u_1 = 3,\ l_2 = 4,\ u_2 = 5$ *and the data sequence* d *is handled as illustrated in Figure 1.*

1 A sequence in a data-sequence is taken into account once and once only for computing the support of a frequent sequence even if several occurrences are discovered.

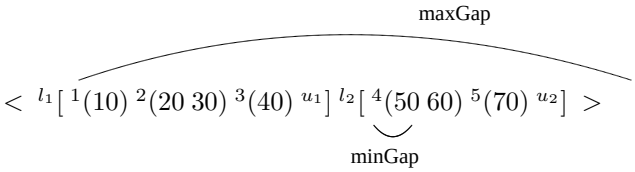

Figure 1. Illustration of the time constraints

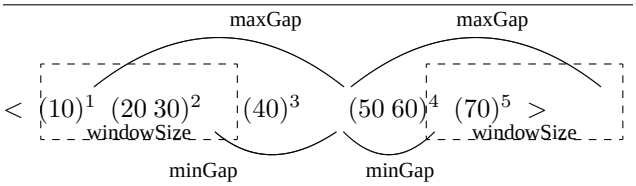

Figure 2. Illustration of the time constraints

In a similar way, the candidate sequence c=< (10 20 30) (40) (50 60 70) > with $windowSize$=1, $minGap$=0 and $maxGap$=2, i.e. $l_1 = 1,\ u_1 = 2,\ l_2 = 3,\ u_2 = 3,\ l_3 = 4$ and $u_3 = 5$ (C.f. Figure 2) is included in the data-sequence d.
The two following sequences c_1=< (10 20 30 40) (70) > and c_2 = < (10 20 30) (60 70) >, with $windowSize$=1, $minGap$=3 and $maxGap$=4 are not included in the data-sequence d. Concerning the former, the windowSize is not large enough to gather together the itemsets (10) $(20\ 30)$ and (40). For the latter, the only possibility for yielding both $(10\ 20\ 30)$ and $(60\ 70)$ is to take into account ws for achieving the following grouped itemsets $[(10)\ (20\ 30)]$ and $[(50\ 60)\ (70)]$. $maxGap$ is respected since $[(10)\ (20\ 30)]$ and $[(50\ 60)\ (70)]$ are spaced 4 days apart($u_2 = 5,\ l_1 = 1$). Nevertheless, in such a case $minGap$ constraint is no longer respected between the two itemsets because they are only 2 days apart ($l_2 = 4$ and $u_1 = 2$) whereas minGap was set to 3 days □

Given a database of data sequences, user-specified minGap and maxGap constraints and a user-specified sliding windowSize, the generalized sequential problem is to find all the sequences whose support is greater than the user-specified $minSupp$.

3. Related Work

In the following section, we review the most important work carried out within a sequential pattern framework. Since they consider the generalized sequence problem and as they are the basic of our approach, particular emphasis is placed on the GSP [6] and PSP [3] algorithms.

The concept of sequential pattern is introduced to capture typical behaviors over time, i.e. behaviors repeated sufficiently often by individuals to be relevant for the decision maker. The GSP algorithm, proposed in [6], is intended for mining Generalized Sequential Patterns. It extends previous proposal by handling time constraints and taxonomies (*is-a* hierarchies).

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 the 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 at computing 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 pair 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}, pairs of sequences (s, s') such that discarding the first element of the former and the last element of the latter results in two fully matching sequences. When such a condition holds for a pair (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. To yield 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. This is performed by navigating downward or upward through the tree, 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, and minimum and maximum time gaps, it is necessary to switch during examination between forward and

backward phases. Forward phases are performed for progressively dealing with items.

Accordingly, in earlier work [3], we proposed a new approach called PSP, Prefix-Tree for Sequential Patterns, which 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 retrieval efficiency. In the hash-tree structure managed by GSP, the transaction cutting is not captured. The main drawback of this approach is that when a leaf of the tree is obtained, an additional phase is necessary in order to check time constraints for all sequences embedded in the leaf.

The tree structure, managed by PSP, is a *prefix-tree*. 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 labeled edges.

Example 3 *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 3. 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* □

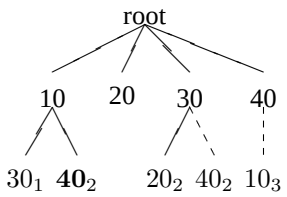

Figure 3. The PSP Tree data structure

4. Motivations

The PSP approach outperforms GSP by using a more efficient data structure. If such a structure seems appropriate to the problem of mining generalized sequential patterns, it seems, on the other hand, that the algorithm can be improved by paying particular attention to time constraint handling. To illustrate, let us consider the following customer data sequence $<$ (1) (2) (3) (4) (5) $>$ of the base given in Figure 4.

Client	Date	Item
C1	01/04/2000	1
C1	03/04/2000	2
C1	04/04/2000	3
C1	07/04/2000	4
C1	17/04/2000	5

Figure 4. A database example

Let us assume that windowSize = 1 and minGap=1. Upon reading the customer transaction, PSP has to determine all combinations of the data sequence in accordance with the time constraints in order to increment the support of a candidate sequence. Then the set of sequences verifying time constraints is the following:

without time constraints	with windowSize & minGap
$<$ (1) $>$	$<$ (2) (5) $>$ •
...	...
$<$ (5) $>$	$<$ (1) (3) (5) $>$ •
...	...
$<$ (1) (2) $>$	$<$ (1) (2 3) (5) $>$ •
...	...
$<$ (3) (4) $>$	$<$ (1) (2) (4) (5) $>$ •
...	$<$ (1) (3) (4) (5) $>$ •
$<$ (1) (2) (3) (4) (5) $>$	$<$ (1) (2 3) (4) (5) $>$ ○

We notice that the sequences marked by a • are included in the sequence marked by a ○. That is to say that if a candidate sequence is supported by $<$ (1) (2) (4) (5) $>$ or $<$ (1) (3) (4) (5) $>$ then such a sequence is also supported by $<$ (1) (2 3) (4) (5) $>$. The test of the two first sequences is of course useless because they are included in a larger sequence.

Client	Date	Item
C1	01/04/2000	1
C1	07/04/2000	2
C1	13/04/2000	3
C1	17/04/2000	4
C1	18/04/2000	5
C1	24/04/2000	6

Figure 5. A database example

Let us now have a closer look at the problem of the windowSize constraint. In fact, the number of included sequences is much greater when considering such a constraint. For instance, let us consider the database given in Figure 5. When windowSize=5 and minGap=1, PSP has to test the following sequences into the candidate tree structure (we only report sequences when windowSize is applied):

$<(1)(2)(3)(5)(6)>$ •
$<(1)(2)(3)(4)(6)>$ •
$<(1)(2)(3)(45)(6)>$ ∘
$<(1)(2)(34)(6)>$ ·
$<(1)(2)(345)(6)>$ ⊙

We notice that the sequences marked by a • (resp. ·) are included in the sequence marked by a ∘ (resp. ⊙). That is to say that we have only to consider the two following sequences $<(1)(2)(3)(45)(6)>$ and $<(1)(2)(3\ 4\ 5)(6)>$ when verifying a data sequence in the candidate tree structure.

In fact, we need to solve the following problem: how to reduce the time required for comparing a data sequence with the candidate sequences. Our proposition, described in the following section, is to precalculate, by means of the GTC (Graph for Time Constraints) algorithm, a relevant set of sequences to be tested for a data sequence. By precalculating this set, we can reduce the time spent analysizing a data sequence when verifying candidate sequences stored in the tree, in the following two ways: (i) The navigation through the candidate sequence tree does not depend on the time constraints defined by the user. This implies navigation without backtracking and better analysis of possible combinations of windowSize which are for PSP, as well as for GSP, computed on the fly.(ii) This navigation is only performed on the longest sequences, that is to say on sequences not included in other sequences.

5. GTC: Graph for Time Constraints

Our approach takes up all the fundamental principles of GSP. It contains a number of iterations. The iterations start at the size-one sequences and, at each iteration, all the frequent sequences of the same size are found. Moreover, at each iteration, the candidate sets are generated from the frequent sequences found at the previous iteration.

The main new feature of our approach which distinguish it from GSP and PSP is that handling of time constraint is done prior to and separate from the counting step of a data sequence.

Upon reading a customer transaction d in the counting phase of pass k, GTC has to determine all the maximal combinations of d respecting time constraints. For instance, in the previous example, only $<(1)(2)(3)(4\ 5)(6)>$ and $<(1)(2)(3\ 4\ 5)(6)>$ are exhibited by GTC. Then the Main algorithm has to determine all the k-candidates supported by the maximal sequences issued from GTC iterations on d and increment the support counters associated with these candidates without considering time constraints any more.

In the following sections, we decompose the problem of discovering non-included sequences respecting time constraints into the following subproblems. First we consider the problem of the minGap constraint without taking into account maxGap or windowSize and we propose an algorithm called GTC$_{minGap}$ for handling efficiently such a constraint. Second, we extend the previous algorithm in order to handle the minGap and windowSize constraints. This algorithm is called GTC$_{ws}$. Finally, we propose an extension of GTC$_{ws}$, called GTC, for discovering the set of all non included sequences when all the time constraints are applied.

5.1. GTC$_{minGap}$ Algorithm: solution for minGap

In this section, we explain how the GTC$_{minGap}$ algorithm provides an optimal solution to the problem of handling the minGap constraint.

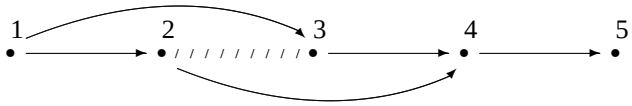

Figure 6. A data sequence representation

To illustrate, let us consider Figure 6, which represents the data sequence of the base given in Figure 4. We note, ////////, the minGap constraint between two itemsets a and b. Let us consider that minGap is set to 1. As items 2 and 3 are too closed according to minGap, they cannot occur together in a candidate sequence. So, from this graph, only two sequences ($<\ (1)\ (2)\ (4)\ (5)\ >$ and $<\ (1)\ (3)\ (4)\ (5)\ >$) are useful in order to verify candidate sequences. We can observe that these two sequences match the two paths of the graph beginning from vertex 1 (standing for source vertex) and ending in vertex 5 (standing for sink vertex).

From each sequence d, a *sequence graph* can be built. A *sequence graph* for d is a directed acyclic graph $G_d(V, E)$ where a vertex v, $v \in V$, is an itemset embedded in d and an edge e, $e \in E$, from two vertices u and v, denotes that u occurred before v with at least a gap greater than the minGap constraint. A *sequence path* is a path from two vertices u and v such as u is a source and v is a sink. Let us note SP_d the set of all sequence paths. In addition, G_d has to satisfy the following two conditions:

1. No sequence path from G_d is included in any other sequence path from G_d.

2. If a candidate sequence is supported by d, then such a candidate is included in a sequence path from G_d.

Given a data sequence d and a minGap constraint, the sequential path problem is to find all the longest paths, i.e. those not included in other ones, by verifying the following conditions:

1. $\forall s1, s2 \in SP_d / s1 \not\subset s2$.
2. $\forall c \in A_s / d \ supports \ c, \ \exists p \in SP_d / p \ supports \ c$ where A_s stands for the set of candidate sequences.
3. $\forall p \in SP_d, \forall c \in A_s / p \ supports \ c, then \ d \ supports \ c$.

The set of all sequences SP_d is thus used to verify the actual support of candidate sequences.

Example 4 *To illustrate, let us consider the graph in Figure 8 representing the application of GTC$_{minGap}$ to the database depicted in Figure 7. Let us assume that the minGap value was set to 2. According to the time constraint, the set of all sequence paths, SP_d, is the following: SP_d=$\{<$ (1) (2) (6) $>, <$ (1) (3 4) (6) $>, <$ (1) (5) $>\}$.*
From this set, the next step of the Main algorithm is to verify these sequences into the candidate tree structure without handling time constraints anymore □

Client	Date	Item
C1	01/04/2000	1
C1	05/04/2000	2
C1	06/04/2000	3 4
C1	07/04/2000	5
C1	09/04/2000	6

Figure 7. An example database

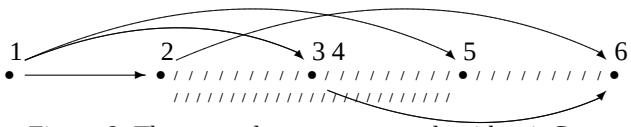

Figure 8. The example sequence graph with minGap = 2

We now describe how the sequence graph is built by the GTC$_{minGap}$ algorithm. Auxiliary data structure can be used to accomplish this task. With each itemset v, the itemsets occurring before v are stored in a sorted array, $v.isPrec$ of size $|E|$. The array is a vector of boolean where 1 stands for an itemset occurring before v. The algorithm operates by performing, for each itemset, the following two sub-steps:

1. **Propagation phase**: the main idea is to retrieve the first itemset u by verifying $(u.date() - v.date() > minGap)$[2], i.e. the first itemset for which the minGap constraint holds, in order to build the edge (u, v). Then for each itemset z such as $(z.date() - y.date() > minGap)$, the algorithm updates $z.isPrec[x]$ indicating that v will reach z traversing the itemset u.
2. **"gap-jumping" phase**: its objective is to yield the set of edges not provided by the previous phase. Such edges (v, t) are defined as follows $(t.date() - x.date() > minGap)$ and $t.isPrec[x] \neq 1$.

Once the GTC$_{minGap}$ has been applied to a data sequence d, the set of all sequences, SP_d, for counting the support for candidate sequences is provided by navigating through the graph of all sequence paths.

The following theorem guarantees that, when applying GTC$_{minGap}$, we are provided with a set of data sequences where the minGap constraint holds and where each yielded data sequence cannot be a sub-sequence of another one.

Theorem 1 *The GTC$_{minGap}$ algorithm provides all the longest-paths verifying minGap.*

First, we prove that for each $p, p' \in SP_d$, $p \not\subset p'$. Next we show that for each candidate sequence c supported by d, a sequence path in G supporting c is found.
Let us assume two sequence paths, $s1, s2 \in SP_d$ such as $s1 \subset s2$. That is to say that the subgraph depicted in Figure 9 is included in G. In other words, there is a path $(a, \ldots, c)$ of length ≥ 2 and an edge (a, c). If such a path (a, c) exists, we have $c.isPrec[a] = 1$. Indeed we can have a path of length ≥ 1 from a to b either by an edge (a, b) or by a path $(a, \ldots, b)$. In the former case, $c.isPrec[a]$ is updated by the statement $c.isPrec[a] \leftarrow 1$, otherwise there is a vertex a' in $(a, \ldots, b)$ such as (a, a') is included in the path. In such a case $c.isPrec[a] \leftarrow 1$ has already occurred when building the edge (a, a'). Then, after building the path $(a, \ldots, b, \ldots, c)$ we have $c.isPrec[a] = 1$ and the edge (a, c) is not built. Clearly the sub-graph depicted in Figure 9 cannot be obtained after GTC$_{minGap}$.
Finally we demonstrate that if a candidate sequence c is supported by d, there is a sequence path in SP_d supporting c. In other words, we want to demonstrate that GTC$_{minGap}$ provides all the longest paths satisfying the minGap constraint. The data sequence d is progressively browsed starting with its first item. Then if an itemset x is embedded in a path satisfying the minGap constraint it is included in SP_d. We

2 where $x.date()$ stands for the transaction time of the itemset x.

have previously noticed that all vertices are included into a path and for each $p, p' \in SP_d$, $p \not\subset p'$. Furthermore if two paths $(x, \ldots, y)(y', \ldots, z)$ can be merged, the edge (y, y') is built when browsing the itemset y.

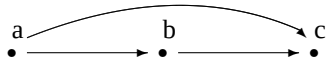

Figure 9. Minimal inclusion schema

5.2. GTC$_{ws}$ Algorithm: solution for minGap and windowSize

In this section, we explain how the algorithm GTC$_{ws}$ provides an optimal solution to the problem of handling minGap and windowSize. As we have already noticed in Section 4, the problem of handling windowSize is much more complicated than handling minGap since the number of included sequences is much greater when considering such a constraint.

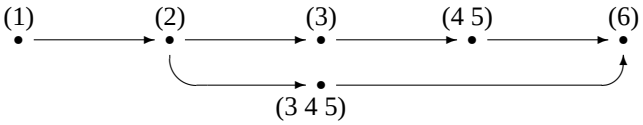

Figure 10. A sequence graph obtained when considering windowSize

To take into account the windowSize constraint we extend the GTC$_{minGap}$ algorithm by generating coherent combinations of windowSize at the beginning of algorithm and, once the graph respecting minGap is obtained, inclusions are detected. The result of this handling is illustrated by Figure 10, which represents the sequence graph of the database given in Figure 5 when windowSize=5 and minGap=1.

Due to lack of space, we no not provide the algorithm but we give an overview of the method.

To yield the set of all windowSize combinations, each vertex x of the graph is progressively browsed and the algorithm determines which vertex can possibly be merged with x. In other words, when navigating through the graph, if a vertex y is such that $y.date() - x.date() < windowSize$, then x and y can be "merged" into the same transaction.
The structure described above is thus extended to handle such an operation. Each itemset, in the new structure, is provided by both the begin transaction date and the end transaction date. These dates are obtained by using the $v.begin()$ and $v.end()$ functions.

Definition 3 *An itemset i is included in another itemset j if and only if the following two conditions are satisfied:*

- $i.begin() \geq j.begin()$,
- $i.end() \leq j.end()$.

Once the graph satisfying minGap is obtained, the algorithm detects inclusions in the following way: for each node x, the set of all its successors $x.next$ must be exhibited. For each node y in $x.next$, if $y \subset z, z \in x.next$ and $y.next \subseteq z.next$ then the node y can be pruned out from the graph.

The following theorem guarantees that, when applying GTC$_{ws}$, we are provided with a set of data sequences where the minGap and windowSize constraints hold and that each yielded data sequence cannot be a sub-sequence of another one.

Theorem 2 *The GTC$_{ws}$ algorithm provides all the longest paths verifying minGap and windowSize.*

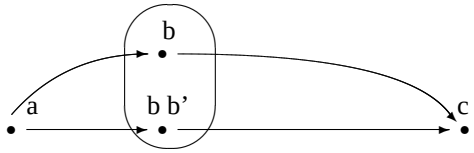

Figure 11. An included path example

Theorem 1 shows that we do not have included data sequences when considering minGap. Let us now examine the windowSize constraint in detail. Let us consider two sequence paths s_1 and s_2 in G_d such that $s_1 \subset s_2$. Figure 11 illustrates such an inclusion. In the last phase of the GTC$_{ws}$ algorithm, we examine for each vertex x of the graph, the set of its successors by using the $x.next$ function. So, for each vertex y in $x.next$, if $y \subset z$, $z \in x.next$ and $y.next \subseteq z.next$, the vertex y is pruned out from the graph. So, by construction, s_1 cannot be in the graph.

5.3. Solution for all time constraints

In order to handle the maxGap constraint in the GTC algorithm, we have to consider the itemset time-stamps into

the graph previously obtained by GTC$_{ws}$. Let us remember that, according to maxGap, a candidate sequence c is not included in a data sequence s if there exist two consecutive itemsets in c such that the gap between the transaction time of the first itemset (called l_{i-1} in Definition 2) and the transaction time of the second itemset (called u_i in Definition 2) in s is greater than maxGap. According to this definition, when comparing candidates with a data sequence, we must find in a graph itemset, the time-stamp for each item since, due to windowSize, items can be gathered together. In order to verify maxGap, the transaction time of the sub-itemset corresponding to the included itemset into the graph, must verify the maxGap delay from the preceding itemset as well as for the following itemset.

To illustrate, let us consider the database depicted in Figure 12. Let us now consider, in Figure 13, the sequence graph obtained from the GTC$_{ws}$ algorithm when windowSize was set to 1 and minGap was set to 0. In order to determine if the candidate data sequence $<$ (2) (4 5) (6) $>$ is included into the graph, we have to examine the gap between item 2 and item 5 as well as between item 4 and item 6. Nevertheless, the main problem is that, according to windowSize, itemset (3) and itemset $(4\ 5)$ were gathered together into $(3\ 4\ 5)$.We are led to determine the transaction time of each component in the resulting itemset.

Customer	Date	Items
C_1	$01/01/2000$	2
C_1	$03/01/2000$	3
C_1	$04/01/2000$	4 5
C_1	$06/01/2000$	6

Figure 12. A database example

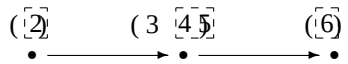

Figure 13. Sequence graph obtained by GTC$_{ws}$

Before presenting how maxGap is taken into account in GTC, let us assume that we are provided with a sequence graph containing information about itemsets satisfying the maxGap constraint. By using such an information the candidate verification can thus be improved as illustrated in the following example.

Example 5 *Let us consider the sequence graph depicted in Figure 13. Let us assume that we are provided with information about reachable vertices into the graph according to maxGap and that maxGap is set to 4 days. Let us now consider how the detection of the inclusion of a candidate sequence within the sequence graph is processed. Candidate itemset (2) and sequence graph itemset (2) are first compared by the algorithm. As the maxGap constraint holds and* $(2) \subseteq (2)$*, the first itemset of the candidate sequence is included in the sequence graph and the process continues. In order to verify the other components of the candidate sequence, we must know what is the next itemset ended by 5 in the sequence graph and verifying the maxGap delay. In fact, when considering the last item of the following itemset, if we want to know if the maxGap constraint holds between the current itemset (2) and the following itemset in the candidate sequence, we have to consider the delay between the current itemset in the graph and the next itemset ended by 5 in this graph. We considered that we are provided with such an information in the graph. This information can thus be taken into account by the algorithm in order to directly reach the following itemset in the sequence graph* $(3\ 4\ 5)$ *and compare it with the next itemset in the candidate sequence* $(4\ 5)$*. Until now, the candidate sequence is included into the sequence graph. Nevertheless, for completeness, we have to find in the graph the next itemset ended by 6 and verifying that the delay between the transaction times of items 4 and 6 is lower than 4 days. This condition occurs with the last itemset in the sequence graph. At the end of the process, we can conclude that* c *is included in the sequence graph of* d *or more precisely that* c *is included in* d*.*
Let us now consider the same example but with a maxGap constraint set to 2. Let us have a closer look at the second iteration. As we considered that we are provided with information about maxGap into the graph, we know that there is no itemset such that it ends in 5 and it satisfies the maxGap constraint with item 2. The process ends by concluding that the candidate sequence is not included into the data sequence and without navigating further through the candidate structure □

Let us now describe how information about itemsets verifying maxGap is taken into account in GTC. Each item in the graph is provided with an array indicating reachable vertices, according to maxGap. Each array value is associated with a list of pointed nodes, which guarantees that the pointed node corresponds to an itemset ending by this value and that the delay between these two items is lower or equal to maxGap. Candidate verification algorithms can thus find candidates included in the graph by using such information embedded in the array. By means of pointed nodes, the maxGap constraint is considered during evaluation of candidate itemset.

6. Experiments

In this section, we present the performance results of our GTC algorithm. The structure used for organizing candidate sequences is a prefix tree structure as in PSP. All experiments were performed on a PC Station with a CPU clock rate at 1.8 GHz, 512M Bytes of main memory, Linux System and a 60G Bytes disk drive.

In order to assess the relative performance of the GTC algorithm and study its scale-up properties, we used different kinds of datasets. Due to lack of space we only report some results obtained with one access log file. It contains about 800K entries corresponding to the requests made during March of 2002 and its size is about 600 M Bytes. There were 1500 distinct URLs referenced in the transactions and 15000 clients.

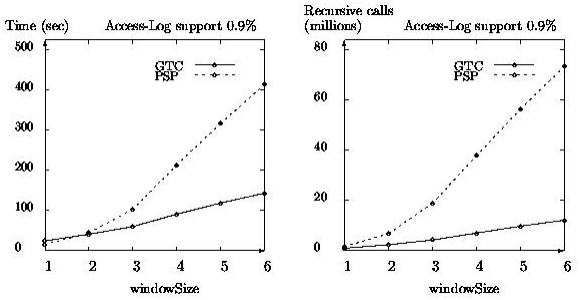

Figure 14. Execution times and recursive calls (minGap=1)

Figure 14 shows experiments conducted on the different datasets using different windowSize ranges to get meaningful response times. minGap was set to 1 and maxGap was set to ∞. Note the minimum support thresholds are adjusted to be as low as possible while retaining reasonable execution times. Figure 14 clearly indicates that the performance gap between the two algorithms increases with increasing windowSize value.The reason is that during the candidate verification, PSP has to determine all combination of the data sequence according to minGap and windowSize constraints. In fact, the more the value of windowSize increases, the more PSP carries out recursive calls in order to apply time constraints to the candidate structure. According to these calls, the PSP algorithm operates a costly backtracking for examining the prefix tree structure.In order to illustrate correlation between the number of recursive calls and the execution times we compared the number of recursive calls needed by PSP and our algorithm and as expected, the number of recursive calls increases with the size of the windowSize parameter.Additional experiments were performed in order to study performance in scale-up databases. they showed that GTC scaled up linearly as the number of transactions is increased.

7. Conclusion

We considered the problem of discovering sequential patterns by handling time constraints. We proposed a new algorithm called GTC based on the fundamental principles of PSP and GSP but in which time constraints are handled in the earlier stage of the algorithm in order to provide significant benefits. Using synthetic datasets as well as real life databases, the performance of GTC was compared against that of PSP. Our experiments showed that GTC performs significantly better than the state-of-the-art approaches since the improvements achieved by GTC over the counting strategy employed by other approaches are two-fold: first, only maximal sequences are generated, and there is no need of an additional phase in order to count candidates included in a data sequence. In order to take advantage of the behavior of the algorithm in the first scans on the database, we are designing a new algorithm called PG-$Hybrid$ using the PSP approach for the two first passes on the database and GTC for the following scans. First experiments are encouraging, even for short frequent sequences.

References

[1] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, Tapei, Taiwan (1995).

[2] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential Pattern Mining Using Bitmap Representation. In *Proocedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada (2002).

[3] F. Masseglia, F. Cathala, and P. Poncelet. The PSP Approach for Mining Sequential Patterns. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98), LNAI, Vol. 1510*, pp. 176–184, Nantes, France (1998).

[4] F. Masseglia, P. Poncelet, and M. Teisseire. Incremental Mining of Sequential Patterns in Large Databases. *Data and Knowledge Engineering*, **46**(1):97–121 (2003).

[5] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and MC. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *Proceedings of 17th International Conference on Data Engineering (ICDE'01)* (2001).

[6] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, pp. 3–17, Avignon, France (1996).