Efficient Mining of Sequential Patterns with Time Constraints: Reducing the Combinations

Florent Masseglia, Pascal Poncelet, Maguelonne Teisseire

▶ **To cite this version:**

Florent Masseglia, Pascal Poncelet, Maguelonne Teisseire. Efficient Mining of Sequential Patterns with Time Constraints: Reducing the Combinations. Expert Systems with Applications, 2009, 36 (2), pp.2677-2690. 10.1016/j.eswa.2008.01.021 . lirmm-00272632

HAL Id: lirmm-00272632

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00272632

Submitted on 5 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Mining Sequential Patterns with Time Constraints: Reducing the Combinations

F. Masseglia[(1)] P. Poncelet[(2)] M. Teisseire[(3)]

[(1)]INRIA Sophia Antipolis - AxIS Project, 2004 route des Lucioles - BP 93, 06902 Sophia Antipolis, France
email: Florent.Masseglia@sophia.inria.fr

[(2)]EMA-LGI2P/Site EERIE, Parc Scientifique Georges Besse, 30035 Nîmes Cedex 1, France
email: Pascal.Poncelet@ema.fr

[(3)]LIRMM UMR CNRS 5506, 161 Rue Ada, 34392 Montpellier Cedex 5, France
email: teisseire@lirmm.fr

Abstract

In this paper we consider the problem of discovering sequential patterns by handling time constraints as defined in the GSP algorithm. While sequential patterns could be seen as temporal relationships between facts embedded in the database where considered facts are merely characteristics of individuals or observations of individual behavior, generalized sequential patterns aim at providing the end user with a more flexible handling of the transactions embedded in the database. We thus 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 data mining process can be highly beneficial. One of the most significant new feature of our approach is that handling of time constraint can be easily taken into account in traditional levelwise approaches since it is carried out prior to and separately from the counting step of a data sequence. Our test shows that the proposed algorithm performs significantly faster than a state-of-the-art sequence mining algorithm.

Keywords: Time constraints, sequential patterns, levelwise algorithms.

1 Introduction

The explosive growth in stored data has enlarged the interest in the automatic transformation of the vast amount of data into useful information and knowledge. Since its introduction of the Apriori algorithm [AIS93] more than a decade ago, the problem of mining patterns is becoming a very active research area and efficient techniques have been widely applied to problems either in industry or

science. For instance, in [AS95], the problem has been refined considering a database storing behavioral facts which occur over time to individuals of the studied population. Thus facts are provided with a time stamp. The concept of *sequential pattern* is introduced to capture typical behaviors over time, *i.e.* behaviors sufficiently repeated by individuals to be relevant for the decision maker. Sequential pattern mining is applicable in a wide range of applications since many types of data are in a time-related format and many interesting sequential patterns algorithms have thus been provided in the past years (*e.g.* [AGYF02, PHMA+01, WH04, KPWD03]). For exemple, from a customer purchase database a sequential pattern can be used to develop marketing and product strategies. By way of a Web Log analysis, data patterns are very useful to better structure a company's Web site for providing easier access to the most popular links. We also can notice telecommunication network, alarm databases, intrusion detection, DNA sequences, etc. However, for effectiveness consideration but also in order to efficiently aid decision making, constraints become more and more essential in many applications. The GSP approach proposed in [SA96] extends previous proposal by handling time constraints. They allow time constraints between elements of the sequential patterns, and also allow all the items in an element to be present in a sliding time window rather than at a single point in time. Although this sliding time windows can be very useful for real-world applications, proposed approaches in the literature mainly address a subset of constraints (*e.g.* [GNPP06, OPS04, MPT04, Zak00, SK02, LRBE03]). Nevertheless, handling such time constraints is far away for trivial since lot of expensive operations have to be performed in order to examine if constraints are verified on sequences.

It is obvious that additional constraints can be verified in a post-processing step, after all patterns exceeding a given minimum support threshold have been discovered. Nevertheless, such a solution cannot be considered satisfactory since users providing advanced pattern selection criteria may expect that the data mining system will exploit them in the mining process to improve performance. It has been shown that pushing constraints deep into the mining process can reduce processing time by more than an order of magnitude [GRS02, WMZ02]. Nevertheless very little work concerning temporal constraint-driven sequential pattern discovery has been done so far. The problems we address are the following: (i) is it possible to enhance traditional levelwise algorithms to handle time constraint? (ii) is it possible to consider time constraints directly in the mining process rather than a post-processing step?

In this paper, we extend a previous work [MPT04] and then propose a new efficient algorithm, called GTC (*Graph for Time Constraints*), for mining generalized sequential patterns in large databases. Our approach is defined in order both to minimize the rewriting of previously existing levelwise

algorithms and to take advantage of already defined structures which are used in these algorithms. The main new feature of GTC is that time constraints are taken into account during the mining process and 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. Section 3 gives an overview of the existing work in the area of mining sequential patterns with time constraints. Section 4 goes deeper into our motivations. 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. Section 7 concludes the paper.

2 From Sequential Patterns to Generalized Sequential Patterns

This section broadly summarizes the formal description of the problem introduced in [AS95] and extended in [SA96]. We first formulate the concept of sequences, and then look at time constraints in detail.

Let DB be a database of customer transactions where each transaction T consists of:

1. a customer identifier, denoted by C_{id};
2. a transaction time, denoted by *time*,
3. a set of items (called *itemset*) involved in the transaction, denoted by *it*.

Definition 1 (Sequence) *Let $I = \{i_1, i_2, ..., i_m\}$ be a finite 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 $\langle it_1 it_2 ... it_n \rangle$ where it_j, $j \in 1...n$, is an itemset. A k-sequence is a sequence of k-items (or of length k).*

Definition 2 (Inclusion of a Sequence) *A sequence $S' = \langle s_1 s_2 ... s_n \rangle$ is a sub-sequence of another sequence $S = \langle s'_1 s'_2 ... s'_m \rangle$, denoted $S' \preceq S$, if $\exists\ i_1 < i_2 < ... < i_n$ 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 the following sequence: $S = \langle (6)\ (2)\ (3)\ (8\ 9) \rangle$. This means that, apart from 8 and 9 which were purchased together (*i.e.* during a common transaction), the items in the sequence were bought separately. The sequence $S' = \langle\ (2)\ (8)\ \rangle$ is a sub-sequence of S, *i.e.* $S' \preceq S$,

because (2) $\subseteq$ (2) and (8)$\subseteq$ (8 9). However $\langle$ (8) (9) $\rangle$ 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 (*i.e.* to avoid capturing sequences that occur too infrequently for it to be worth attempting to learn lessons from such particular cases). Performing such a task requires providing any data sub-sequence S in DB with a support value giving its number of occurrences in DB.

Definition 3 (Support) *A sequence transaction database is a set of sequences The support of a sequence S in a transaction database DB, denoted $Support(S, DB)$, is defined as: $Support(S, DB) = |\{C \in DB | S \preceq C\}|$. The frequency of S in DB is $\frac{Support(S,DB)}{|DB|}$.*

Definition 4 (Frequent Sequential Pattern Problem) *Given a user-defined minimal support threshold, denoted σ, the problem of sequential pattern mining is the extraction of sequences S in DB such that $Support(S, DB) \geq \sigma$. Such sequences are called frequent.*

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* have been proposed in [SA96]. 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, sub-sequences 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, insofar as the end user is provided with the following advantages:

- To group together itemsets when their transaction times are rather close via the *windowSize* constraint. For example, it does not matter if itemsets in a sequential pattern were present in two different transactions, as long as the transaction times of those transactions are within some small window.
- To regard itemsets as too close or distant to appear in the same frequent sequence with the *minGap* or *maxGap* constraint. For example, the end user probably does not care if a client bought 'Star Wars Episode IV' followed by 'Star Wars Episode III' two years later.

We now recall the definition [SA96] of frequent sequences when handling time constraints:

Definition 5 (Generalized Sequences) *Given a user-specified minimum time gap (minGap), a maximum time gap (maxGap) and a time window size (windowSize), a data sequence* $d = \langle d_1 ... d_m \rangle$ *is said to support a sequence* $S = \langle s_1 ... s_n \rangle$ *if there exist integers* $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ *such that:*

1. s_i *is contained in* $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$;
2. *transaction-time* (d_{u_i}) *- transaction-time* $(d_{l_i}) \leq ws$, $1 \leq i \leq n$;
3. *transaction-time* (d_{l_i}) *- transaction-time* $(d_{u_{i-1}}) > minGap$, $2 \leq i \leq n$;
4. *transaction-time* (d_{u_i}) *- transaction-time* $(d_{l_{i-1}}) \leq maxGap$, $2 \leq i \leq n$.

Definition 6 (Frequent Generalized Sequential Pattern Problem) *Given a user-defined minimal support threshold, denoted* σ*, user-specified minGap and maxGap constraints, and a user-specified sliding windowSize, the problem of generalized sequential pattern mining is the extraction of sequences* S *in* DB *such that* $Support(S, DB) \geq \sigma$.

Example 2 In order to illustrate how time constraint are handled, let us consider the following data sequence describing the purchased items for a customer:

$$d = \langle\ (1)^1\ (2\ 3)^2\ (4)^3\ (5\ 6)^4\ (7)^5\ \rangle$$

where each itemset is stamped by its access day. For instance, $(5\ 6)^4$ means that the items 5 and 6 were purchased together at time 4.

Let us now consider the sequence d_1=$\langle\ (1\ 2\ 3\ 4)\ (5\ 6\ 7)\ \rangle$ and time constraints specified such as $windowSize$=3, $minGap$=0 and $maxGap$=5. The sequence d_1 is considered as included in d for the two following reasons:

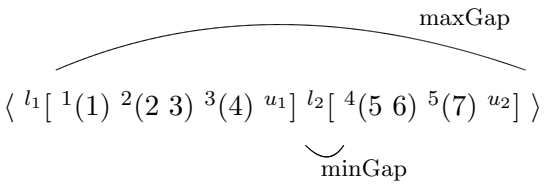

Figure 1: Illustration of the time constraints

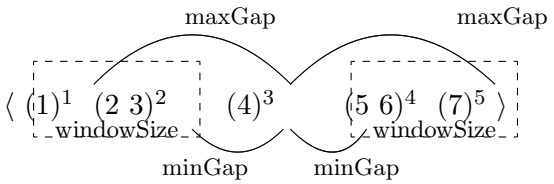

Figure 2: Illustration of the time constraints

1. the *windowSize* parameter makes it possible to group together both the itemsets (1) (2 3) with (4), and the itemsets (5 6) with (7) in order to obtain the itemsets (1 2 3 4) and (5 6 7);
2. the *minGap* constraint between the itemsets (4) and (5 6) holds.

Considering the integers l_i and u_i in Definition 5, 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. In a similar way, the sequence d_2=⟨ (1 2 3) (4) (5 6 7) ⟩ 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 $u_3 = 5$ (see Figure 2) is included in d.

Nevertheless, the two following sequences d_3=⟨ (1 2 3 4) (7) ⟩ and d_4 = ⟨ (1 2 3) (6 7) ⟩, with *windowSize*=1, *minGap*=3 and *maxGap*=4 are not included in d. Concerning the former, the windowSize is not large enough to group together the itemsets (1) (2 3) and (4). For the latter, the only possibility for yielding both (1 2 3) and (6 7) is to take into account ws for achieving the following grouped itemsets [(1) (2 3)] and [(5 6) (7)]. *maxGap* is respected since [(1) (2 3)] and [(5 6) (7)] 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.

3 Related Work

The task of discovering all the frequent sequences is quite challenging since the search space is extremely large: let $\langle s_1\ s_2\ ...s_m\rangle$ be a provided sequence and $n_i = |s_j|$ cardinality of an itemset. Then the search space, *i.e.* the set of all potentially frequent sequences is $2^{n_1+n_2+...n_m}$. From the definition presented so far, different approaches were proposed to mine sequential patterns (*e.g.* GSP [SA96], PSP [MCP98], SPADE [Zak01], FREESPAN [HPMa+00], PREFIXSPAN [PHMA+01], SPAM [AGYF02], APPROXMAP [KPWD03], BIDE [WH04], ...). In this section we mainly focus on approaches used for mining sequential patterns by handling time constraints.

We shall now briefly review the GSP algorithm principle [SA96] which was the first Apriori-based approach (or levelwise approach) [AIS93]. To build up candidates and frequent sequences, the GSP algorithm makes multiple passes over the database. The first step aims at computing the support of each item in the database. When this step has been completed, the frequent items (*i.e.* those that satisfy the minimum support) have been discovered. They are considered as frequent 1-sequences (sequences having a single itemset, itself a singleton). The set of candidate 2-sequences is built up according to the following assumption: candidate 2-sequences could be any couple of frequent items, whether embedded in the same transaction or not. Frequent 2-sequences are determined by counting the support. From this point, candidate k-sequences are generated from frequent (k-1)-sequences obtained in pass-(k-1). The main idea of candidate generation is to retrieve, from among (k-1)-sequences, 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 supports for these candidates are then computed and those with minimum support become frequent sequences. The process iterates until no more candidate sequences are formed. 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. Let us notice that during this operation the minGap condition applies in order to

skip itemsets that are too close to their precedent. And while selecting items, the sliding window is used for resizing transaction cutting. Backward phases are required as soon as the maxGap condition no longer holds. In such a case, it is necessary to discard all the items for which the maxGap constraint is violated and to resume browsing the sequence starting with the earliest item satisfying the maxGap condition. Another method based on the Generating-Pruning principle is PSP [MCP98] where a prefix-tree based approach is used and time constraints are handled when enumerating frequent sequences, *i.e.* when counting candidate sequences. This prefix-tree based structure has been proved more efficient than the hash tree used in GSP. Even if these approaches are efficient they suffer a serious setback due to the number of backtracking operations. The CSPADE [Zak00] extended the SPADE algorithm to incorporate minGap, maxGap and windowSize but the *windowSize* refers to the window of occurrence of the whole sequence rather than a sliding time window. More recently, the DELISP algorithm [LL05] has been proposed for mining sequential patterns with time constraints. DELISP is based on the mining scheme of PREFIXSPAN [PHMA+01]. Actually, the original database will be divided into multiple subsets for each prefix of a potential sequential pattern. During the writing of the subsets, DELISP reduces the size of the projected databases by bounded and windowed projection techniques. DELISP captures the advantages of PREFIXSPAN and offers its own advantages: (i) no candidate generation (since prefix growing methods do not generate candidates); (ii) focused search (the mining schema between generating-pruning method such as GSP and prefix-growing methods such as PREFIXSPAN is very different); (iii) constraint integration (in particular, DELISP benefits from maxGap since some elements can be discarded when they are inaccessible). The pattern-growth methodology is also used in the CTSP algorithm [LHC06] where the authors extract closed sequential patterns with time constraints. However, such techniques are restricted to prefix-growth algorithms while our goal is to provide a generic solution for levelwise algorithms.

4 Motivations

Let us consider the following customer data sequence $d_1 = \langle\ (1)^1\ (2)^3\ (3)^4\ (4)^7\ (5)^{17}\ \rangle$. Let us assume that windowSize = 1 and minGap=1. Upon reading the customer transaction, a sequential pattern algorithm 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
$\langle$ (1) $\rangle$	$\langle$ (2) (5) $\rangle\bullet$
...	...
$\langle$ (5) $\rangle$	$\langle$ (1) (3) (5) $\rangle\bullet$
...	...
$\langle$ (1) (2)$\rangle$	$\langle$ (1) (2 3) (5) $\rangle\bullet$
...	...
$\langle$ (3) (4)$\rangle$	$\langle$ (1) (2) (4) (5)$\rangle\bullet$
...	$\langle$ (1) (3) (4) (5)$\rangle\bullet$
$\langle$ (1) (2) (3) (4) (5) $\rangle$	$\langle$ (1) (2 3) (4) (5)$\rangle\circ$

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

Let us now take 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 following data sequence $d_2 = \langle\ (1)^1\ (2)^7\ (3)^{13}\ (4)^{17}\ (5)^{18}\ (6)^{24}\ \rangle$. When windowSize=5 and minGap=1, a traditional algorithm has to test the following sequences into the candidate tree structure (we only report sequences when windowSize is applied):

$\langle$ (1) (2) (3) (5) (6) $\rangle\bullet$

$\langle$ (1) (2) (3) (4) (6) $\rangle\bullet$

$\langle$ (1) (2) (3) (4 5) (6) $\rangle\circ$

$\langle$ (1) (2) (3 4) (6) $\rangle\cdot$

$\langle$ (1) (2) (3 4 5) (6) $\rangle\odot$

Here also, we notice that the sequences marked by a $\bullet$ (resp. $\cdot$) are included in the sequence marked by a $\circ$ (resp. $\odot$).

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 a relevant set of sequences to be tested for a data sequence. By precalculating this set, we can reduce the time spent analyzing a data sequence when verifying candidate sequences stored in the tree, in the following two ways:

- 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 in traditional sequential pattern algorithms computed on the fly.
- 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 (see Algorithm 1) takes up all the fundamental principles of levelwise algorithms. 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.

Data: A frequency threshold σ, *minGap*, *maxGap* and *windowSize*, and a database DB of data sequences.
Result: The collection L of frequent sequences, k the maximal frequent length.

$L_0 = 0$; $k = 1$;
$C_1 = \{\{\langle i \rangle\}/i \in I\}$;
// all 1-frequent sequences
while $C_k \neq \emptyset$ **do**
  foreach $d \in DB$ **do**
    $\mathcal{G}$ =GTC(d);
    // $\mathcal{G}$ stands for the graph representing
    // all Time Constraints combinations of d.
    $countsupport(C_k, \sigma, \mathcal{G})$;
  $L_k = \{c \in C_k / Support(c) > \sigma\}$;
  $C_k + 1 = CandidateGeneration(L_k)$;
  $k = k + 1$;
return $L = \bigcup_{j=0}^{k} L_j$;

Algorithm 1: The TCLW Algorithm

The main new feature of the TCLW (*Time Constraints LevelWise*) algorithm which distinguishes it from traditional levelwise algorithms is that, thanks to the GTC function, 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 (sequence d_2), only $\langle$ (1) (2) (3) (4 5) (6) $\rangle$ and $\langle$ (1) (2) (3 4 5) (6) $\rangle$ are exhibited by GTC. Then the TCLW 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 describe the GTC$_{minGap}$ algorithm which provides an optimal solution to the problem of handling the minGap constraint.

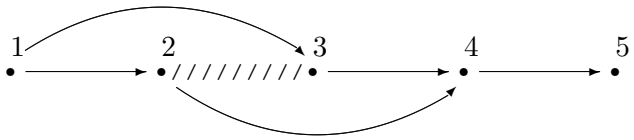

Figure 3: A data sequence representation

To illustrate, let us consider Figure 3, which represents d_1, given in Section 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 ($\langle$ (1) (2) (4) (5) $\rangle$ and $\langle$ (1) (3) (4) (5) $\rangle$) 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.

Definition 7 (Sequence Graph) *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.*

Definition 8 (The Sequential Path Problem) *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 3 *To illustrate, let us consider the graph in Figure 4 representing the application of* GTC_{minGap} *to the data sequence* $\langle (1)^1\ (2)^5\ (3\ 4)^6\ (5)^7\ (6)^9 \rangle$. *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=$\{\langle\ (1)\ (2)\ (6)\ \rangle, \langle\ (1)\ (3\ 4)\ (6)\ \rangle, \langle\ (1)\ (5)\ \rangle\}$. *From this set, the next step of the* TCLW *algorithm is to verify these sequences into the candidate tree structure without handling time constraints anymore.*

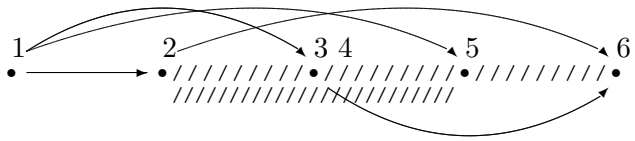

Figure 4: The example sequence graph with minGap = 2

We now describe how the sequence graph is built by the GTC_{minGap} algorithm (see Algorithm 2). 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 $(u.date() - v.date() > minGap)$[1] (*i.e.* the first itemset for which the minGap constraint holds) in order

[1] where $x.date()$ stands for the transaction time of the itemset x.

Data: A data sequence d.
Result: The sequence graph $G_d(V,E)$.

foreach *itemset* $i \in d$ **do**
  $V = V \cup \{i\}$;
foreach $x \in V$ **do**
  //Propagation phase
  $y = x$;
  while $y.date() - x.date() < minGap$ **do** $y++$;
  $E = E \cup \{x, y\}$;
  $i_p = \{i \in V / i.date() - y.date() > minGap\}$;
  foreach $z \in i_p$ **do**
    $z.isPrec[x] = 1$;
  // Gap-jumping phase
  $j_p = \{j \in V / j.date() > y.date()$ *and* $j.isPrec[x] = 0\}$;
  foreach $t \in j_p$ **do**
    $E = E \cup \{x, t\}$;

Algorithm 2: GTC$_{minGap}$, solution for minGap

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.

Example 4 *In order to illustrate how the sequence paths are provided, let us consider sequence d_2, given in Section 4, when minGap is set to 1. First, the set V standing for itemsets embedded in the data sequence is created. Then, for each itemset in V, propagation and gap jumping phase are applied. The result is depicted in Figure 5. Let us now consider each phase in detail.*

- *$x = 1$: propagation phase. The algorithm is led to find the first itemset u such as $u.date() - v.date() > minGap$. For the first itemset (1), we would first find (2). Next itemsets (4) and (5) can be reached from (2) and the minGap constraint is verified. Their associated isPrec array is updated ($4.isPrec[1] = 1$ and $5.isPrec[1] = 1$) in order to mark that these itemsets are reachable from (1) but with a longer path than (1,4) or (1,5).*
  gap jumping phase: for each itemset t, successor of (2), satisfying both the minGap constraint and $t.isPrec[1] = 0$, the algorithm builds the edge (x, t). In our case, we have a new edge from

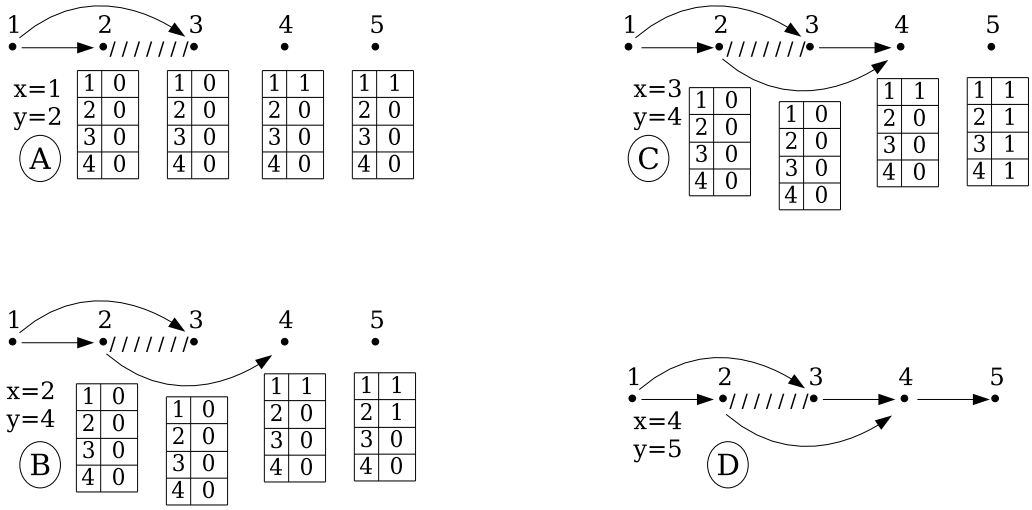

Figure 5: Building of a sequence graph

(1) to (3).

- $x = 2$*: propagation phase. For the second itemset of the data sequence, we would find (4) then the edge (2,4) is built. As (5) follows the itemset (4), its array is updated. The gap jumping phase is not applied since there is no itemset satisfying the minGap constraint anymore.*
- $x = 3$*: propagation phase. The edge* $(3, 4)$ *is built since (4) is the first itemset following (3). Next* $5.isPrec[3]$ *is updated because (5) follows the itemset (2). As there is no itemset satisfying the minGap constraint, the next phase is not considered any more.*
- $x = 4$*: propagation phase. The edge (4,5) is built and the process completes since (5) is not followed by other itemsets.*

From the database example, the set of longest paths verifying time constraints is thus obtained by navigating through the graph. Then the minimal number of sequence paths for counting the actual support for candidate sequences is reduced to two: $\langle(1)\ (2)\ (4)\ (5)\rangle$ *and* $\langle(1)\ (3)\ (4)\ (5)\rangle$*. From these two sequences, a navigation through the structure used to manage candidate can be performed without any backtracking.*

The following theorem guarantees that, when applying $\textsc{Gtc}_{minGap}$, 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.*

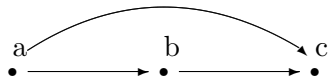

Figure 6: Minimal inclusion schema

Proof 1 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 6 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 6 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 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.

Theorem 2 *The time complexity of Algorithm* GTC_{minGap} *is* $O(n^2)$.

In the worst case the minGap constraint is set to 0. In fact, in this case, there is no use applying the GTC_{minGap} algorithm. Nevertheless, we provide an analysis of time complexity because such analysis will be necessary when considering the windowSize constraint.

Proof 2 If minGap=0, the graph is progressively browsed and for each itemset x, GTC_{minGap} is then led to test the gap between x ant its successive itemset y. From y, GTC_{minGap} has to test its successive itemset twice:

(i) the propagation phase is executed for y;

(ii) the gap jumping phase is performed for x. The cost of such an operation can be expressed by $\sum_{k=1}^{n-1} 2k - 1$.

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

In this section, we describe the algorithm GTC$_{ws}$ which 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.

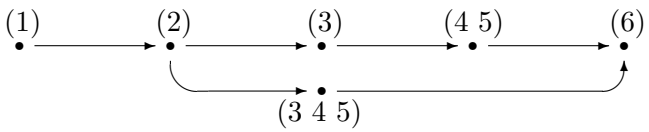

Figure 7: A sequence graph obtained when considering windowSize

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 7, which represents the sequence graph of sequence d_1, given in Section 4, when windowSize=5 and minGap=1.

The GTC$_{ws}$ method, providing solution for minGap and windowSize is defined in Algorithm 3. 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 9 (Inclusion of Itemsets) *An itemset i is included in another itemset j if and only if the following two conditions are satisfied: $i.begin() \geq j.begin()$ and $i.end() \leq j.end()$.*

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

function GTC$_{ws}$
Data: A data sequence d.
Result: The sequence graph G$_d$(V,E).

foreach *itemset* $i \in d$ **do**
  $V = V \cup \{i\}$;
$addWindowSize(V)$; //add transactions to V when considering windowSize
foreach $x \in V$ **do**
  //Propagation phase
  $y = x$;
  while $y.date() - x.date() < minGap$ **do** $y++$;
  $E = E \cup \{x, y\}$;
  $i_p = \{i \in V / i.date() - y.date() > minGap\}$;
  foreach $z \in i_p$ **do**
    $z.isPrec[x] = 1$;
  // Gap-jumping phase
  $j_p = \{j \in V / j.date() > y.date()$ *and* $j.isPrec[x] = 0\}$;
  foreach $t \in j_p$ **do**
    $E = E \cup \{x, t\}$;
$pruneIncluded(V, E)$; // prune vertices from included sequences
end function GTC$_{ws}$

Algorithm 3: GTC$_{ws}$ algorithm for minGap and windowSize

$y \subset z, z \in x.next$ and $y.next \subseteq z.next$ then the node y can be pruned out from the graph.

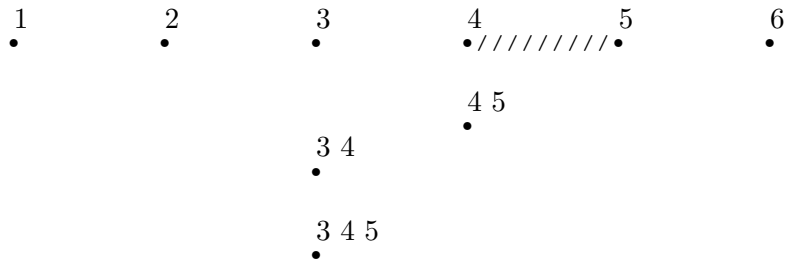

Figure 8: A sequence graph after the first phase

Example 5 *Let us consider sequence d_2 in Section 4, the graph resulting from the first phase of the algorithm is represented by Figure 8. Indeed, windowSize being fixed at 4, the items* 3, 4 *and* 5 *can be merged together into the same transaction. However, we have to consider that either* $\langle(3)\ (4\ 5)\rangle$ *or* $\langle(3\ 4)\ (5)\rangle$ *can be a part of a candidate sequence and thus must be tested. This is why the algorithm builds the vertices corresponding to the itemsets* (3 4), (4 5) *and* (3 4 5). *The graph resulting of the second phase is depicted in Figure 9.*

The method used to detect inclusion is illustrated in Figure 10. We note that the sequences $\langle$ *(3) (4)*

```
procedure addWindowSize
Data: The set of all vertices V sorted with begin transaction date as the major key and end
      transaction date as the minor key.
a = V.first();
while a ≠ V.last() do
    b = V.succ(a);
    while b.end() − a.begin() < windowSize do
        i = group(a, b);
        V.insert(i, b); // i is inserted before b in V
        a = V.succ(a);
        b = V.succ(a);
    a = V.succ(a);
end procedure addWindowSize
```

Algorithm 4: WindowSize combination for a data sequence

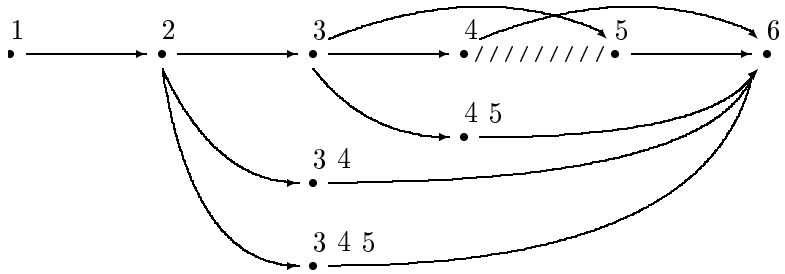

Figure 9: A sequence graph after the second phase

(6) ⟩ and ⟨ (3) (5) (6) ⟩ are included in the sequence ⟨ (3) (4 5) (6) ⟩. Indeed 3.*next*={*(4), (5), (4 5)* }, 4.*next* = 5.*next* = (4 5).*next* = 6 *and as* 4 ⊂ (4 5) *and* 5 ⊂ (4 5) *vertices 4 and 5 can be removed. On the other hand,* 2.*next* = {(3), (3 4), (3 4 5)} *but* 3.*next* ⊄ (3 4 5).*next thus vertex 3 is not removed. The graph used during the checking of the candidates is illustrated by Figure 7.*

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.

```
procedure pruneIncluded
Data: The sequence graph G_d(V, E)
foreach x ∈ V do
    foreach y ∈ x.next do
        foreach z ∈ x.next do
            if y ⊂ z and y.next ⊆ z.next then prune(y);
end procedure pruneIncluded
```

Algorithm 5: Discovering and Pruning included data sequences

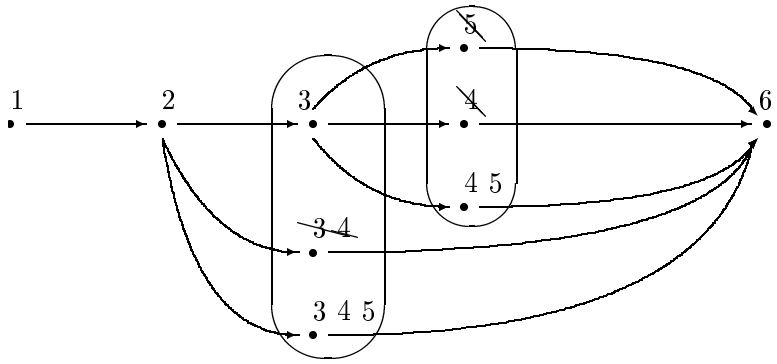

Figure 10: Inclusion discovery method

Theorem 3 *The* GTC$_{ws}$ *algorithm provides all the longest paths verifying minGap and windowSize.*

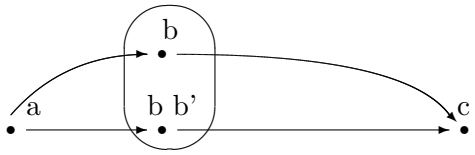

Figure 11: An included path example

Proof 3 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 GTC Algorithm: 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 5) and the transaction time of the second itemset (called u_i in Definition 5) 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.

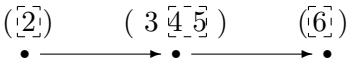

Figure 12: Sequence graph obtained by GTC$_{ws}$

To illustrate, let us consider the following data sequence: $\langle(2)^1\ (3)^3\ (4\ 5)^4\ (6)^6\rangle$. Let us now consider, in Figure 12, 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 $\langle$ (2) (4 5) (6) $\rangle$ 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.

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 6 *Let us consider the sequence graph depicted in Figure 12. 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. The GTC algorithm is defined in algorithm 6.

```
function GTC
Input: a data sequence d
Output: the sequence graph G_d(V,E)
G_d(V,E)=GTC_ws(d);
foreach item i ∈G_d(V,E) do
    foreach item j ∈G_d(V,E) do
        if J.isPrec[i]=1 OR j ∈ i.next then
            addMax(i,j); // Adds j to the pointer list for the j valued cell associated to i
end function GTC
```

Algorithm 6: GTC algorithm for minGap, windowSize and maxGap

Example 7 *To illustrate, let us consider the sequence graph obtained in Example 5 from sequence* d_1*, given in Section 4. Let us assume that maxGap is set to 2. According to the previous discussion, the graph resulting is depicted in Figure 13. Let us now examine the itemset* (2). *According to maxGap, the vertex* (3 4 5) *is reachable from* (2). *Nevertheless, as the maxGap constraint does not hold between item* 3 *and the following item* 6, *there is no reachable itemset from* 3 *and the associated value is the empty set. On the other hand, according to maxGap, the vertex* (3) *is reachable from* (2). *From this*

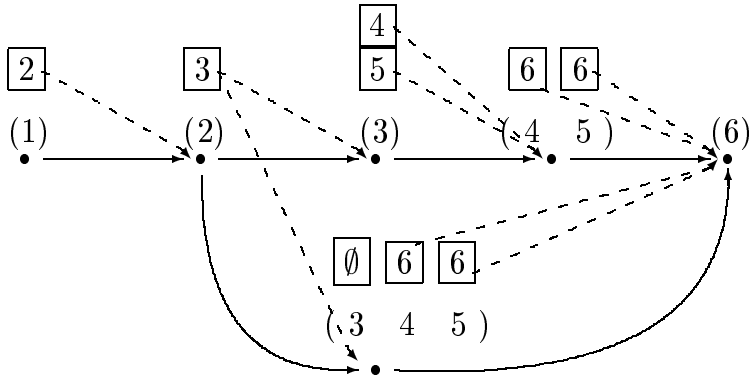

Figure 13: Sequence graph obtained by GTC

|D|	Number of customers (size of Database)
|C|	Average number of transactions per Customer
|T|	Average number of items per Transaction
|S|	Average length of maximal potentially large Sequences
|I|	Average size of Itemsets in maximal potentially large sequences
N_S	Number of maximal potentially large Sequences
N_I	Number of maximal potentially large Itemsets
N	Number of items

Table 1: Parameters

vertex, the item 5 *and* 4 *verify the maxGap constraint. Finally, from both item* 4 *and item* 5 *in the itemset* (4 5)*, we can reach the itemset* (6) *while respecting the maxGap constraint.*

6 Experiments

In this section, we present the performance results of our GTC algorithm. As we are only interested in the performance of the preprocessing of the time constraints, experiments on GTC were carried out by considering that GTC is the implementation of the TCLW Algorithm (see Algorithm 1) and 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 450 MHz, 64M Bytes of main memory, Linux System and a 9G Bytes disk drive (SCSI).

Dataset	C	T	S	D	N
C20-D100-S10-N10	20	2.5	10	100K	10K
C20-D100-S8-N10	20	2.5	8	100K	10K
C20-D1-S10-N1	20	2.5	10	1K	1K

Table 2: Synthetic datasets

In order to assess the relative performance of the GTC algorithm and study its scale-up properties, we used two kinds of datasets: synthetic data, simulating *market-basket* data and access log files.

Synthetic data The synthetic datasets were generated using the program described in [SA95] (the synthetic data generation program is available at the following URL *http://www.almaden.ibm.com/cs/quest*) and parameters taken by the program are shown in Table 1. These datasets mimic real world transactions, where people buy a sequence of sets of items: some customers may buy only some items from the sequences, or they may buy items from multiple sequences. Like [SA96], we set $N_S = 5000$, $N_I = 25000$ and $I = 1.25$. The dataset parameter settings are summarized in Table 2.

Access log dataset The access log file was obtained from the Lirmm Home Page. The log contains entries corresponding to the requests made and its size is about 85 M Bytes. There were 1500 distinct URLs referenced in the transactions and 2000 clients.

6.1 Comparison of GTC with PSP

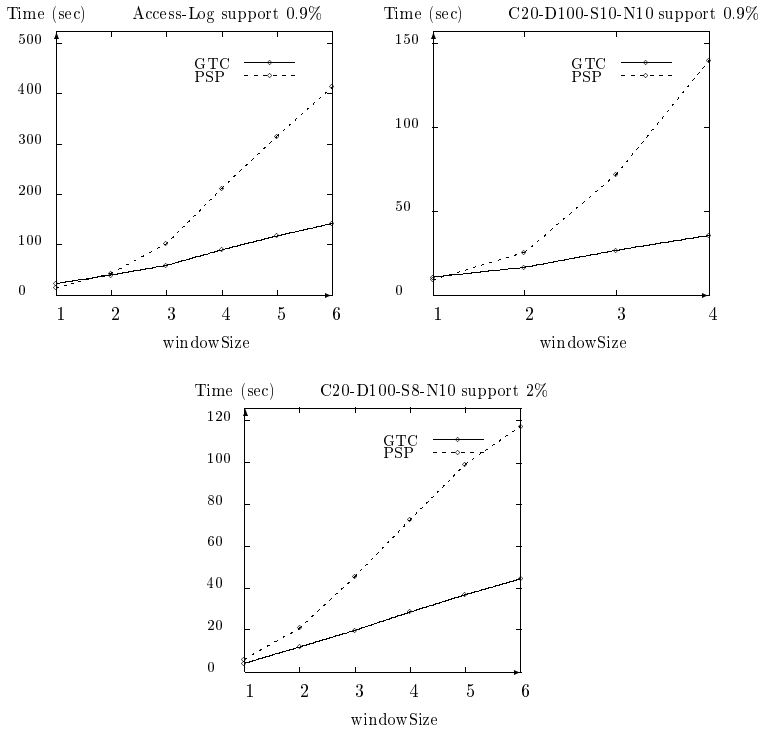

Figure 14: Execution times for synthetic and real datasets (minGap=1)

Figure 14 shows experiments conducted on the different datasets using different windowSize ranges to get meaningful response times. According to the previous discussion, for these experiments, minGap

was set to 1 and maxGap was set to ∞ (there is no maxGap constraint). 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. Results are given in Figure 15. As expected, we can observe that the number of recursive calls increases with the size of the windowSize parameter. Let us examine more closely igh windowSize values. For instance, in the C20-D100-S8-N10 dataset, we can notice that 35 million recursive calls are done with PSP. In this dataset, as the delay between two transactions is one, with a windowSize value of 6, PSP generates all the combination of the data sequence in this range and verify all generated data sequence into the candidate tree structure. In the same case, less than 10 millions of recursive calls are required by the GTC algorithm.

Note that during experiments, datasets are generated by using a high value for the average length of maximal potentially large sequences. As the complexity of the algorithm depends on the size of the result, we observe that GTC is well adapted to such sequences.

6.2 Performance in Scaled-up Databases

We examined how GTC behaves as the number of total transactions is increased. We would expect the algorithm to have near linear scaleup. This is confirmed by Figure 16 which shows that GTC scales up linearly as the number of transactions is increased ten-fold, from 0.1 million to 1 million. All the experiments were performed on the C20-D100-N10-S10 dataset with three levels of window size (2, 3 and 4). The support value σ was set to 2%. The execution times are normalized with respect to the time for the 0.1 million dataset.

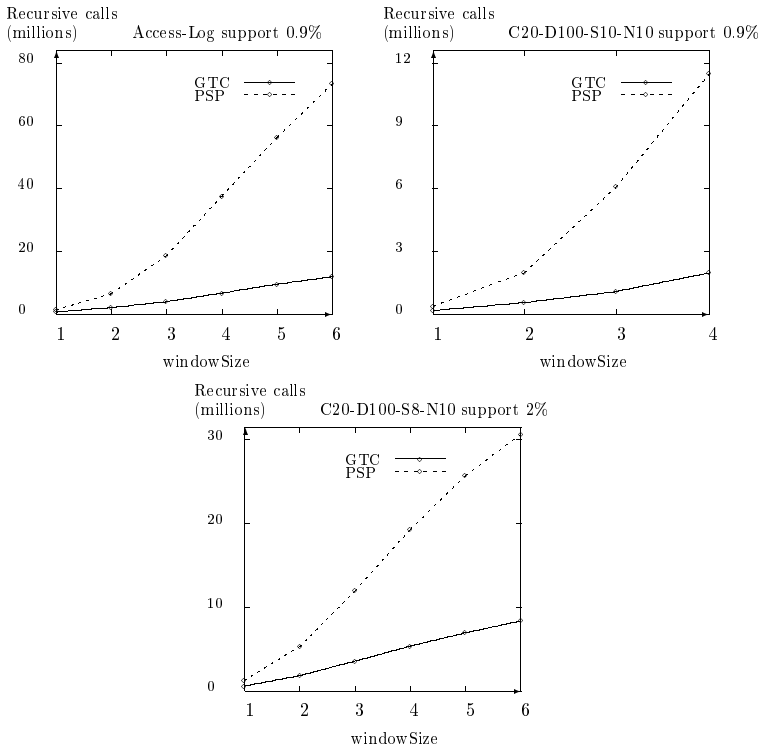

Figure 15: Recursive calls between Psp and Gtc (minGap=1)

6.3 Towards a Hybrid Approach

We designed some experiments in order to illustrate that, for the sake of efficiency, the time required for building the sequence graph must be less than the difference between the time for candidate verification in Psp and candidate verification in Gtc.

First, in order to find the time for building the sequence graph, we carried out some experiments to analyze the performance of Psp with Gtc when mining sequential patterns without time constraints.

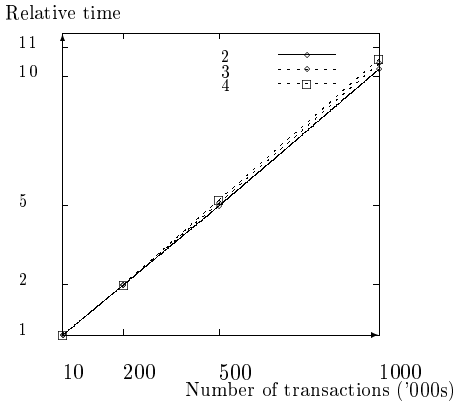

Figure 16: Scale-up: Number of customers

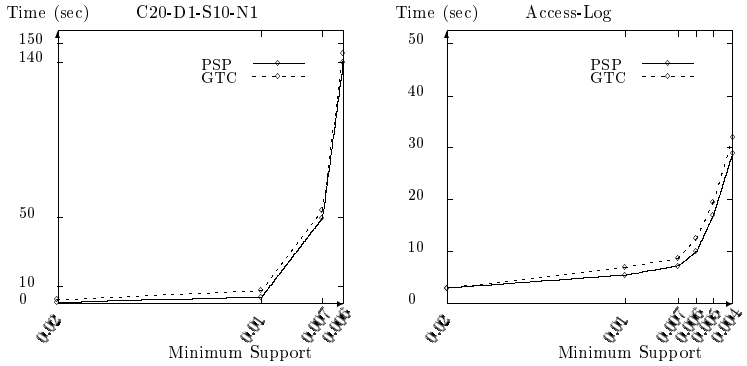

Figure 17: Execution times, windowSize=0

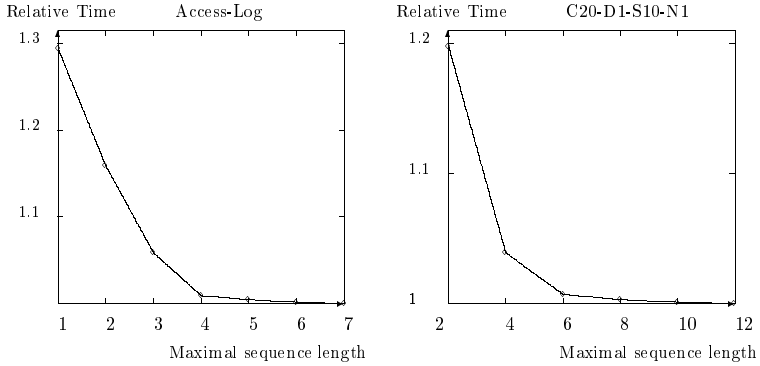

Figure 18: Relative execution times between GTC and PG-*Hybrid* (windowSize=1)

We used the C20-D1-S10-N1 database as well as the access log file with minimum support ranges. Figure 17 clearly indicates that, even if GTC is less efficient than PSP, the performance gap between the two algorithms is still constant. Furthermore, as expected, we also can observe that the execution time of GTC as well as PSP increases with decreasing minimum support.

Second, once the time for building the graph was exhibited, we carried out new experiments with time constraints in order to analyze the time spent in each phase of the algorithms. We observed that, even when considering low values for time constraints (for instance when windowSize or minGap was set to 1), at the second pass for verifying candidate, PSP was more efficient. Nevertheless, despite this efficiency, GTC was faster for the whole process. In order to take advantage of this behavior, we defined a new algorithm, called PG-*Hybrid*, using the PSP approach for the two first passes on the database and GTC for the following passes. As illustrated in Figure 18, this algorithm is very efficient when the length of the maximal sequences is low. For instance, when the length of the frequent sequence is one, PG-*Hybrid* is 1.3 times faster than GTC.

7 Conclusion

We considered the problem of discovering sequential patterns by handling time constraints. Instead of mining these constraints on the fly during the data sequence analysis, we attempted to use time constraint preprocessing for efficiently identifying generalized frequent sequences. We proposed a new algorithm called GTC based on the fundamental principles of levelwise algorithms but in which time constraints are handled in the earlier stage of the algorithm in order to provide significant benefits. According to the goal of minimizing the rewriting of previously defined algorithms, we pinpointed limitations due to the candidate structure handled. Nevertheless, ways for optimizing such a management structure were proposed. 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 designed a new algorithm called PG-*Hybrid* using the PSP approach for the two first passes on the database and GTC for the following scans. Experiments showed that this approach is very efficient even for short frequent sequences.

References

[AGYF02] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential PAttern Mining Using Bitmaps. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, July 2002.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 207–216, Washington DC, USA, May 1993.

[AS95] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, Tapei, Taiwan, March 1995.

[GNPP06] F. Giannotti, M. Nanni, D. Pedreschi, and F. Pinelli. Mining Sequences with Temporal Annotations. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC 06)*, pages 593–597, 2006.

[GRS02] M. Garofalakis, R. Rastogi, and K. Shim. Mining Sequential Patterns with Regular Expression Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 14(3):530–552, May 2002.

[HPMa+00] J. Han, J. Pei, B. Mortazavi-asl, Q. Chen, U. Dayal, and M. Hsu. FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD 00)*, pages 355–359, Boston, USA, 2000.

[KPWD03] H. Kum, J. Pei, W. Wang, and D. Duncan. ApproxMAP: Approximate Mining of Consensus Sequential Patterns. In *Proceedings of SIAM Int. Conf. on Data Mining*, San Francisco, CA, 2003.

[LHC06] M.-Y. Lin, S.-C. Hsueh, and C.-W. Chang. Mining Closed Sequential Patterns with Time Constraints. In *Proceedings of Workshop on Software Engineering, Databases, and Knowledge Discovery, International Computer Symposium 2006*, Taipei, Taiwan, 2006.

[LL05] Ming-Yen Lin and Suh-Yin Lee. Efficient Mining of Sequential Patterns with Time Constraints by Delimited Pattern Growth. *Knowledge and Information Systems*, 7(4):499–514, May 2005.

[LRBE03] M. Leleu, C. Rigotti, J.F. Boulicaut, and G. Euvrard. Constraint-based Mining of Sequential Patterns over Datasets with Consecutive Repetitions. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 03)*, pages 303–314, Cavtat-Dubrovnik, Croatia, September 2003.

[MCP98] 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*, pages 176–184, Nantes, France, September 1998.

[MPT04] F. Masseglia, P. Poncelet, and M. Teisseire. Pre-Processing Time Constraints for Efficiently Mining Generalized Sequential Patterns. In *Proceedings of the 11th International Symposium on Temporal Representation and Reasoning (TIME 04)*, pages 87–95, Tatihou Island, France, July 2004.

[OPS04] S. Orlando, R. Perego, and C. Silvestri. A New Algorithm for Gap Constrained Sequence Mining. In *Proceedings of the ACM Symposium on Applied Computing (SAC 04)*, pages 540–547, Nicosia, Chyprus, March 2004.

[PHMA+01] 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 *17th International Conference on Data Engineering (ICDE)*, 2001.

[SA95] R. Srikant and R. Agrawal. Mining Generalized Association Rules. In *Proceedings of the 21 st International Conference on Very Large Databases (VLDB'95)*, pages 407–419, Zurich, Switzerland, September 1995.

[SA96] 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)*, pages 3–17, Avignon, France, September 1996.

[SK02] M. Seno and G. Karypis. SLPMiner: an Algorithm for Finding Frequent Sequential Patterns using Length-decreasing Support Constraint. In *Proceedings of the IEEE International Conference on Data Mining (ICDM 02)*, pages 481–425, Maebashi City, Japan, December 2002.

[WH04] J. Wang and J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *Proceedings of the International Conference on Data Engineering (ICDE'04)*, Boston, M.A., March 2004.

[WMZ02] M. Wojciechowski, T. Morzy, and M. Zakrzewicz. Efficient Constraint-based Sequential Pattern Mining using Dataset Filtering Techniques. In *Proceedings of the Baltic Conference (BalticDB&IS 2002)*, pages 213–224, 2002.

[Zak00] M.J. Zaki. Sequence Mining in Categorical Domains: Incorporating Constraints. In *Proceedings of the 9th International Conference on Information and Knowledge Management (CIKM 00)*, pages 422–429, Washington, DC, November 2000.

[Zak01] M.J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning Journal*, 42(1):31–60, February 2001.