Incremental mining of frequent sequences from a window sliding over a stream of itemsets

Thomas Guyet[*,**], René Quiniou[***]

[*]AGROCAMPUS OUEST, UMR6074 IRISA, F-35042 Rennes
[**]Université européenne de Bretagne
[***]INRIA Centre de Rennes - Bretagne Atlantique

Abstract. Nous introduisons le problème de fouille des séquences fréquentes dans une fenêtre glissante sur un flux d'itemsets. Pour résoudre ce problème, nous présentons un algorithme incrémental, complet et correct, basé sur une représentation des séquences fréquentes inspiré par l'algorithme PSP et sur une méthode de comptage des occurrences minimales. Les expériences ont été menées sur des données simulées et sur des données réelles consommation instantanée d'électricité. Les résultats montrent que notre algorithme incrémental améliore de manière significative le temps de calcul par rapport à une approche non-incrémentale.

1 Introduction

Sequential pattern mining has been studied extensively in static contexts (Masseglia et al. (1998); Pei et al. (2004); Srikant and Agrawal (1996)). Most of the proposed algorithms make use of the Apriori antimonotonicity property stating that if a pattern is frequent then all its sub-patterns are also frequent. All these algorithms make use of a pattern counting method based on the number of transactions that contain the pattern without taking into account the multiple occurrences of the pattern in a transaction.

Counting the number of occurrences of a motif, or episode, in a window introduces some complexity compared to previous approaches as two occurrences of an episode can overlap. Mannila et al. (1997) introduced the algorithms Minepi and Winepi for extracting frequent episodes and counting their minimal occurrences. Since then, other counting methods have been proposed to solve this problem while preserving the antimonotonicity properties required for the effectiveness of pattern occurrences search (see Achar et al. (2010)).

In the context of data streams, where data arrive in a continuous flow, specific algorithmic solutions must be designed for the extraction of frequent sequences. A common practice consists in sliding a window over the data and then in extracting frequent episodes from the successive itemsets inside this window. But the itemset series is evolving continuously: when a new itemset arrives, the oldest itemset at the beginning of the window becomes obsolete. In existing static approaches, it is necessary to restart an entire mining process to extract the frequent episodes of the new period.

As the context of data streams imposes to process the incoming data very fast, the naïve approach above has a prohibitive computation time. Thus, an incremental method that updates efficiently the current set of sequential patterns is needed. We propose a method based on a representation of frequent sequences inspired by PSP (Masseglia et al. (1998)), an improvement of GSP (Srikant and Agrawal (1996)) for mining frequent itemset subsequences from a database of sequences of itemsets. This representation enables an efficient update of sequential pattern occurrences due to new data and obsolete data as well.

In section 2, we present related work. In section 3, we present a formal setting of the problem of mining itemsets from a data stream. In section 4, we present an incremental algorithm that solves this problem. In section 5, we introduce the data structure used to collect the history of frequent sequences. In section 6, we present experimental results on simulated data as well as real data related to instantaneous power consumption.

2 Related work

Since the publication of the seminal paper of Agrawal et al. (1993) establishing the foundations of itemset mining, many proposals have been made. We are particularly interested in sequential pattern mining, with a special focus on incremental and progressive methods, such as those proposed in the field of data streams.

Many methods have been proposed to discover frequent subsequences or sequential patterns either from a sequence database or from a single long sequence, in a static context or in data streams. GSP (Srikant and Agrawal (1996)), PSP (Masseglia et al. (1998)), PrefixSpan (Pei et al. (2004)), Spade (Zaki (2001)), Spam (Ayres et al. (2002)), to cite a few, proposed different approaches to the problem of mining sequential patterns (*i.e.* frequent sequences of itemsets) from a static sequence (of transactions) database. There are less proposals for mining frequent recurring patterns from a single long sequence. Minepi, Winepi (Mannila et al. (1997)), WinMiner (Méger and Rigotti (2004)), for instance, count the minimal occurrences of episodes in a given sequence, while Laxman et al. Laxman et al. (2007) counts the non-overlapped occurrences of episodes. These approaches consider either serial episodes where items (events) are strictly ordered or parallel episodes where items are unordered. Recently, Tatti and Cule (2011) introduced episodes with simultaneous events where multiple events may have the same timestamp, aiming at handling quasi-simultaneous events. These works do not address compute th whole set of frequent sequential patterns and do not address the problem of forgetting obsolete data.

Data streams introduce a challenge to sequential pattern mining since data are received at high rate and are possibly infinite. To cope with such massive data, several approaches of itemset mining in data streams have proposed approximate solutions based on landmark windows (Manku and Motwani (2002)), tilted-time windows (Giannella et al. (2004)), damped windows (Chang and Lee (2003)) or sliding windows (Li and Lee (2009)). Few proposals have been made for sequential pattern mining over data streams. Chen et al. (2005) consider multiple streams recoded as a (single) sequence of itemsets grouping the items occurring at more or less the same time. Their algorithm Mile adapts PrefixSpan to mine frequent sequences that appear in a sufficient number of fixed size windows. SPEED (Raïssi et al. (2006)) considers successive batches of sequences extracted from a data stream. Frequent sequential patterns are stored in a tilted-time based structure which prunes less frequent or too old patterns.Marascu and Masseglia (2006) propose to cluster stream data to build a summary from which sequential patterns can be extracted. IncSpam (Ho et al. (2006)) uses a bit-sequence representation of items to maintain the set of sequential patterns from itemset-sequence streams with a transaction-sensitive sliding window. Most of these approaches aim at maintaining a good approximation of the newest and most frequent sequential patterns of the data stream. Moreover, they deal with a transaction-based scheme representation of the data wheareas we are interested of mining sequential patterns from a single long sequence.

Mining sequential patterns from data streams is similar to mining sequential patterns incrementally from dynamic databases. ISE (Masseglia et al. (2003)) considers the arrival of new customers and new items and extends the set of candidate patterns from the original database accordingly. IncSP (Lin and Lee (2004)) uses efficient implicit merging and counting over appended sequences. IncSpan (Cheng et al. (2004); Nguyen et al. (2005)) maintains a tree of frequent and semi-frequent patterns to avoid many multiple scans upon database updates. However, these incremental methods handle new data but not obsolete data. Recently, some proposals have been made that view sequential pattern mining as an incremental process: a window slides continuously over the data and, as time goes by, new data are added to the window and old data are removed. SSM (Ezeife. and Monwar (2007)) maintains three data structures to mine incrementally sequential patterns from a data stream. D-List maintains the support of all items in the data stream. PLWAP tree stores incrementally frequent sequences from batches. The frequent sequential pattern tree FSP is constructed incrementally from batch to batch. Pisa (Huang et al. (2008)) mines the most recent sequential patterns of a progressive sequence database. It maintains a PS-tree (progressive sequential tree) to keep the information data from a window sliding of the stream. PS-tree stores the timestamped sequence items and also efficiently accumulates the occurrence frequency of every candidate sequential pattern. Update operations remove obsolete information and add new information to the three data structures. Recently, Patnaik et al. (2012) proposed a streaming algorithm for pattern mining over an event stream. Their aim is to extract the top-k frequent episodes of an arbitrary length l from the successive batches of a window of interest defined over an event stream. Episodes are parallel with no overlapping occurrences. They use an Apriori-like method to maintain the set of top-k episodes on each successive batch. The incremental algorithm makes use of the negative border to reduce, as much as possible, the set of candidates from one batch to the other.

3 Basic concepts and problem statement

3.1 Items, itemsets and sequences

From now on, $[n]$ denotes the set of the n first integers, *i.e.* $[n] = \{1, \dots, n\}$.

Let $(\mathcal{E}, =, <)$ be the set of items and $<$ a total order (*e.g.* lexicographical) on this set. An *itemset* $A = (a^1, a^2, \dots, a^n),\ a^i \in \mathcal{E}$ is an ordered set of distinct items, *i.e.* $\forall i \in [n-1],\ a^i < a^{i+1}$ and $i \neq j \Rightarrow a^i \neq a^j$. The size of an itemset α, denoted $|\alpha|$ is the number of items it contains. An itemset $\beta = (b^1, \dots, b^m)$ is a sub-itemset of $\alpha = (a^1, \dots, a^n)$, denoted $\beta \sqsubseteq \alpha$, iff β is a subset of α.

A *sequence* S is an ordered series of itemsets $S = \langle s_1, s_2, ..., s_n \rangle$. The length of a sequence S, denoted $|S|$, is the number of itemsets that make up the sequence. The size of a sequence S, denoted $\|S\|$, is the total number of items it contains $\|S\| = \sum_{i=1}^{|S|} |s_i|$. $T = \langle t_1, t_2, ..., t_m \rangle$ is a *sub-sequence* of $S = \langle s_1, s_2, ..., s_n \rangle$, denoted $T \preceq S$, iff there exists a sequence of integers $1 \leq i_1 < i_2 < ... < i_m \leq n$ such that $\forall k \in [m], t_k \sqsubseteq s_{i_k}$.

Example 1. *Let $\mathcal{E} = \{a, b, c\}$ with the lexicographical order ($a < b$, $b < c$) and the sequence $S = \langle a(bc)(abc)cb \rangle$. To simplify the notation, we omit the parentheses around itemsets containing a single item. The size of S is 8 and its length is 5. For instance, sequence $\langle (bc)(ac) \rangle$ is a sub-sequence of S.*

The relation $\preceq$ is a partial order on the set of sequences.

3.2 Stream of itemsets

A *stream of itemsets* $F = \{f_i\}_{i \in \mathbb{N}}$ is an infinite sequence of itemsets that evolves continuously. It is to be assumed that only a small part of it can be kept in memory.

The *window* W of size w at time t is the sequence $W = \langle f_{i_1}, f_{i_2}, ..., f_{i_n} \rangle$ such that $\forall k \in [n],\ t \leq i_k \leq t + w$ and $\forall f_i \in F \setminus W,\ i < t$ or $i > t + w$. The *current window* of size w of a stream F is the window beginning at time $t - w + 1$ where t is the position of the last itemset appeared in the stream. This window includes the most recent itemsets of the stream.

Definition 1 (Instances of a sequence in a window)**.** *The set of instances of a sequence $S = s_1, \ldots, s_n)$ of length n in a window $W = (w_1, \ldots, w_m)$, denoted $\mathcal{I}_W(S)$, is the list of n-tuples of positions (within W) corresponding to the* ***minimal occurrences*** *of S in W (see Mannila et al. (1997)).*

$$\mathcal{I}_W(S) = \big\{ (i_j)_{j \in [n]} \in [m] \mid \begin{array}{ll} \forall j \in [n], s_j \sqsubseteq w_{i_j}, & (1) \\ \forall j \in [n-1],\ i_j < i_{j+1}, & (2) \\ (w_j)_{j \in [i_1+1, i_n]} \npreceq S, & (3) \\ (w_j)_{j \in [i_1, i_n - 1]} \npreceq S \ \} & (4) \end{array}$$

Condition (1) requires that any itemset of S is a sub-itemset of an itemset of W. Condition (2) specifies that the order of itemsets of W must be respected. In addition, any itemset of W cannot be a super-itemset of two distinct itemsets of S. This condition does not impose any time constraint between itemsets. Conditions (3) and (4) specify minimal occurrences: if an instance of S has been identified in the interval $[i_1, i_n]$, there can't be any instance of S in a strict subinterval of $[i_1, i_n]$.

Example 2. *Let $W = \langle a(bc)(abc)cb \rangle$ be a window on some stream. We focus on instances of the sequence $S = \langle (ab)b \rangle$. To find an instance of this sequence, we have to locate itemsets of S: (ab) appears only at position 3 ($(ab) \sqsubseteq (abc)$). b appears at positions 2, 3 and 5. Sequence S has only one instance: $(3, 5)$. Thus $\mathcal{I}_W(\langle (ab)b \rangle) = \{(3, 5)\}$.*

Now, let us consider the window $W = \langle acbbc \rangle$ to illustrate the conditions (3) and (4). Without these conditions, the instances of the sequence $\langle ab \rangle$ would be $\{(1, 3), (1, 4)\}$. Condition (4) prohibits the instance $(1, 4)$ because $(1, 3)$ is an instance of $\langle ab \rangle$ in the window such that $[1, 3] \subset [1, 4]$. *Thus, $\mathcal{I}_W(\langle ab \rangle) = \{(1, 3)\}$.*

For $W = \langle aaaa \rangle$, $\mathcal{I}_W(\langle aa \rangle) = \{(1, 2), (2, 3), (3, 4)\}$ and $\mathcal{I}_W(\langle aaa \rangle) = \{(1, 2, 3), (2, 3, 4)\}$.

Let W be a window and S a sequence of itemsets. The **support** of the sequence S in window W, denoted $supp_W(S)$ is the cardinality of $\mathcal{I}_W(S)$, *i.e.* $supp_W(S) = card\,(\mathcal{I}_W\,(S))$.

Note that, in the general case, the support function $supp_W(\cdot)$ is not anti-monotonic on the set of sequences with associated partial order $\preceq$ (see Tatti and Cule (2012)).

Definition 2 (Mining a stream of itemsets)**.** *Given a threshold σ, we say that a sequence S is* frequent *in a window W of size w iff $supp_W(S) \geq \sigma$. Mining a stream of itemsets consists in extracting at every time instant, all the frequent sequences in the most recent sliding window.*

In approaches that consider multiple parallel streams, *e.g.* Ho et al. (2006); Marascu and Masseglia (2006); Raïssi et al. (2006), the support of a sequence is usually defined as the number of streams in which this sequence appears. In this work, we consider a single stream and the support of a sequence is the number of instances of this sequence in the current window corresponding to the most recent data.

4 Incremental algorithm for mining a stream of itemsets

In this section, we present an incremental algorithm for mining frequent sequences in a window sliding over a stream of itemsets. The aim of such an algorithm is to efficiently update the set of frequent sequences following two kinds of window transformations: the addition of an itemset at the end of the window and the removal of an itemset at the beginning of the window.

The algorithm 1 presents this general idea of incremental mining of itemset stream. Algorithmic difficulties lie in both function DETETEFIRST and ADD which must efficiently update the set of frequent sequences. These functions will be detailled in the following of this section.

```
Input: $\mathcal{F}$: itemset stream, $w$: windows size, $\sigma$: threshold
 1: $t \leftarrow 1$
 2: while $t \leq w$ do                                         ▷ Stream beginning
 3:     $\alpha \leftarrow \mathcal{F}(t)$
 4:     $\mathcal{A} \leftarrow$ ADD($\alpha, \mathcal{A}$)                       ▷ Update $\mathcal{A}$ by adding itemset $\alpha$
 5:     $t \leftarrow t + 1$
 6: end while
 7: while true do
 8:     $\alpha \leftarrow \mathcal{F}(t)$
 9:     $\mathcal{A} \leftarrow$ DELETEFIRST($\mathcal{A}$)      ▷ Update $\mathcal{A}$ by removing references to the first itemset
10:     $\mathcal{A} \leftarrow$ ADD($\alpha, \mathcal{A}$)
11:     $t \leftarrow t + 1$
12: end while
```

FIG. 1 – *Incremental mining of a stream of itemset $\mathcal{F}$.*

The algorithm relies on representing the set of frequent sequences in a tree, the structure of which is inspired by the prefixing method of PSP of Masseglia et al. (1998). While GSP represents the set of frequent sequences of itemsets as a tree where the edges mean sequentiality, PSP represents a set of frequent sequences as a tree with two types of edges: the edges representing sequentiality between itemsets and the edges representing the composition of itemsets. Masseglia et al. showed that this representation is equivalent to GSP while requiring less memory.

4.1 Representing sequences by a PSP tree

The set of frequent sequences is represented by a prefix tree, the nodes of which have the structure defined below.

Definition 3 (Node)**.** *A node N is a 4-tuple $\langle \alpha, \mathcal{I}, \mathcal{S}, \mathcal{C} \rangle$ where*

- *$\alpha = (a_1, \dots, a_n)$ is a sequence of size n,*
- *$\mathcal{I} = \mathcal{I}_W(\alpha)$, the instance list of sequence α in W,*
- *$\mathcal{S}$ is the set of descendant nodes which represent sequences $\beta = (b_1, \dots, b_{n+1})$ of size $\|\alpha\| + 1$ such that $\forall i \in [n],\ a_i = b_i$, (i.e. b_{n+1} is a strict successor of a_n),*
- *$\mathcal{C}$ is the set of descendant nodes which represent sequences $\beta = (b_1, \dots, b_n)$ of size $\|\alpha\| + 1$ such that $\forall i \in [n-1],\ a_i = b_i$, $a_n \sqsubseteq b_n$ and $\forall j < |a_n|,\ a_n^j < b_n^{|a_n|+1}$, (i.e. itemset b_n extends itemset a_n with the item $b_n^{|a_n|+1}$).*

Definition 4 (Tree of frequent sequences)**.** *$\mathcal{A}_\sigma(W)$ denotes the tree that represents all sequences of W having a support greater than σ. The root node of a prefix tree is a node of the form $\langle \{\}, \emptyset, \mathcal{S}, \mathcal{C} \rangle$.*

Let N be a node of $\mathcal{A}_\sigma(W)$. The subtree rooted at node N represents the tree composed of all descendants of N (including N).

Despite the non-general anti-monoticity property of the support, it may be proved that the support is anti-monotone wrt the PSP-tree relations. If a node has a support greater than or equal to σ then all its ancestors are frequent sequences in W. In addition, each node – apart from the root – has a single parent. This ensures that recursive processing the PSP tree is complete and non-redundant.

Example 3. *Let $W = \langle a(bc)(abc)cb \rangle$ and $\sigma = 2$. Figure 2 shows the tree $\mathcal{A}_\sigma(W)$. Solid lines indicate membership in the set $\mathcal{S}$ (Succession in the sequence), while the dotted lines indicate membership in the set $\mathcal{C}$ (Composition with the last itemset). The node $(bc)b$, highlighted in gray, has the sequence node (bc) as parent, since $(bc)b$ is obtained by concatenating b to (bc). The parent node of (bc) is (b) and is obtained by itemset composition (dotted line). At each node of Figure 2, the instance list of the sequence is displayed in the index. For example, the sequence $(bc)c$ has two instances: $\mathcal{I}(\langle (bc)c \rangle) = \{(2,3),\ (3,5)\}$.*

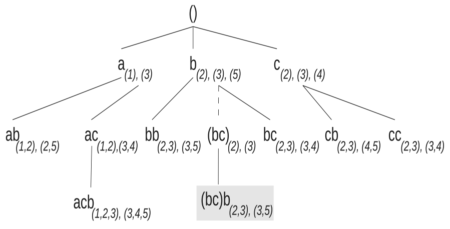

FIG. 2 – *Example of a tree of frequent sequences ($\sigma = 2$)*

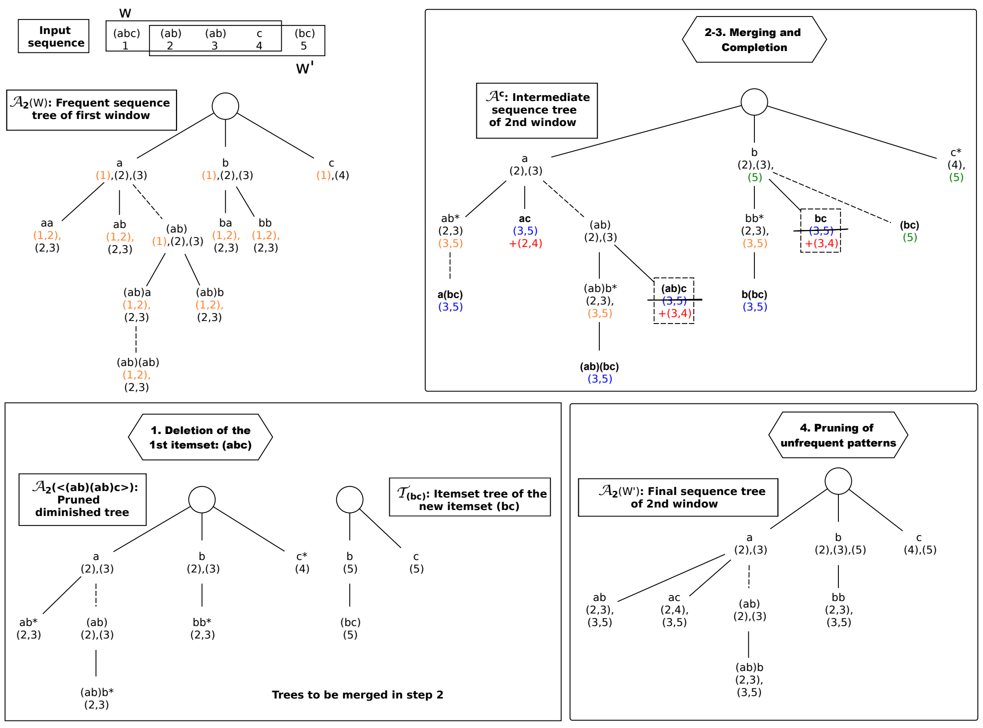
Input sequence
W
(abc) 1
(ab) 2
(ab) 3
c 4
(bc) 5
W'
$\mathcal{A}_2$(W): Frequent sequence tree of first window
2-3. Merging and Completion
$\mathcal{A}^c$: Intermediate sequence tree of 2nd window
1. Deletion of the 1st itemset: (abc)
$\mathcal{A}_2$(<(ab)(ab)c>): Pruned diminished tree
$\mathcal{T}_{(bc)}$: Itemset tree of the new itemset (bc)
Trees to be merged in step 2
4. Pruning of unfrequent patterns
$\mathcal{A}_2$(W'): Final sequence tree of 2nd window

FIG. 3 – *Successive steps for updating the sequence tree upon the arrival of itemset* (bc) *in the window* $W = \langle (abc)(ab)(ab)c \rangle$.

4.2 Illustration of the method

This section illustrates the principle of the proposed method. For space reasons, the algorithm details have been omitted. The incremental process aims at updating the frequent sequence tree from data in the most recent window of the stream and determining whether the sequences are frequent. The arrival of a new itemset in the stream triggers two steps: (1) the deletion of instances related to the first itemset in the window, (2) the addition of sequences and instances related to the new incoming itemset (see Figure 1). The addition step carries contributes to most of the computational load. It involves three substeps: merging sub-itemsets of the new itemset into the current tree, completing the instance lists and pruning non-frequent sequence nodes.

The deletion step is performed before the addition of a new itemset in order to reduce the size of the tree before the time-consuming merging and completion substeps.

Let us consider the window $W = \langle (abc)(ab)(ab)c \rangle$ of length 4, at position 1 of the stream. Assume that $\mathcal{A}_2(W)$, *i.e.* the sequence tree with support greater than 2, has been already built. Figure 3 describes the successive steps for transforming the frequent sequence tree $\mathcal{A}_2(W)$ into the tree $\mathcal{A}_2(W')$ at the arrival of the new itemset (bc).

1. Deletion of the first itemset: all instances starting at the first (oldest) position of the window (orange instances at position 1, in the example) are deleted. Then, sequences having a number of instances less than $\sigma = 2$ are deleted from the tree. The result is the tree $\mathcal{A}_2(\langle(ab)(ab)c\rangle)$ where a, (ab), b are frequent. Quasi-frequent sequences (marked with a $\star$ in the example) are not frequent but may become frequent as they have have a frequence equal to $\sigma - 1$ and they are ended by an item present in the new itemset, (bc) here. Such nodes are kept in the frequent tree with their occurence list because no completion (see below) will be necessary for them.

2. Merging the new current itemset (bc) **with every node of the sequence tree**: this step generates all the new candidate sequences of the new window. Intuitively, a sequence is a new candidate (*i.e.* potentially frequent) only if it is the concatenation of a sub-itemset of (bc) to a frequent sequence of $\langle(ab)(ab)c\rangle$. In the frequent sequence tree, this concatenation can be seen as extending each node of $\mathcal{A}_2(\langle(ab)(ab)c\rangle)$ with the itemset tree $\mathcal{T}_{(bc)}$ representing all sub-itemsets of (bc).

In Figure 3, the tree $\mathcal{T}_{(bc)}$ is merged with the four nodes of $\mathcal{A}_2(\langle(ab)(ab)c\rangle)$ not marked with a *:

- with the root node (green instances): all subsequences of (bc) become potentially frequent.
- with the nodes a, (ab), b (blue instances): all sequences starting with one of this three sequences (frequent in $\langle(ab)(ab)c\rangle$) and followed by a sub-itemset of (bc) become potentially frequent.

We talk about "tree merging" because if a node already exists in the tree (*e.g.* node (b)), the instance related to the new itemset is added to the list of existing instances. The list of instances of (b) becomes $\{(2), (3), (5)\}$. We know that each of these nodes holds all the instances of the associated sequence in W'. New nodes are noted in bold face in the frequent tree after the merging step in Figure 3. Each of these new nodes of $\mathcal{A}^f$, *e.g.* the node (bc), has an occurrence list consisting of only one instance of a sub-itemset of (bc). Quasi-frequent nodes (nodes marked with a *) are not merged with the itemset tree $\mathcal{T}_{(bc)}$. Their occurrence lists are simply updated, if needed.

3. Completion of instance lists: For new candidate nodes but only for those, it is necessary to scan the window W' once again to build the complete list of instances of a sequence. For example, the node ab is associated with the instance list $\{(3, 5)\}$. This list must be completed with the list of instances of ab in the previous window where it was unfrequent, *i.e.* $\{(2, 3)\}$. Red instances of the tree $\mathcal{A}^c$ in Figure 3 show the instances added by completion.

4. Pruning non-frequent sequences: $\mathcal{A}^c$, the tree obtained after completion, contains new candidate sequences with complete instance lists. The last step removes sequences with an instance list of size strictly lower than $\sigma = 2$ yielding the tree $\mathcal{A}_2(W')$.

4.3 Merging an itemset tree into a frequent sequence tree

Now, we detail the merging step which integrates the itemset tree $\mathcal{T}$ into the sequence tree $\mathcal{A}$. Then, we explain instance list completion.

Merging the itemset tree $\mathcal{T}$ with every node of the frequent sequence tree $\mathcal{A}$ consists of two main steps (see Figure 4):

- prefixing the itemset tree $\mathcal{T}$ with the sequence of node N,
- recursively merging the prefixed $\mathcal{T}$ with descendants of node N (*cf.* Algorithm 5).

FIG. 4 – MERGING*: merging the itemset tree $\mathcal{T}$ with every node of the sequence tree $\mathcal{A}$.*

```
function MERGING($\mathcal{A}$, $\mathcal{T}$)
    $\mathcal{T}' \leftarrow \mathcal{T}$
    for $N \in \mathcal{A}$ do
        for $n \in \mathcal{T}'$ do                                  ▷ Prefixing $\mathcal{T}'$
            $n.\alpha = N.\alpha \oplus n.\alpha$                    ▷ Prefixing the sequence with $N.\alpha$
            for all $I \in n.\mathcal{I}$ do    ▷ Prefixing the instances with the last element of $N.\mathcal{I}$, noted $d$
                $I = d \cup I$
            end for
        end for
        RECMERGE($\mathcal{T}'$, $N$)                    ▷ Recursive merging of $\mathcal{T}'$ with nodes $N$ of $\mathcal{A}$
    end for
    return $\mathcal{A}$
end function
```

Let $N.\alpha$ denote the sequence associated with node N from the sequence tree $\mathcal{A}$ and $N.\mathcal{I}$ denote the instance list associated with the same node N. For each node N of $\mathcal{A}$, the itemset tree $\mathcal{T}$ is first prefixed by N: on the one hand, the sequences of each node of $\mathcal{T}$ are prefixed by $N.\alpha$; on the other hand, all instances of $\mathcal{T}$ are prefixed by the last instance of $N.\mathcal{I}$. Using the last instance of $N.\mathcal{I}$ enforces the third property of Definition 1.

FIG. 5 – RECMERGE*: recursively merging the prefixed itemset tree $\mathcal{T}$ with a node of $\mathcal{A}$*

```
Input: $n$: itemset node tree, $N$: node of the sequence tree to be merged with $n$ and such that $n.\alpha = N.\alpha$
1:  function RECMERGE($n$, $N$)
2:      $N.\mathcal{I} \leftarrow N.\mathcal{I} \cup n.\mathcal{I}$                         ▷ Merging instance lists
3:      for $s_N \in N.\mathcal{S} \cup N.\mathcal{C}$ do                                  ▷ Recursion
4:          for $s_n \in n.\mathcal{S} \cup n.\mathcal{C}$ do
5:              if $s_N.\alpha = s_n.\alpha$ then
6:                  $found \leftarrow$ True
7:                  RECMERGE($s_n$, $s_N$)
8:              end if
9:          end for
10:         if not $found$ then
11:             if $s_n \in n.\mathcal{S}$ then
12:                 $N.\mathcal{S} \leftarrow N.\mathcal{S} \cup \{$COPY($s_n$)$\}$
13:             else
14:                 $N.\mathcal{C} \leftarrow N.\mathcal{C} \cup \{$COPY($s_n$)$\}$
15:             end if
16:         end if
17:     end for
18: end function
```

In a second step, the algorithm recursively merges the root of the itemset tree $\mathcal{T}$ prefixed by N. Algorithm 5 details this merging operation. We must first make sure that $n.\alpha = N.\alpha$ to verify that the two nodes represent the same sequence. At line 2, instance lists of nodes n and N are merged. By construction of the new instance, the conditions of Definition 1 are satisfied.

Then, the descendants of n are processed recursively. For each node of $n.\mathcal{S}$ (resp. $n.\mathcal{C}$), we search a node s_n in $N.\mathcal{S}$ (resp. $N.\mathcal{C}$) such that these nodes represent the same sequence. If such a node is found, then the function `RecMerge` is recursively applied. Otherwise, a copy of the entire subtree of s_n is added to $n.\mathcal{S}$ (resp. $n.\mathcal{C}$).

4.4 Instance lists completion

One of the difficult task is merging the instance lists of nodes n into the one of N. When a new sequence is introduced in the tree, other instances of this sequence may be present in the previous window (corresponding to the beginning of the current window) but unfrequent and, so, they were not stored in the tree (except quasi-frequent sequences). For example, in Figure 3, the sequence $\langle bc\rangle$ (node surrounded by a dotted line square) is not frequent in W and is not present in the frequent sequence tree $\mathcal{A}_2(W)$. However, after the arrival of itemset (bc) the sequence $\langle bc\rangle$ may become frequent in W. Thus, it is necessary to scan W' to find all instances of $\langle bc\rangle$ to compute its frequency.

For thesake of completeness, the instance list must be completed. To make the completion efficient, the algorithm uses the projection principle of PrefixSpan to reduce the number of scans over the sequence W. For succession nodes, the completion scans only the sub-sequence of W' composed of the itemsets between $i_{|\beta|}+1$ and $j_{|\beta|-1}$, where $J=(j_1,...,j_{|\beta|})$ is the instance after I in the list of instances of β. Properties (2) and (4) of Definition 1 ensure that the completed instance lists are correct and complete. Thus, the overall algorithm is complete and correct.

4.5 Time complexity analysis

To the best of our knowledge, there exists no results about the theoretical complexity of the minimal occurrence mining task. Ding et al. Ding et al. (2009) have shown that testing whether the support of a sequence is equal to k is NP for counting non-overlapping instances and NP-complete for counting strong non-overlapping instances. However, these two counting methods are not equivalent to minimal occurrences counting. These two counting methods are not equivalent to minimal occurrences counting.

Two instances $(i_j)_{j\in[m]}$ and $(i'_j)_{j\in[m]}$ of a sequence S of size m are overlapping iff $\exists 1 \leq l < m : i_l = j_l$. Two instances are strong-overlapping iff $\exists 1 \leq l < m$ and $1 \leq l' < m : i_l = j'_l$.

In this section, we estimate the time complexity of updating the frequent tree. For this purpose, we consider a sequence of items, *i.e.* consisting of itemsets of size 1 only.

Proposition 1. *Let W be a window of size w, ql the size of the vocabulary and σ the frequency threshold. Considering a sequence S of size n, we denote by $N_{ql,n,\sigma}$ the number of nodes in the tree $\mathcal{A}_\sigma(S)$.*

The time complexity of updating a frequent sequence tree decomposes as follows:

1. *Deletion of obsolete instances: $\mathcal{O}(N_{ql,w,\sigma})$,*
2. *Merging the current itemset tree with the current frequent sequence tree: $\mathcal{O}\left(N_{ql,w-1,\sigma}\right)$,*
3. *Completion of instance lists (in the worst case): $\mathcal{O}\left(N_{ql,w-1,\sigma}\times w\right)$*

In the worst case, the time complexity of the updating a frequent sequence tree is

$$\mathcal{O}\left(N_{ql,w,\sigma} \times (1+w) + N_{ql,w-1,\sigma}\right)$$

Proof.

1. The deletion of the obsolete itemset is performed by processing each node of the tree. Considering that we cope with minimal occurrences only and that instance lists are ordered from the oldest to the newest, it is sufficient to process the first instance of instance lists. Pruning is performed during the same operation. Thus, the time complexity of the deletion is $\mathcal{O}(N_{ql,w,\sigma})$. This step yields the pruned diminished sequence tree.
2. In case of sequence of items, $\mathcal{T}$, the tree associated with the new itemset, has only one node. In this case, the merging operation consists in processing each node of the frequent sequence tree sequentially by adding an instance or by adding a new node. The overall time complexity is then $\mathcal{O}(N_{ql,w-1,\sigma})$ where $N_{ql,w-1,\sigma}$ is the size of the frequent sequence tree corresponding to the old window where the first itemset has been deleted. This step yields the merged sequence tree.
3. In the worst case, *i.e.* when the new item is not in the merged sequence tree, the number of created nodes is equal to the number of nodes in the pruned diminished sequence tree, thus it is lower or equal to $N_{ql,w-1,\sigma}$. For each node, the algorithm has to complete the instance list. In the worst case, the algorithm browses the whole sequence W. Thanks to the minimal occurrences constraints, it's sufficient to complete the instance list.

□

5 Frequent sequences history over the stream

In real applications, it is interesting to highlight the mode changes of the observed system by analyzing its behavior through frequent sequences. In this section, we describe the construction of an history of frequent itemsets without trivial matches (Lin et al. (2002)). The history data structure is computed while processing a stream to give a quick view of the distribution of frequent patterns over the stream. We do not address the issue of history compression such as some proposals, *e.g.* Tilted-Time Windows (Giannella et al. (2003)) or REGLO (Marascu and Masseglia (2006)). The memory space required by this data structure grows linearly with time.

Definition 5 (History of frequent sequences)**.** *Let S be a sequence. $\mathcal{H}^{\sigma,w}(S)$ denotes the history of the sequence S over a stream and is defined by*

$$\mathcal{H}^{\sigma,w}(S) = \left([l_1,u_1], \ldots, [l_m,u_m]\right),$$

with $l_i, u_i \in \mathbb{N}$ such that $\forall i \in [m],\ 0 < l_i \leq u_i$ and $\forall i \in [m-1],\ u_i < l_{i+1}$.

This means that the sequence S is frequent, according to the threshold σ, in the windows of size w starting at time t in the stream for all $t \in [l_i, u_i]$ with $i \in [m]$.

Algorithm of Figure 6 describes the function that updates the sequence history with the current frequent sequences. This function may be inserted in the algorithm of the Figure 1 line 11 in order to construct the sequence history on the stream. The principle of the algorithm is to update the history at the end of the processing of the arrival of a new itemset in the sequence.

For each sequence frequent in the window beginning at time t, if the sequence was already frequent at time $t-1$ the last interval is enlarged with the date of the current window ; if the sequence was unfrequent at $t-1$ a new interval is created.

FIG. 6 – HISTORYUPDATE:

Input: t: current window date, $\mathcal{A}$: frequent sequences PSP-tree, $\mathcal{H}^{\sigma,w}$: sequences history

```
function HistoryUpdate($t$)
    for all $N \in \mathcal{A}$ do                                ▷ For all frequent sequence
        $([l_i, u_i])_{i\in[m]} \leftarrow \mathcal{H}^{\sigma,w}(N.\alpha)$
        if $u_m = t-1$ then
            $u_m \leftarrow t-1$
        else
            $\mathcal{H}^{\sigma,w} \leftarrow \mathcal{H}^{\sigma,w} \times [t,t]$
        end if
    end for
    return $\mathcal{H}^{\sigma,w}$
end function
```

Example 4 (History of frequent sequences)**.** *Given the stream $\mathcal{F} = \langle a(bc)ab(abc)abc(ab)a\rangle$, the history of frequent sequences for $\sigma = 2$ and $w = 4$ is illustrated by Figure 7. Using the proposed interval-based structure, the history for sequences a and ba are respectively $\mathcal{H}^{\sigma,w}(a) = \{[1,7]\}$ and $\mathcal{H}^{\sigma,w}(ba) = \{[2,3],[7,7]\}$.*

In this Figure 7, a black cell indicates that the sequential pattern for this line is frequent at the stream position in column. For instance, the sequential pattern $\langle ab\rangle$ were frequent only in the first window of the stream while sequencial patterns a and b were frequent all over the stream.

	1	2	3	4	5	6	7
a	■	■	■	■	■	■	■
ab	■						
b	■	■	■	■	■	■	■
c		■			■		
ba		■	■	■			■
(bc)		■					
bb				■			
aa			■				

FIG. 7 – *History of frequent sequences.*

6 Experiments and results

Insofar, as there is no pre-existing method that performs the same task as our algorithm named SEQ, we developed a second naïve algorithm, from now on named BAT. BAT performs the same task as SEQ, *i.e.* extracts the frequent sequences from a window sliding on a data stream, but non incrementally. This algorithm, based on PrefixSpan and using the PSP tree structure, rebuilds the entire tree $\mathcal{A}_\sigma(W)$ for each consecutive window of size ws on the data stream.

The algorithms were developed in C++ and executed on a processor at 2.2 GHz, with 2GB of RAM.

6.1 Experiments on simulated data

In this section, we compare the results obtained by SEQ with those of algorithm BAT on experiments over simulated data produced by the IBM data generator. This generator is usually used to generate transactions, but it can generate a single long transaction as well. The transaction will simulate a data stream input to algorithms. The results were obtained from 1200 executions performed by varying the parameters ws (window size), σ (minimal support) and ql (item vocabulary size, $card(\mathcal{E})$).

The curves below were obtained by averaging the results over all the experiments. For example, the point on the curve SEQ of Figure 8 - (a) for $\sigma = 5$ is obtained by averaging all the results of experiments running SEQ with $\sigma = 5$ when the other parameters vary freely.

We can note first that the memory usage of SEQ is slightly higher than BAT. This comes from the fact that SEQ makes use of merged trees ($\mathcal{A}^c$ in Figure 3) the size of which is a little larger than the basic tree of frequent sequences. The tree size decreases exponentially when the vocabulary size grows or the support threshold increases. We can observe this same trend for memory usage. Finally, we note in Figure 8-(f) that, on average, the window size ws has a low influence on the required memory.

Figures 8-(a), (c) and (e) show that the computation time of both algorithms is exponential with the window size ws, but it grows faster than exponential as σ or ql decreases. In such cases, the number of frequent sequences increases and trees are larger.

On average, the computation time of SEQ is 80 $\%$ lower than the computation time of BAT. Figure 8-(a) shows that the more σ decreases, the larger is the performance gap between the two algorithms. By contrast, Figure 8-(e) shows that this improvement decreases with the window size (77 $\%$ for $ws = 25$).

6.2 Experiments on smart electrical meter data

Smart electrical meters record the power consumption of an individual or company in intervals of 30 mn and communicate that “instant” information to the electricity provider for monitoring and billing purposes. The aim of smart meters is to better anticipate the high consumption of a distribution sector by awarding a consumption profile to each meter, that is to say, a dynamic model of changes in consumption. However, consumption profiles are not stable over time. Depending on the period of the year (seasons, holidays), of the week (weekdays, weekends) or of the day, consumption patterns change in a non-predictable manner. Consequently, there is no meter profile to predict medium to long-term consumption. Our algorithm can be used to extract, online, profiles of short-term consumption. Similar individual household electric power consumption may be downloaded from the UCI repository (see Frank and Asuncion (2010)).

The annual series of instantaneous consumption is a flow of about 18,000 values. We use the SAX algorithm (Lin et al. (2003)) to discretize the set of consumption values. A vocabulary size of $|\mathcal{E}| = 14$ and a PAA aggregation window $W = 24$ have been chosen. The consumption profile of a smart meter at time t is the set of frequent consumption sequences during the period $[t - w, t]$ (sliding window of predefined size $w = 28$ itemsets, *i.e.* 2 weeks).

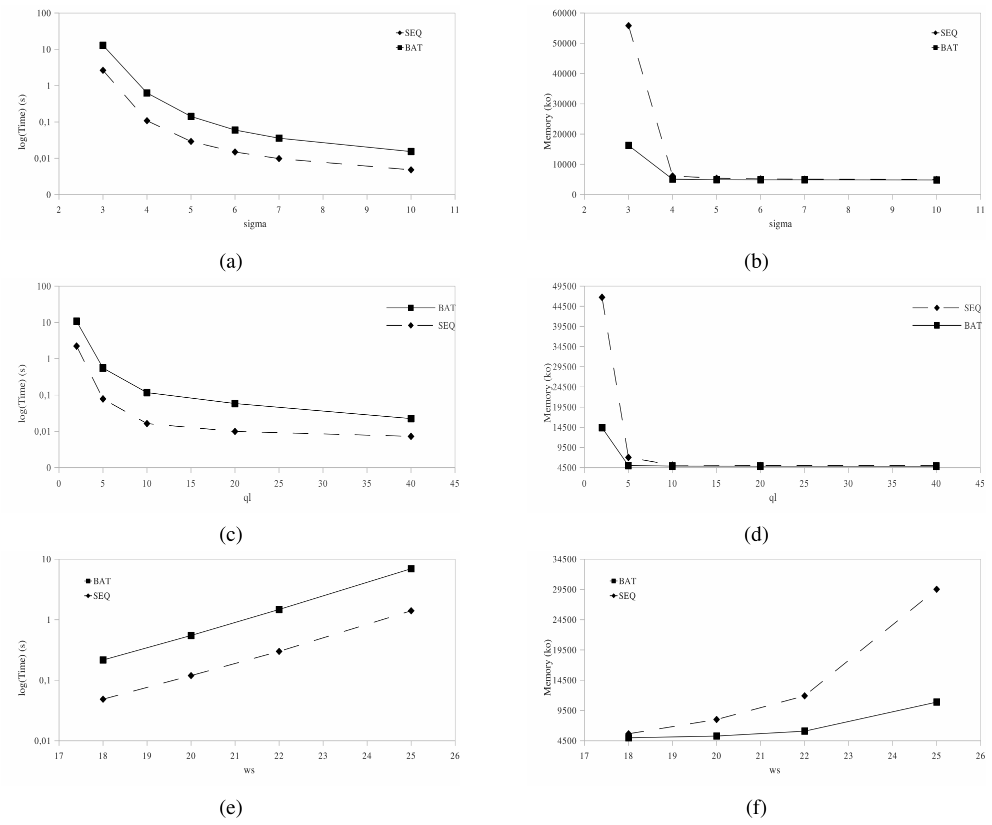

FIG. 8 – *Comparison of processing time (logarithmic scale) and memory usage with respect to the support threshold σ (with $ws < 25$), the number of symbols ql (with $\sigma > 3$) and the size of the sliding window ws.*

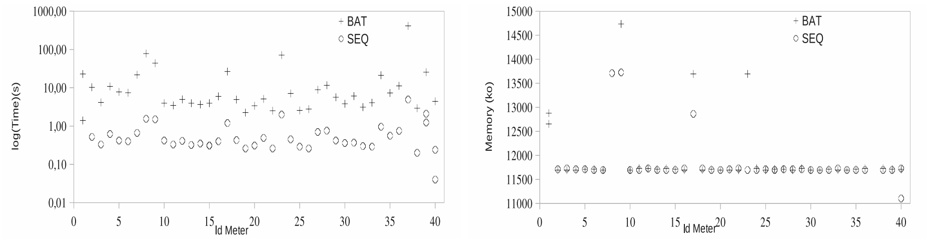

FIG. 9 – *Comparison of computation time (left) and memory usage (right) for the mining power consumption streams.*

Figure 9 shows the results for 40 meters. The results obtained on these real data are close to those obtained on simulated data. On the one hand, memory usage is slightly higher for the incremental algorithm but on the other hand, the processing time of SEQ is improved by 89%, on average, compared to BAT.

For some meters processing time is very long (about few minutes) while for most of the meters processing times are around seconds. This disparity is explained by the observed variability of consumption. The sequences that are difficult to process are quite constant (*e.g.* industrial consumption). These sequences include many symbol repetitions leading to many frequent sequences of repeated symbols.

7 Conclusion and perspectives

We have presented the problem of mining frequent sequences in a sliding window over a stream of itemsets. To address this problem, we have proposed an incremental algorithm that efficiently updates a tree data structure inspired by PSP. Our algorithm is based on counting minimal occurrences of a sequence in a window. The proposed algorithms are complete and correct.

Experiments conducted on simulated data and real instantaneous power consumption data show that our algorithm significantly improves the execution time of an algorithm based on a non-incremental naïve approach. For both algorithms, the time complexity is exponential with the size of the sliding windows and beyond exponential with respect to the number of items or the support threshold. These execution time performance were obtained with a memory usage close to the one of the naïve approach.

Future work will consider incremental mining of multiple data streams. In particular, the proposed tree representation of frequent sequences can be extended to design an algorithm that can extract frequent sequences in multiple itemsets streams and take into account the repetitions in each stream. Also, a lot research has been done on condensed representations of patterns e.g. closed patterns. We want to investigate such condensed representations in the context of incremental mining.

The frequent sequence history built from the stream of itemset proposes a new view on the stream. It would be insteresting to mine it in order to highlight concept changes. The adaptation of algorithms for mining interval-based sequences, such as the method proposed in Guyet and Quiniou (2011), to streams would be a possible solution but constitutes a great challenge.

References

Achar, A., S. Laxman, and P. S. Sastry (2010). A unified view of automata-based algorithms for frequent episode discovery. *CoRR abs/1007.0690.*

Agrawal, R., T. Imielinski, and A. Swami (1993). Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 207–216.

Ayres, J., J. Flannick, J. Gehrke, and T. Yiu (2002). Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 429–435. ACM.

Chang, J. H. and W. S. Lee (2003). Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 487–492.

Chen, G., X. Wu, and X. Zhu (2005). Sequential pattern mining in multiple streams. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, pp. 585–588.

Cheng, H., X. Yan, and J. Han (2004). IncSpan: incremental mining of sequential patterns in large database. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 527–532.

Ding, B., D. Lo, J. Han, and S.-C. Khoo (2009). Efficient mining of closed repetitive gapped subsequences from a sequence database. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pp. 1024–1035.

Ezeife., C. and M. Monwar (2007). A PLWAP-based algorithm for mining frequent sequential stream patterns. *International Journal of Information Technology and Intelligent Computing 2*(1), 89–116.

Frank, A. and A. Asuncion (2010). UCI machine learning repository.

Giannella, C., J. Han, J. Pei, X. Yan, and P. S. Yu (2003). Mining frequent patterns in data streams at multiple time granularities. In K. S. H. Kargupta, A. Joshi and Y. Yesha (Eds.), *Proceedings of the Next Generation Data Mining Workshop*.

Giannella, C., J. Han, J. Pei, X. Yan, and P. S. Yu (2004). *Next generation data mining*, Chapter Mining frequent patterns in data streams at multiple time granularities. AAAI/MIT Press.

Guyet, T. and R. Quiniou (2011). Extracting temporal patterns from interval-based sequences. In *Proceedings of International Join Conference on Artificial Intelligence*, pp. 1306–1311.

Ho, C.-C., H.-F. Li, F.-F. Kuo, and S.-Y. Lee (2006). Incremental mining of sequential patterns over a stream sliding window. In *IWMESD Workshop at ICDM*, pp. 677 –681.

Huang, J.-W., C.-Y. Tseng, J.-C. Ou, and M.-S. Chen (2008). A general model for sequential pattern mining with a progressive database. *IEEE Transactions on Knowledge and Data Engineering 20*(9), 1153 –1167.

Laxman, S., P. S. Sastry, and K. P. Unnikrishnan (2007). A fast algorithm for finding frequent episodes in event streams. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 410–419.

Li, H.-F. and S.-Y. Lee (2009). Mining frequent itemsets over data streams using efficient window sliding techniques. *Journal of Expert Systems with Applications 36*(2), 1466–1477.

Lin, J., E. Keogh, S. Lonardi, and B. Chiu (2003). A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the Workshop on Research Issues in Data Mining and Knowledge Discovery*.

Lin, J., E. Keogh, S. Lonardi, and P. Patel (2002). Finding motifs in time series. In *Proceedings of the 2nd Workshop on Temporal Data Mining*, pp. 53–68.

Lin, M.-Y. and S.-Y. Lee (2004). Incremental update on sequential patterns in large databases by implicit merging and efficient counting. *Journal of Information Systems 29*, 385–404.

Manku, G. S. and R. Motwani (2002). Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 346–357.

Mannila, H., H. Toivonen, and A. I. Verkamo (1997). Discovering frequent episodes in event sequences. *Journal of Data Mining and Knowledge Discovery 1*(3), 210–215.

Marascu, A.-M. and F. Masseglia (2006). Mining sequential patterns from data streams: a centroid approach. *Journal for Intelligent Information Systems 27*, 291–307.

Masseglia, F., F. Cathala, and P. Poncelet (1998). The PSP approach for mining sequential patterns. In *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, pp. 176–184.

Masseglia, F., P. Poncelet, and M. Teisseire (2003). Incremental mining of sequential patterns in large databases. *Journal of Data and Knowledge Engineering 46*, 97–121.

Méger, N. and C. Rigotti (2004). Constraint-based mining of episode rules and optimal window sizes. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pp. 313–324.

Nguyen, S. N., X. Sun, and M. E. Orlowska (2005). Improvements of IncSpan: Incremental mining of sequential patterns in large database. In *Proceedings of the 9th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining*, pp. 442–451.

Patnaik, D., N. Ramakrishnan, S. Laxman, and B. Chandramouli (2012). Streaming algorithms for pattern discovery over dynamically changing event sequences. *CoRR abs/1205.4477.*

Pei, J., J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu (2004). Mining sequential patterns by pattern-growth: the prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering 16*(11), 1424–1440.

Raïssi, C., P. Poncelet, and M.Teisseire (2006). Need for SPEED: Mining sequential pattens in data streams. In *Proceedings of Data Warehousing and Knowledge Discovery.*

Srikant, R. and R. Agrawal (1996). Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology*, pp. 3–17.

Tatti, N. and B. Cule (2011). Mining closed episodes with simultaneous events. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1172–1180.

Tatti, N. and B. Cule (2012). Mining closed strict episodes. *Data Mining and Knowledge Discovery 25*(1), 34–66.

Zaki, M. J. (2001). SPADE: An efficient algorithm for mining frequent sequences. *Journal of Machine Learning 42*(1/2), 31–60.

Résumé

We introduce the problem of mining frequent sequences of itemsets in a window sliding over a stream of itemsets. To address this problem, we present a complete and correct incremental algorithm based on a representation of frequent sequences inspired by the PSP algorithm and a method for counting the minimal occurrences of a sequence. The experiments were conducted on simulated data and on real instantaneous power consumption data. The results show that our incremental algorithm improves significantly the computation time compared to a non-incremental approach.