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

Thomas Guyet[1,2] **René Quiniou**[3]
[1] AGROCAMPUS OUEST, UMR6074 IRISA, F-35042 Rennes
[2] Université européenne de Bretagne
[3] INRIA Centre de Rennes - Bretagne Atlantique
thomas.guyet@agrocampus-ouest.fr rene.quiniou@inria.fr

Résumé

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

Abstract

We introduce the problem of mining frequent sequences 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 significantly improves the computation time compared to a non-incremental approach.

1 Introduction

Sequential pattern mining has been studied extensively in static contexts [17, 21, 23]. 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. [15] 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 antimonotonicity properties required for the effectiveness of pattern occurrences search (see [1]).

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 extract 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 is too costly in 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 [17], an improvement of GSP [23] for mining frequent itemset subsequences from a window sliding over a stream 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 present experimental results on simulated data as well as real data related to instantaneous power consumption. Finally, section 6 concludes the work and gives some perspectives.

2 Related work

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 [23], PSP [17], PrefixSpan [21], Spade [25], Spam [2], 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 [15], WinMiner [19], for instance, count the minimal occurrences of episodes in a given sequence, while Laxman et al. [10] 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 [24] introduced episodes with simultaneous events where multiple events may have the same timestamp, aiming at handling quasi-simultaneous events.

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 [14], tilted-time windows [7], damped windows [3] or sliding windows [11]. Few proposals have been made for sequential pattern mining over data streams. Chen et al. [4] 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 [22] 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 et al. [16] propose to cluster stream data to build a summary from which sequential patterns can be extracted. IncSpam [8] 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 aims 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.

Mining sequential patterns from data streams is similar to mining sequential patterns incrementally from dynamic databases. ISE [18] considers the arrival of new customers and new items and extends the set of candidate patterns from the original database accordingly. IncSP [13] uses efficient implicit merging and counting over appended sequences. IncSpan [5, 20] 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 [6] 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 [9] 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.

3 Basic concepts and problem statement

3.1 Items, itemsets and sequences

In the sequel, $\mathbb{N}_n$ denotes the set of the n first integers, *i.e.* $\mathbb{N}_n = \{1, \ldots, n\}$.

Let $(\mathcal{E}, =, <)$ be the set of items and $<$ a total order (*e.g.* lexicographical) on this set. An *itemset* $A = (a_1, a_2, ..., a_n),\ \forall i \in \mathbb{N}_n,\ a_i \in \mathcal{E}$ is an ordered set of distinct items, *i.e.* $\forall i \in \mathbb{N}_{n-1},\ a_i < a_{i+1}$ and $\forall i, j \in \mathbb{N}_n,\ i \neq j \Rightarrow a_i \neq a_j$. The size of an itemset α, denoted $|\alpha|$ is the number of items it contains.

An itemset $\beta = (b_i)_{i \in \mathbb{N}_m}$ is a sub-itemset of $\alpha = (a_i)_{i \in \mathbb{N}_n}$, denoted $\beta \sqsubseteq \alpha$, iff there exists a sequence of integers $1 \leq i_1 \leq i_2 \leq ... \leq i_m \leq n$ such that $\forall k \in \mathbb{N}_m, b_k = a_{i_k}$.

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 $1 \leq i_1 \leq i_2 \leq ... \leq i_m \leq n$ such that $\forall k \in \mathbb{N}_m, t_k \sqsubseteq s_{i_k}$.

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.*

Proposition 1. *The relation $\preceq$ is a partial order on the set of sequences.* [1]

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 $\forall k \in \mathbb{N}_n,\ t \leq i_k \leq t + w$ 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 $S = (s_i)_{i \in \mathbb{N}_n}$ of length n in a window $W = (w_i)_{i \in \mathbb{N}_m}$, 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 [15]).*

$$\mathcal{I}_W(S) = \{(i_j)_{j \in \mathbb{N}_n} \in \mathbb{N} \mid \begin{array}{ll} \forall j \in \mathbb{N}_n, s_j \sqsubseteq w_{i_j}, & (1) \\ \forall j \in \mathbb{N}_{n-1},\ i_j < i_{j+1}, & (2) \\ (w_j)_{j \in [i_1+1, i_n]} \not\preceq S, & (3) \\ (w_j)_{j \in [i_1, i_n-1]} \not\preceq S\ \} & (4) \end{array}$$

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.* $supp_W(S) = card\,(\mathcal{I}_W\,(S))$.

Proposition 2. *The support function $supp_W(\cdot)$ is anti-monotonic on the set of sequences with associated partial order $\preceq$.*

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 considers multiple parallel streams [8, 16, 22], the support of a sequence is usually defined as the number of streams in which this sequence appears. Here 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 relies on representing the set of frequent sequences in a tree, the structure of which is inspired by the prefixing method of PSP [17]. 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

1. We omit the proofs of propositions for space reasons.

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*
- $\alpha = (a_i)_{i \in \mathbb{N}_n}$ *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 $\beta = (b_i)_{i \in \mathbb{N}_{n+1}}$ of size $\|\alpha\|+1$ such that $\forall i \in \mathbb{N}_n,\ a_i = b_i$, (i.e. b_{n+1} is a strict successor of a_n),*
- $\mathcal{C}$ *is the set of descendant nodes which represent sequences $\beta = (b_i)_{i \in \mathbb{N}_n}$ of size $\|\alpha\|+1$ such that $\forall i \in \mathbb{N}_{n-1},\ a_i = b_i$, $a_n \sqsubseteq b_n$ 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).

Thanks to the anti-monotonicity property, we know that 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 a recursive processing of a PSP tree is complete and non-redundant.

Example 3. *Let $W = \langle a(bc)(abc)cb \rangle$ and $\sigma = 2$. Figure 1 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 1, the instance list of the sequence is displayed in the index. For example, the sequence $(bc)c$ has two instances : $\mathcal{I}(\langle (bc)c \rangle) = \{(2,3),\ (3,5)\}$.*

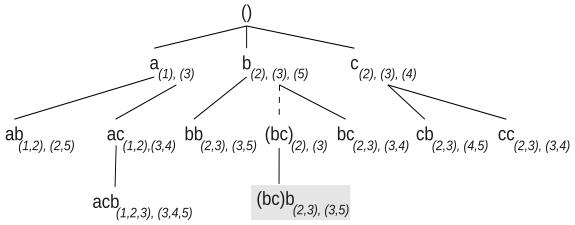

FIGURE 1 – Example of a tree of frequent sequences ($\sigma = 2$)

4.2 Illustration of the method

The incremental process aims at updating the tree of frequent sequences gathering the instances of sequences 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. The addition step brings 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 nodes of non-frequent sequences.

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

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

1. Deletion of the first itemset : all instances starting at the first (oldest) position of the window (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 $\mathcal{A}_2(\langle (bc)ab \rangle)$ where only (b) is frequent.

2. Merging the new current itemset (abc) **with every node of the tree of sequences** : 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 (abc) to a frequent sequence of $\langle (bc)ab \rangle$. In the tree representation of frequent sequences, this concatenation can be seen as extending each node of $\mathcal{A}_2(\langle (bc)ab \rangle)$ with the itemset tree $\mathcal{T}_{(abc)}$, *i.e.* the tree representing all sub-itemsets of (abc).

In Figure 2, the tree $\mathcal{T}_{(abc)}$ is merged with two nodes of $\mathcal{A}_2(\langle (bc)ab \rangle)$:
- with the root node (orange instances) : all subsequences of (abc) become potentially frequent.

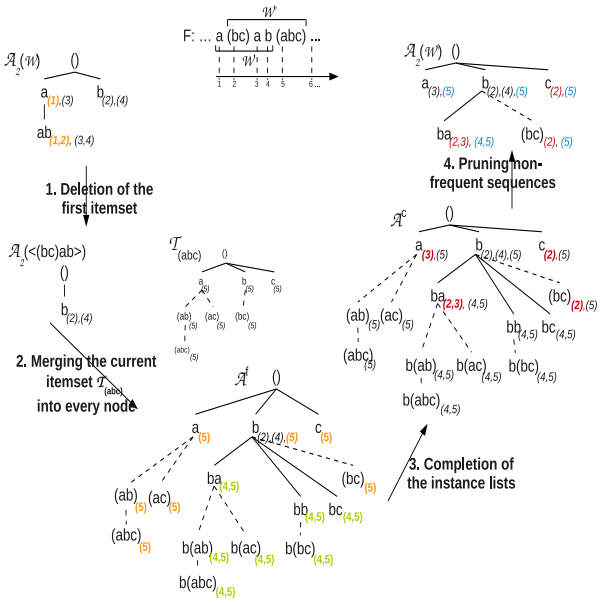

FIGURE 2 – Successive steps for updating the tree of sequences upon the arrival of itemset (abc) in the window $W = \langle a(bc)ab \rangle$.

– with the node (b) (green instances) : all sequences starting with (b) (frequent in $\langle (bc)ab \rangle$) and followed by a sub-itemset of (abc) 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 $\{(2), (4), (5)\}$. We know that each of these nodes holds all the instances of the associated sequence in W'. For each new node of $\mathcal{A}^f$, *e.g.* the node (a), the list has only one instance made of a sub-itemset of (abc).

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 (a) is associated to the instance list $\{(5)\}$. This list must be completed with the list of instances of (a) in the previous window in which (a) was unfrequent, *i.e.* $\{(3)\}$. Red nodes of the tree $\mathcal{A}^c$ in Figure 2 show the instances added by completion.

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

4.3 Merging an itemset tree into a tree of sequences

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.

Algorithm 1 describes how the itemset tree $\mathcal{T}$ is merged with every node of the tree of sequences $\mathcal{A}$. It consists in two main steps :

– 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 2).

Let $N.\alpha$ be the sequence associated with node N from the tree of sequences $\mathcal{A}$ and $N.\mathcal{I}$ be 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.

In a second step, the algorithm recursively merges the root of the itemset tree $\mathcal{T}$ prefixed by N. Algorithm 2 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. $n.\mathcal{C}$).

Algorithme 1 : `Merging` : merging the itemset tree $\mathcal{T}$ with every node of the tree of sequences $\mathcal{A}$.

```
   input  : A, T
   output : A^f : merged tree
1  for N ∈ A do
2      //Prefixing T
3      for n ∈ T do
4          //Prefixing the sequence with N.α
5          n.α = N.α ⊕ n.α
6          //Prefixing the instances with the last element of
           N.I, noted d
7          for I ∈ n.I do
8              I = d ∪ I
9      //Recursive merging of T with nodes N of A
10     RecMerge(N, T)
```

Algorithme 2 : `RecMerge` : recursively merging the prefixed itemset tree $\mathcal{T}$ with a node of $\mathcal{A}$

input : n : itemset node tree, N : node of the tree of sequences to be merged with n and such that $n.\alpha = N.\alpha$

```
1  //Merging instance lists :
2  N.I = N.I ∪ n.I
3  //Recursion :
4  for s_N ∈ N.S ∪ N.C do
5      for s_n ∈ n.S ∪ n.C do
6          if s_N.α = s_n.α then
7              found =true
8              RecMerge(s_n, s_N)
9      if not found then
10         if s_n ∈ n.S then N.S=N.S ∪ {Copy (s_n)}
11         else N.C=N.C ∪ {Copy (s_n)}
```

4.4 Instance lists completion

When a new sequence is introduced in the tree, it is quite possible that other instances of this sequence were present in the previous window (corresponding to the beginning of the current window) but unfrequent and, so, they were not stored in the tree. For example, in Figure 2, the sequence $\langle ab \rangle$ is not frequent in W and not present in the tree of sequences $\mathcal{A}_2(W)$. However, after the arrival of itemset (abc) the sequence $\langle ab \rangle$ becomes frequent in W'. Thus, it is necessary to scan W to find all instances of $\langle ab \rangle$ to assess its frequency.

The completion algorithm is applied only to the nodes newly introduced in the tree $\mathcal{A}^f$. While ensuring the completeness, this method limits the number of completions.

To make the completion efficient, the instance list of a sequence β is recursively constructed from the list of its direct parent along the following principles :

- each instance $I = (i_1, ..., i_{|\delta|})$ of a sequence δ obtained by adding an item e to the last itemset of β (composition) are necessarily instances of β, thus the algorithm tests whether e is included in the itemset $W_{i_{|\delta|}}$.
- each instance $I = (i_1, ..., i_{|\epsilon|})$ of a sequence ϵ, obtained by adding an itemset e to β (succession), are necessarily constructed by adding the element $i_{|\epsilon|}$ to an instance of β, thus the algorithm browses a sub-sequence of W to test the presence of e.

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.

For example, on tree $\mathcal{A}^f$ in Figure 2, the instance list of $\langle (bc) \rangle$ is $\mathcal{I}(\langle (bc) \rangle) = \{(5)\}$. The single instance of $\mathcal{I}(\langle (bc) \rangle)$ is one of the instances of $\langle b \rangle$: $\mathcal{I}(\langle b \rangle) = \{(2), (4), (5)\}$. $(4, 5)$ is an instance of $\langle bc \rangle$ which is obtained by adding the element 5 to the instance (4) of sequence $\langle b \rangle$. To complete the instance (2) from $\mathcal{I}(\langle b \rangle)$, the algorithm looks for one occurrence of c in W in position between 3 (=2+1) and 4, the second instance of $\mathcal{I}(\langle b \rangle)$.

Proposition 3. *The algorithm is complete and correct : the tree of sequences constructed for each window W on the stream is $\mathcal{A}_\sigma(W)$.*

5 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, named BAT in the sequel. 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, rebuilts 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.

5.1 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 by performing 1200 executions by varying the parameters ws (window size), σ (minimal support) and ql (item vocabulary size, $card(\mathcal{E})$).

The curves below are obtained by averaging the results over all the results. For example, the point on the curve SEQ of Figure 3 - (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 ($\mathcal{A}^f$ in Figure 2) 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 3-(f) that, on average, the window size ws has a low influence on the required memory.

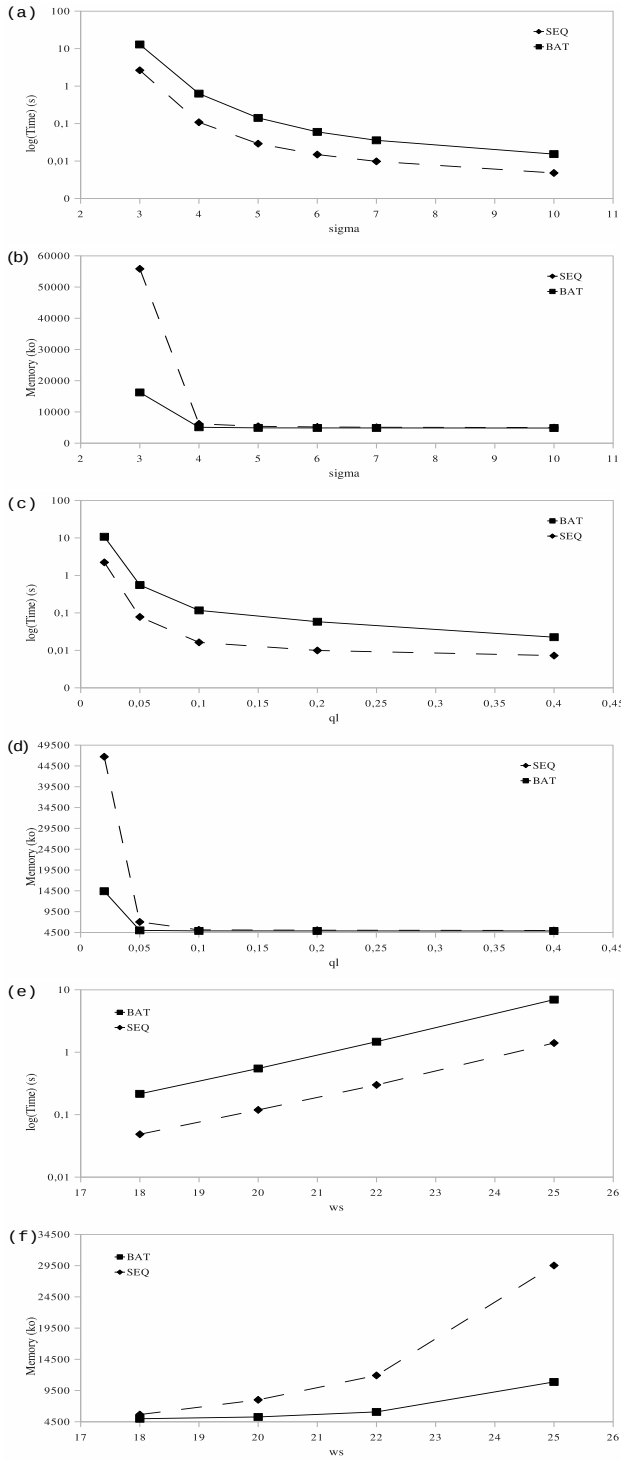

FIGURE 3 – Comparison of processing time (logarithmic scale) and memory usage with respect to the support threshold σ (with $ws < 25$), the number of symbols ql (with $\sigma > 3$) and the size of the sliding window ws.

Figures 3-(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 3-(a) shows that the more σ decreases, the better the performance gap between the two algorithms. By contrast, Figure 3-(e) shows that this improvement decreases with the window size (77 % for $ws = 25$).

5.2 Experiments on smart electrical meter data

Smart electrical meters record the power consumption of an individual or company in intervals of 30 mn and communicates 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 year (seasons, holidays), week (weekdays, weekends) or 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.

The annual series of instantaneous consumption is a flow of about 18,000 values. We use the SAX algorithm [12] for discretizing the consumption values. A vocabulary size of $|\mathcal{E}| = 14$ and a window aggregation of $PAA = 24$ 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).

Figure 4 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 repetitions of symbol leading to a high number of frequent sequences.

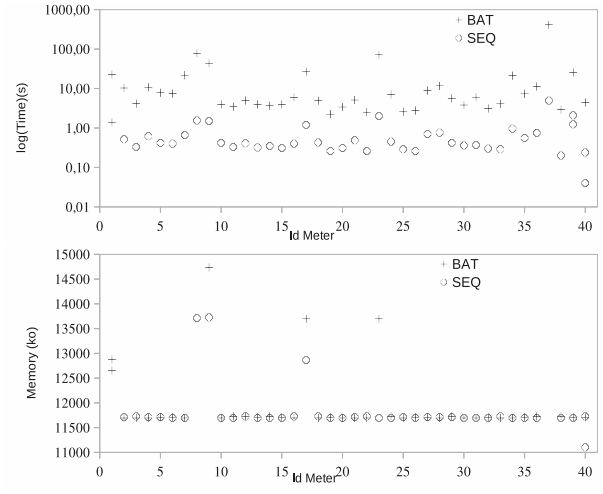

FIGURE 4 – Comparison of computation time (top) and memory usage (bottom) for the mining power consumption streams.

6 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 computation time is exponential with the size of the sliding windows and beyond exponential complexity with respect to the number of items or the support threshold. These execution time performance are obtained with a memory usage close to the one of the naïve approach. The results of these experiments have to be confirmed by a theoretical analysis of algorithms.

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 taking into account the repetitions in each stream. Also, a lot research has been made on condensed representations of patterns e.g. closed patterns. We want to investigate such condensed representations in the context of incremental mining.

Acknowledgements

We thank the project ICAME from EDF-R&D for providing smart meter consumption data.

Références

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

[2] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *KDD*, pages 429–435. ACM, 2002.

[3] J. H. Chang and W. S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *KDD*, pages 487–492, 2003.

[4] G. Chen, X. Wu, and X. Zhu. Sequential pattern mining in multiple streams. In *ICDM*, pages 585–588, 2005.

[5] H. Cheng, X. Yan, and J. Han. IncSpan : incremental mining of sequential patterns in large database. In *KDD*, pages 527–532, 2004.

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

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

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

[9] Jen-Wei Huang, Chi-Yao Tseng, Jian-Chih Ou, and Ming-Syan Chen. A general model for sequential pattern mining with a progressive database. *Knowledge and Data Engineering, IEEE Transactions on*, 20(9) :1153 –1167, 2008.

[10] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *KDD*, pages 410–419, 2007.

[11] H.-F. Li and S.-Y. Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Syst. Appl.*, 36(2) :1466–1477, 2009.

[12] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. 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*, 2003.

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

[14] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[15] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. *Data Mining and Knowledge Discovery*, 1(3) :210–215, 1997.

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

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

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

[19] N. Méger and C. Rigotti. Constraint-based mining of episode rules and optimal window sizes. In *PKDD*, pages 313–324, 2004.

[20] S. N. Nguyen, X. Sun, and M. E. Orlowska. Improvements of IncSpan : Incremental mining of sequential patterns in large database. In *PAKDD*, pages 442–451, 2005.

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

[22] C. Raïssi, P. Poncelet, and M.Teisseire. Need for SPEED : Mining sequential pattens in data streams. In *Proceedings of DAWAK*, 2006.

[23] R. Srikant and R. Agrawal. Mining sequential patterns : Generalizations and performance improvements. In *Proceedings of the 5th Int. Conf. on Extending Database Technology*, pages 3–17, 1996.

[24] N. Tatti and B. Cule. Mining closed episodes with simultaneous events. In *KDD*, pages 1172–1180, 2011.

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