SPAMS : a novel Incremental Approach for Sequential Pattern Mining in Data Streams

Lionel VINCESLAS, Jean-Emile SYMPHOR, Alban MANCHERON and Pascal PONCELET

Abstract Mining sequential patterns in data streams is a new challenging problem for the datamining community since data arrives sequentially in the form of continuous rapid and infinite streams. In this paper, we propose a new on-line algorithm, SPAMS, to deal with the sequential patterns mining problem in data streams. This algorithm uses an *automaton-based structure* to maintain the set of frequent sequential patterns, *i.e.* SPA (Sequential Pattern Automaton). The sequential pattern automaton can be smaller than the set of frequent sequential patterns by two or more orders of magnitude, which allows us to overcome the problem of combinatorial explosion of sequential patterns. Current results can be output constantly on any user's specified thresholds. In addition, taking into account the characteristics of data streams, we propose a well-suited method said to be approximate since we can provide near optimal results with a high probability. Experimental studies show the relevance of the SPA data structure and the efficiency of the SPAMS algorithm on various datasets. Our contribution opens a promising gateway, by using an automaton as a data structure for mining frequent sequential patterns in data streams.

Lionel VINCESLAS
CEREGMIA, Université des Antilles et de la Guyane, Martinique - France
e-mail: `lionel.vinceslas@martinique.univ-ag.fr`

Jean-Emile SYMPHOR
CEREGMIA, Université des Antilles et de la Guyane, Martinique - France
e-mail: `je.symphor@martinique.univ-ag.fr`

Alban MANCHERON
LIRMM, 161 rue Ada 34392 Montpellier CEDEX 5 - France
e-mail: `alban.mancheron@lirmm.fr`

Pascal PONCELET
LIRMM - UMR 5506 - 161 rue Ada 34392, Montpellier Cedex 5 - France
e-mail: `pascal.poncelet@lirmm.fr`

1 Introduction

Concerned with many applications (*e.g.* medical data processing, marketing, safety and financial analysis), mining sequential patterns is a challenging problem within the datamining community. More recently these last years, many emerging applications, such as traffic analysis in networks, web usage mining or trend analysis, generate a new type of data, called data streams. A data stream is an ordered sequence of transactions, potentially infinite, that arrives in a timely order. The characteristics of a data stream can be expressed as follows (*cf.* Lin 2005) :

- **Continuity.** Data continuously arrive at a high speed.
- **Expiration.** Data can be read only once.
- **Infinity.** The total amount of data is unbounded.

Therefore, mining in data streams should meet the following requirements as well as possible. Firstly, owing to the fact that past data cannot be stored, the methods can provide approximate results but accuracy guarantees are required. Secondly, the unbounded amount of data supposes that the methods are adjustable according to the available ressources, especially for the memory. Lastly, a model is needed which adapts itself to continuous data stream over a time period.

Previous work.

Initially, the first work deal with the case of static databases and propose exact methods for mining sequential patterns. We can quote as an example, the algorithms GSP, PSP, FreeSpan, SPADE, PrefixSpan, SPAM and PRISM, respectively proposed by [14, 11, 6, 15, 12, 2, 5]. Thus, the first algorithms mentioned above for mining sequential patterns are not adapted any more in the context of data streams. In [13], authors propose to use sampling techniques for extracting sequential patterns in data streams. Nevertheless, the context is quite different from our proposal since they mainly focus on a summarization of the stream by using a reservoir sampling-based approach. In that case, the sampling could be considered as a static database and then any sequential pattern mining algorithm can be applied. It was shown in [4], that methods, said to be approximate, are well adapted to the context of data streams. However, the principal difficulty resides in the search of a trade-off between time and memory performances, and the quality of the mining results as well as in recall as in precision. So far, the literature concerning the mining of sequential patterns in data streams is relatively poor. In [3], the authors proposed the algorithm eISeq, using a tree-based data structure. This algorithm is a one pass algorithm, which processes the stream sequentially, transaction per transaction. However, the longer the sequential patterns are, the less this algorithm is performant. That is due to the generation of all the sequential sub-patterns which increase exponentially. For example, if $< a1, \cdots, ai >$ is a sequential pattern, there are $(2^i - 1)$ sequential sub-patterns to be created. To alleviate this difficulty , the GraSeq algorithm have been presented in [9]. Their approach is based on an oriented graph data structure to limit the sequential-sub-patterns generation phase. However, this approach supposes a costly pre-processing step for regrouping the transactions.

Our contribution.

In this paper, we propose a new one-pass algorithm: SPAMS (Sequential Pattern Automaton for Mining Streams). SPAMS is based on the on-line and the incremental building and updating of an automaton structure : the SPA. The SPA (Sequential Patterns Automaton) is a deterministic finite automaton, which indexes the frequent sequential patterns in data streams. The remainder of the paper is organized as follows. Section 2 states the problem formally. In Section 3, we recall some prerequisite preliminary concepts and we present our approach in section 4. Experimental studies are provided in the section 5 and the conclusion is presented in the last section.

2 Problem Definition

In this section, we give the formal definition of the problem of mining sequential patterns in data streams. First, we give a brief overview of the traditional sequence mining problem by summarizing the formal description introduced in [14]. Second, we examine the problem when considering streaming data. Let $\mathbb{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called items and let DB a database of customer transactions where each transaction T consists of customer-id, transaction time and a set of items involved in the transaction. An itemset is a non-empty set of items. A sequential pattern s is a set of itemsets ordered according to their timestamp and is denoted by $< s_1 s_2 \cdots s_n >$, where s_j, for $j \subseteq [1..n]$, is an itemset. A k sequential pattern is a sequential pattern of k items or of length k. A sequential pattern $S' =< s'_1 s'_2 \cdots s'_n >$ is a sub-pattern of another sequential pattern $S =< s_1 \; s_2 \; \cdots \; s_n >$, denoted $S' \prec S$ if there exists integers $i_1 < i_2 < \cdots i_j \cdots < i_n$ such that $s'_1 \subseteq s_{i_1}$, $s'_2 \subseteq s_{i_2}$, $\cdots$, $s'_n \subseteq s_{i_n}$. All transactions from the same customer are grouped together and sorted in an increasing order and are called a data sequence. A support value (denoted supp(S)) for a sequential pattern gives its number of actual occurrences in DB. Nevertheless, a sequential pattern in a data sequence is taken into account only once to compute the support even if several occurrences are discovered. A data sequence contains a sequential pattern S if S is a sub-pattern of the data sequence. In order to decide whether a sequential pattern is frequent or not, a minimum support value (denoted σ) is specified by the user, and the sequential pattern is said to be θ-frequent if $supp(S) \geq \sigma$, where $\sigma = \ulcorner \theta \times |DB| \urcorner$ with $\theta \in]0;1]$ and $|DB|$ the size of the database. Given a database of customer transactions, the problem of sequential pattern mining is to find all the sequential patterns whose support is greater than a specified threshold minimum support. Extended to the case of data streams, this problem can be expressed as follows. Formally, a data stream DS can be defined as a sequence of transactions, $DS = (T_1, T_2, \cdots, T_i, \cdots)$, where T_i is the i-th arrived transaction. Each transaction, identified by a Tid, is associated with an Cid identifier (cf. the example in the table 1). Mining frequent sequential patterns remains to find all the sequential patterns, whose support value is equal or greater than the fixed minimum support threshold for the known part of the data stream at a given time.

Table 1 A data sequence built on $\mathbb{I} = \{a, b, c, d\}$.

C_1	$< (b,d)\ (a,b,d)\ (a,c,d) >$
C_2	$< (b,c,d)\ (b,d) >$
C_3	$< (a,b)\ (c) >$

3 Prerequisites on statistical covers

We briefly present in this section the required theoretical materials on statistical covers that we have presented in [8]. So, we recall the following theorem.

Theorem 1. *$\forall \theta, 0 < \theta \leq 1, \forall \delta, 0 < \delta \leq 1$, we denote by m and m^* respectively the (θ-frequent and θ-infrequent) number of sequential patterns in the known part of the stream and in the whole stream. If we choose ϵ such that:*

$$\epsilon \geq \sqrt{\frac{1}{2m} \ln \frac{m^*}{\delta}},$$

then `Recall` $= 1$ and respectively `Precision` $= 1$ with a probability of at least $1 - \delta$, when discarding all the sequential patterns that are not θ'-frequent from the observation, where $\theta' = \theta - \epsilon$ and respectively $\theta' = \theta + \epsilon$.

The parameter δ is the statistical risk parameter potentially fixed by the user and the values $\theta' = \theta \pm \epsilon$ are the statistical supports.

The **sup-(θ, ϵ)-cover** is the near-optimal smallest set of sequential patterns with a probability of at least $1 - \delta$) containing all the sequential patterns that are θ-frequent in the whole stream (eventually infinite). There are no false negative results with high probability. The **inf-(θ,ϵ)-cover** is the near-optimal biggest set of sequential patterns with a probability of at least $1-\delta$) containing only sequential patterns that are θ-frequent in the whole stream (eventually infinite). In this set, there are no false positive results with high probability, but false negative ones. We provided the proof of this theorem in [8]. By near-optimal, we express that any technique obtaining better bounds is condemned to make mistakes (the criterion to be maximized is not equal any more to 1). We precised also, that there is no assumption on the distribution of the stream.

4 The SPAMS approach

Our approach is based on the incremental construction of an automaton which indexes all frequent sequential patterns from a data stream. For the mining process, we do not make the assumption of an ideal data stream where transactions are sorted by customers. In fact, we make no assumptions either on the order of data, or on customers, or on the alphabet of the data. It's a real incremental approach for knowledge discovery in data streams. Moreover, to obtain the best quality of approximation, in both recall and precision, we also index the $(\theta-\epsilon)$-frequent sequential patterns of the statistical cover, in addition to those θ-frequent. In this way, we retain the minimum number of candidates, which limits the combinatorial explosion.

4.1 SPA : the Sequential Patterns Automaton

In a more formal way, we define in this section the automaton of sequential patterns, SPA. For further information on the automata theory, we suggest the presentation made by [7].

Definition 1 (Finite state automaton). A finite state automaton $\mathcal{A}$ is a 5-tuple such that $\mathcal{A} = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$, where $\mathcal{Q}$ is a finite set of states, Σ an input alphabet, $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is a set of transitions, $\mathcal{I} \subseteq \mathcal{Q}$ and respectively $\mathcal{F} \subseteq \mathcal{Q}$ are the set of initials and finals states.

Definition 2 (Deterministic finite state automaton). A finite state automaton $\mathcal{A} = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$ is deterministic if and only if it exists a unique initial state (*i.e.* $|\mathcal{I}| = 1$) and if $\forall\ p,q \in \mathcal{Q}$ and $\alpha \in \Sigma,\ \ (p,\alpha,q),(p,\alpha,q') \in \delta \Rightarrow q = q'$.

The label of a transition t going from a state q_i to a state q_j, denoted $t = q_i \overset{\alpha}{\longmapsto} q_j$ is the symbol α. A path in $\mathcal{A}$ is a sequence $c = t_1, \cdots, t_n$ of consecutive transitions. The label of a path c is denoted $|c| = \alpha_1 \cdots \alpha_n$, or $c : q_0 \overset{w}{\longmapsto} q_n$ with $w = |c|$. A label is also called a *word*. A path $c : q_i \overset{w}{\longmapsto} q_j$ is said to be *successful* if and only if $q_i \in \mathcal{I}$ and $q_j \in \mathcal{F}$. A word w is said to be *accepted* or *recognised* by the automaton $\mathcal{A}$ if it is the label of a successful path.

Definition 3 (Language accepted by a DFA). Let $\mathcal{A} = (\mathcal{Q},\, q_0,\, \mathcal{F},\, \Sigma,\, \delta)$ be a deterministic finite state automaton (DFA). The *language accepted* or recognised by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all accepted words :

$$\mathcal{L}(\mathcal{A}) = \left\{ w \subseteq \Sigma^* \mid \exists\, c : q_0 \xmapsto{w} q_j,\; q_j \in \mathcal{F} \right\}$$

Definition 4 (The Sequential Patterns Automaton). The sequential patterns automaton (SPA) is a deterministic finite state automaton, *i.e.* a 5-tuple $SPA = (\mathcal{Q},\, q_0,\, F,\, \Sigma,\, \delta)$, whose accepted language $\mathcal{L}(\mathcal{SPA})$ is the set of frequent sequential patterns.

Definition 5 (The sequence item). Let $SPA = (\mathcal{Q},\, q_0,\, F,\, \mathbb{I},\, \delta)$ be the automaton of sequential patterns. We add to the set Σ, a special item called the *sequence item*, denoted arbitrarily #. This item is an item that separates itemsets within sequential patterns (*cf.* figure 1).

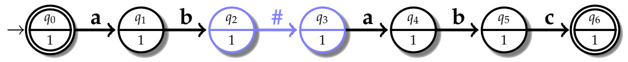


Fig. 1 An automaton indexing the sequential pattern $<(a,b)(a,b,c)>$

Definition 6 (The sink state). Let $SPA = (\mathcal{Q},\, q_0,\, F,\, \mathbb{I},\, \delta)$ be the automaton of sequential patterns. We add to the set $\mathcal{Q}$, a special state called the *sink state*, denoted q_∞. It's a temporary state used by the transition function to generate the other states of the automaton.

Definition 7 (Support of a state). Let $SPA = (\mathcal{Q},\, q_0,\, F,\, \mathbb{I},\, \delta)$ be the automaton of sequential patterns, and $q \in \mathcal{Q}$, a final state. We define the *support of the state* q, denoted $|q|$, as an integer representing the support of sequential patterns recognised in this state.

Lemma 1. *Let $\mathcal{L}_q \subseteq \mathcal{L}(SPA)$ be the set of words (i.e. sequential patterns) recognised in the state $q \in Q$. According to definition 7, the following assertion is definitely obvious :*

$$\forall\, w_i, w_j \in \mathcal{L}_q \subseteq \mathcal{L}(SPA) \;\; (1 \le i, j \le |\mathcal{L}_q|) \;\;,\; supp(w_i) = supp(w_j)$$

Property 1. Let $SPA = (\mathcal{Q},\ q_0,\ F,\ \Sigma,\ \delta)$ be the sequential patterns automaton :

$$\forall q_i \overset{\alpha}{\longmapsto} q_j \in SPA \ \ (q_i, q_j \in \mathcal{Q},\ \alpha \in \Sigma)\ ,\ \ |q_i| \geq |q_j|$$

Proof. *Let $c_1 : q_0 \overset{w}{\longmapsto} q_i$ and $c_2 : q_i \overset{\alpha}{\longmapsto} q_j \in SPA$ be two paths ($\alpha \in \Sigma$; $w \in \Sigma^*$). According to the Apriori property [1] (i.e. for any frequent itemset, all sub-itemsets are frequent), if $z = w \cdot \alpha$ is the label of a successful path $c_3 : q_0 \overset{z}{\longmapsto} q_j$, then c_1 is also a successful path and $supp(w) \geq supp(z)$. According to definition 7, $supp(w) = |q_i|$ and $supp(z) = |q_j|$.* ***This shows that*** $|\mathbf{q_i}| \geq |\mathbf{q_j}|$

Property 2. Let $SPA = (\mathcal{Q},\ q_0,\ F,\ \Sigma,\ \delta)$ be the sequential patterns automaton, $\mathcal{R}(\mathcal{Q},\alpha)$ be the set of reachable states by α and $\mathcal{R}(\mathcal{Q},\beta)$ be the set of reachable states by β :

$$\forall \alpha, \beta \in \Sigma\ ,\ \ \mathcal{R}(\mathcal{Q},\alpha) \cap \mathcal{R}(\mathcal{Q},\beta) = \emptyset$$

4.2 The SPAMS algorithm

4.2.1 Notations

In the following, we define some of the notations used in SPAMS :

- ⋄ $\mathcal{T}$ is the set of transition of the automaton.
- ⋄ $\mathcal{T}_s$ is the set of *ingoing transitions* on the state $s \in \mathcal{Q}$.
- ⋄ $|\mathcal{T}_s|$ is the number of *ingoing transitions* on the state $s \in \mathcal{Q}$.
- ⋄ $\mathcal{Q}^{\#}$ is the set of *reachable states* by the *sequence item*.
- ⋄ $\mathcal{Q}_{\texttt{cid}}$ is the set of reachable states for a customer id `cid`.
- ⋄ $\mathcal{T}^r$ is the set of *reachable transitions*, *i.e.* transitions labelled by the item being processed.
- ⋄ $\mathcal{C}$ is the set of customers id.
- ⋄ $\mathcal{C}_s$ is the set of customers id for a state $s \in \mathcal{Q}$, *i.e.* the customers whose indexed sequential patterns use the state s.

4.2.2 Presentation

According to definition 7, a state may recognise several sequential patterns whose support is the same. So, if the support of one or more sequential patterns recognised in a state q, has to change (*i.e.* their support is incremented by 1), the definition 7 is no longer respected. To resolve this problem, we make a copy q' of the state q : all sequential patterns recognised in the state q are not moved. We move only on the state q', the sequential patterns whose support has changed. This is done by a movement of some ingoing transitions from the state q to the state q'. It is evident that all sequential patterns recognised in the state q' have the same support (*cf.* definition 7). Finally, we create the same outgoing transitions of the state q for the state q'.

Our algorithm is divided into three main modules which are Insert, Prune and Next.

The Insert module : This module is called by the SPAMS algorithm for each item read from the data stream. Let *cid* be the customer id, and $\alpha \in \Sigma$ the item being processed. This module is responsible for the creation of new transitions in the automaton, and therefore of the application of definition 7. So, the Insert module will try to create all necessary transitions of the form $s \overset{\alpha}{\longmapsto} s'$. Therefore, we need to know the corresponding states s and s'. The state s is obtained by scanning the list of reachable states for the customer id *cid*, denoted $\mathcal{Q}_{cid}$. This means each customer id has its own set of reachable states. We proceed in the following way :

- ◇ First, if this customer id is new ($cid \notin \mathcal{C}$), we update the following sets : $\mathcal{C} = \mathcal{C} \cup \{ cid \}$, $\mathcal{Q}_{cid} = \{ q_0 \}$ and $\mathcal{C}_{q_0} = \mathcal{C}_{q_0} \cup \{cid\}$.
- ◇ Then, for each state $s \in \mathcal{Q}_{cid}$, if there is no state $s' \in \mathcal{Q}$ such that the transition $s \overset{\alpha}{\longmapsto} s'$ exist, we create a new transition to the sink state ($\mathcal{T} = \mathcal{T} \cup \{s \overset{\alpha}{\longmapsto} q_\infty\}$) and update the set : $\mathcal{T}^r = \mathcal{T}^r \cup \{s \overset{\alpha}{\longmapsto} q_\infty\}$. Otherwise, if the transition $s \overset{\alpha}{\longmapsto} s'$ exists, we update the set : $\mathcal{T}^r = \mathcal{T}^r \cup \{s \overset{\alpha}{\longmapsto} s'\}$.
- ◇ For each state s' such that $s \overset{\alpha}{\longmapsto} s' \in \mathcal{T}^r$, we make the following step :

  - If the state $\mathsf{s}' \neq \mathsf{q}_\infty$ and $|\mathcal{T}_{\mathsf{s}'}| = |\mathcal{T}_{\mathsf{s}'} \cap \mathcal{T}^{\mathsf{r}}|$, then :
    1. we update the set : $\mathcal{Q}_{cid} = \mathcal{Q}_{cid} \cup \{ s' \}$.
    2. if the customer id $cid \notin \mathcal{C}_{s'}$, then $|s'| = |s'| + 1$ and we update the set : $\mathcal{C}_{s'} = \mathcal{C}_{s'} \cup \{ cid \}$.
    3. if $|s'| < min\_sup$, we call the prune module : Prune(s')
  - Otherwise (*i.e.* $s' = q_\infty$ or $|\mathcal{T}_{s'}| \neq |\mathcal{T}_{s'} \cap \mathcal{T}^r|$) :
    1. we create a new state p and update the set : $\mathcal{Q}_{cid} = \mathcal{Q}_{cid} \cup \{ p \}$.
    2. we also update the set : $\mathcal{C}_p = \mathcal{C}_{s'} \cup \{ cid \}$
    3. if the customer id $cid \notin \mathcal{C}_{s'}$, then $|p| = |s'| + 1$, otherwise $|p| = |s'|$
    4. if the item α is the sequence item, we update the set : $\mathcal{Q}^\# = \mathcal{Q}^\# \cup \{ p \}$

5. for each ingoing transition $s \overset{\alpha}{\longmapsto} s' \in \mathcal{T}^r$, we delete it and create the ingoing transition $s \overset{\alpha}{\longmapsto} p : \mathcal{T} = \mathcal{T} \setminus \left\{ s \overset{\alpha}{\longmapsto} s' \right\} \cup \left\{ s \overset{\alpha}{\longmapsto} p \right\}$
6. for each outgoing transition $s' \overset{\beta}{\longmapsto} s'' \in \mathcal{T}$ $(\beta \in \Sigma,\ s'' \in \mathcal{Q})$, we create the same outgoing transition for the state $p : \mathcal{T} = \mathcal{T} \cup \left\{ p \overset{\beta}{\longmapsto} s'' \right\}$
7. if $|p| < min\_sup$, we call the prune module : PRUNE(p)

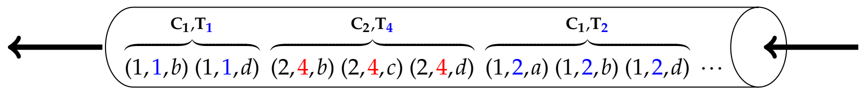
◇ We update the set of reachable transitions : $\mathcal{T}^r = \emptyset$

The PRUNE module : This module is called by the INSERT module in order to prune a state from the automaton. Not only does it erases the concerned state but also the states and transitions reachable from itself.

The NEXT module : When the module INSERT has processed all items of a transaction, for a given customer id (*cid*), the module NEXT is called.
This module works as follows :

1. We save the set $\mathcal{Q}_{cid} : \mathcal{Z} = \mathcal{Q}_{cid}$
2. We update the set $\mathcal{Q}_{cid} : \mathcal{Q}_{cid} = \mathcal{Q}_{cid} \setminus \{\mathcal{Q}_{cid} \cap \mathcal{Q}^- \cup \{q_0\}\}$
3. We call the module INSERT giving as parameters the customer id (*cid*) and the sequence item (#).
4. We update the set $\mathcal{Q}_{cid} : \mathcal{Q}_{cid} = \mathcal{Z} \cap \mathcal{Q}^\# \cup \{q_0\}$

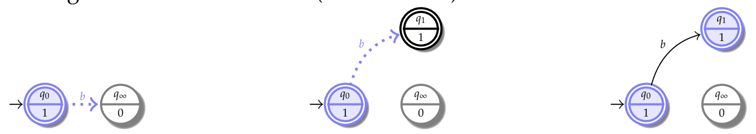
Fig. 2 Example of an unordered data stream generated from table 1



4.2.3 An example of construction

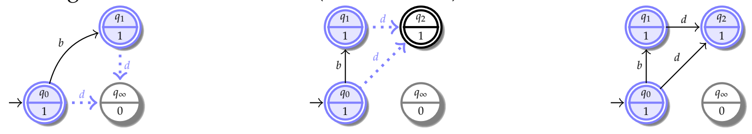
To illustrate the functioning of our algorithm, we process the example of Table 1, as an unordered stream database (*cf.* figure 2), using $\theta = 0.4$ as the support threshold. Thus, we work in the general case of data streams, *i.e.* without assuming any ordering of transactions by customer id.
Figures 3, 4, 6, 7, 8 and 10 illustrate the module INSERT, *i.e.* the reading and the insertion of an item (*cf.* Section 4.2.2 for further explanation).
Figures 5 and 9 illustrate the module NEXT, *i.e.* the end of the call to the module INSERT, which also corresponds to the end of processing every item of a transaction (*cf.* Section 4.2.2 for further explanation).

Fig. 3 Reading and insertion of item *b* (transaction 1)



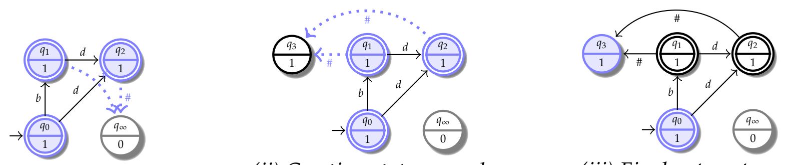
(i) Reading of item b *(ii) Creating state q_1, and moving reachable transitions.* *(iii) Final automaton*

Fig. 4 Reading and insertion of item *d* (transaction 1)



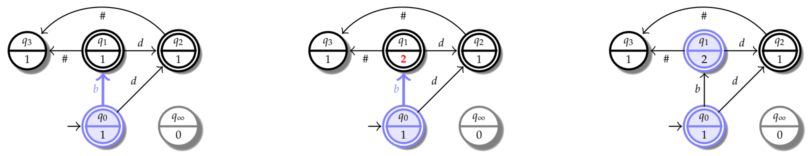
(i) Reading of item d *(ii) Creating of state q_2, and moving reachable transitions.* *(iii) Final automaton*

Fig. 5 End of processing transaction 1



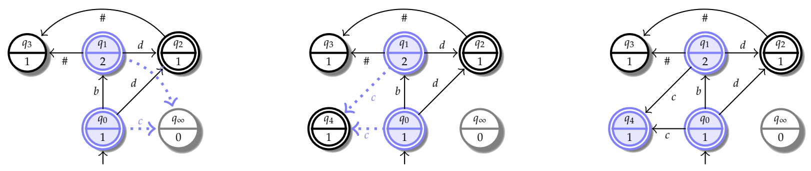
(i) Reading of item # *(ii) Creating state q_3, and moving reachable transitions.* *(iii) Final automaton.*

Fig. 6 Reading and insertion of item *b* (transaction 2)



(i) Reading of item b *(ii) Incrementing support of state q_1* *(iii) Final automaton*

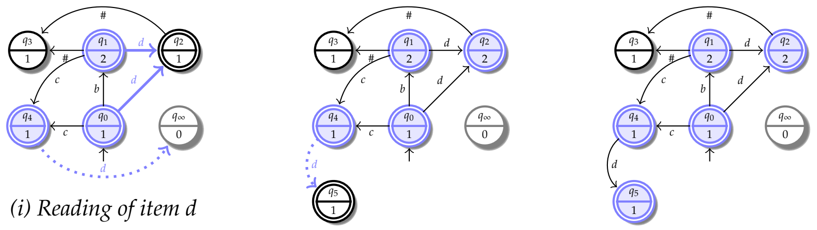
Fig. 7 Reading and insertion of item *c* (transaction 2)



(i) Reading of item c *(ii) Creating state q_4, and moving reachable transitions.* *(iii) Final automaton*

After processing the table 1 as a stream database, the resulting automaton has 38 states and 80 transitions, and contains 233 sequential patterns : this

Fig. 8 Reading and insertion of item *d* (transaction 2)

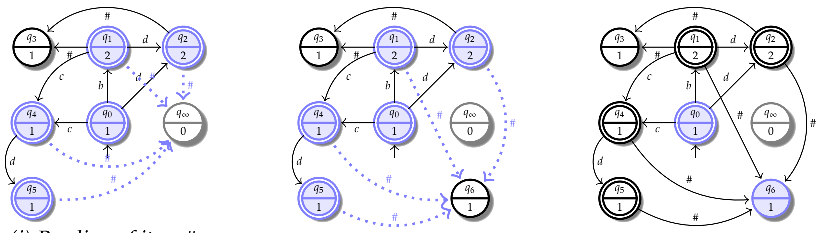


(i) Reading of item d

(ii) Creating state q_5, and moving reachable transitions

(iii) Final automaton

Fig. 9 End of processing transactions 2

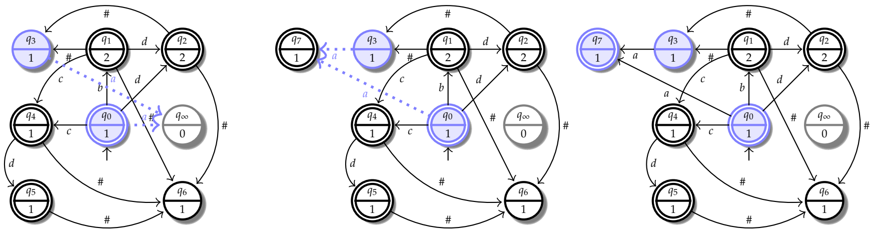


(i) Reading of item #

(ii) Creating state q_6, and moving reachable transitions.

(iii) Final automaton

Fig. 10 Reading and insertion of item *a* (transaction 3)



(i) Reading of item a

(ii) Creating state q_7, and moving reachable transitions.

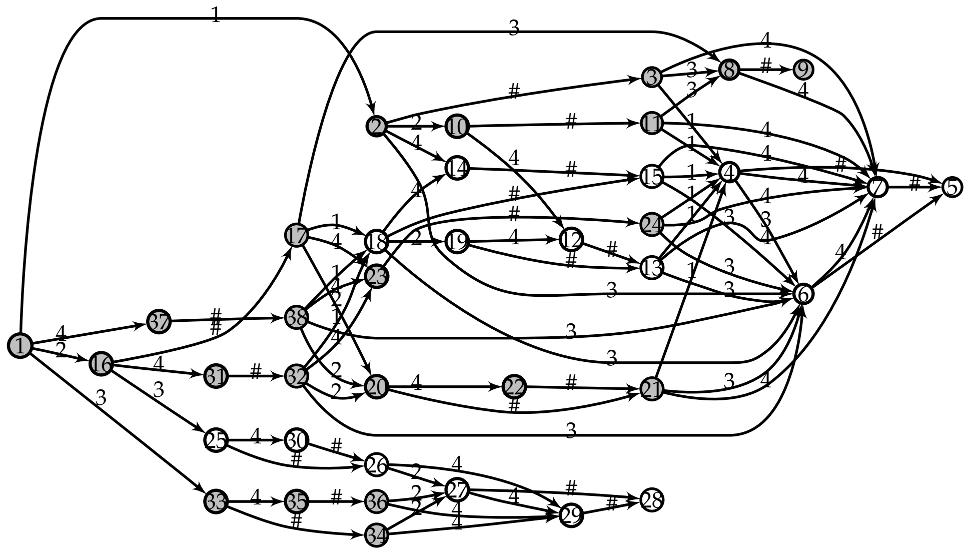
(iii) Final automaton

automaton indexes sequential patterns whose support is equal or greater than the statistical support threshold $\theta - \epsilon$ (*cf.* figure 11).
By traversing the automaton, we can extract the sequential patterns whose support is strictly greater than θ. In this case, 12 states and 19 transitions are used to index the corresponding sequential patterns, *i.e.* 19 sequential patterns (*cf.* table 2).

Table 2 This is the set of sequential patterns extracted by SPAMS using table 1 as a stream database ($\theta = 0.4$)

< (1) >:2	< (2) >:3	< (2)(4) >:2	< (2,4)(4) >:2	< (4)(2) >:2
< (1)(3) >:2	< (2)(2) >:2	< (2,4) >:2	< (3) >:3	< (4)(2,4) >:2
< (1,2) >:2	< (2)(2,4) >:2	< (2,4)(2) >:2	< (3,4) >:2	< (4)(4) >:2
< (1,2)(3) >:2	< (2)(3) >:2	< (2,4)(2,4) >:2	< (4) >:2	

Fig. 11 This is the resulting automaton generated by SPAMS, indexing all frequent sequential patterns of the statistical cover ($\theta = 0.4$) : the filled states have a support equal or greater than θ, while the white states have a support belonging to $[\theta - \epsilon; \theta[$.



4.2.4 SPAMS pseudo-code

In the following, we present the pseudo-code of our algorithm. In Section 4.2.2, the module Insert is the subject of a detailed explanation from which it is easy to deduce the pseudo-code. It's the same for the module Next. Thus, we choose to present only the pseudo-code of the main module of our algorithm as well as that of the module Prune (*cf.* algorithms 1 & 2).

5 Experimental Results

We have now designed a great number of performance tests in order to highlight our algorithm efficiency. We have used a SPAMS implementation in C++, using the Standard Template Library (STL) and the ASTL [10] li-

Algorithm 1: Main()

```
Data: $Stream, \theta$
Result: $\textsc{spa}_\theta$
begin
    Create two states $q_0$ and $q_\infty$ : $\mathcal{Q} \longleftarrow \{ q_0, q_\infty \}$
    $\mathcal{T} \longleftarrow \emptyset$
    $cid \longleftarrow NULL$
    $tid \longleftarrow NULL$
    $\mathcal{C} \longleftarrow \emptyset$
    $\mathcal{C}_{q_0} \longleftarrow \emptyset$
    $\mathcal{C}_{q_\infty} \longleftarrow \emptyset$
    $\delta \longleftarrow 0.01$
    $minSup \longleftarrow 0$
    for each $(cid', tid', \alpha) \in Stream$ do
        if $(cid \neq cid')$ or $(tid \neq tid')$ then
            Next($cid$)
            $cid \longleftarrow cid'$
            $tid \longleftarrow tid'$
        Insert($\alpha$, $cid$)
end
```

Algorithm 2: Prune()

```
Data: $s', \alpha, \mathcal{T}^r, cid$
begin
    for each $s \overset{\alpha}{\longmapsto} s' \in \mathcal{T}$ do
        Delete the transition $s \overset{\alpha}{\longmapsto} s'$ : $\mathcal{T} \longleftarrow \mathcal{T} \setminus \left\{ s \overset{\alpha}{\longmapsto} s' \right\}$
        if $s \overset{\alpha}{\longmapsto} s' \in \mathcal{T}^r$ then
            $\mathcal{T}^r \longleftarrow \mathcal{T}^r \setminus \left\{ s \overset{\alpha}{\longmapsto} s' \right\}$

    for each $s' \overset{\beta}{\longmapsto} s'' \in \mathcal{T}$ do
        Prune($s''$, $\beta$, $\mathcal{T}^r$, $cid$)
    $\mathcal{Q}_{cid} \longleftarrow \mathcal{Q}_{cid} \setminus \{ s' \}$
    for each $cid' \in \mathcal{C}_{s'}$ do
        $\mathcal{Q}_{cid'} \longleftarrow \mathcal{Q}_{cid'} \setminus \{ s' \}$
    Delete the set $\mathcal{W}_{s'}$
    Delete the state $s'$ : $\mathcal{Q} \longleftarrow \mathcal{Q} \setminus \{ s' \}$
end
```

brary, compiled with the option -03 of the g++ compiler on a 700MHz Intel Pentium(R) Core2 Duo PC machine with 4G memory, running Linux Debian Lenny.

Several experiments have been carried out in order to test the efficiency of our approach. Empirical experiments were done on synthetic datasets (*cf.* table 3) generated by the IBM data generator in [14].

Table 3 Parameters used in datasets generation

Symbols	**Meaning**
D	Number of customers in 000s
C	Average number of transactions per customer
T	Average number of items per transaction
N	Number of different items in 000s
S	Average length of maximal sequences

We illustrate on figures 12-(i),12-(ii) the time and the memory consumption performances of SPAMS, for different support values, on small medium and large datasets, respectively *D7C7T7S7N1*, *D10C10T10S10N1* and *D15C15T15S15N1*. Figures 12-(iii), 12-(iv), 12-(v), 12-(vi) represent the evolution of the running time, the memory and the number of customers in relation to the number of transactions on the dataset *D15C15T15S15N1*, with a fixed support value of *40%*. Figure 12-(viii) illustrates that the statistical support used tends to the support threshold θ during the insertion of new transactions, which reduce the $(\theta-\epsilon)$-frequent patterns of the statistical cover. To calculate the ϵ parameter (see section 3), we have chosen the value of *0.01* for the statistical risk parameter δ. These experiments show that we have found a satisfactory compromise between time performances, memory consumption and the quality of the mining results in recall as well as in precision (*cf.* figure 12-(vii)). They also show the applicability and the scalability of the SPAMS algorithm for mining data streams.

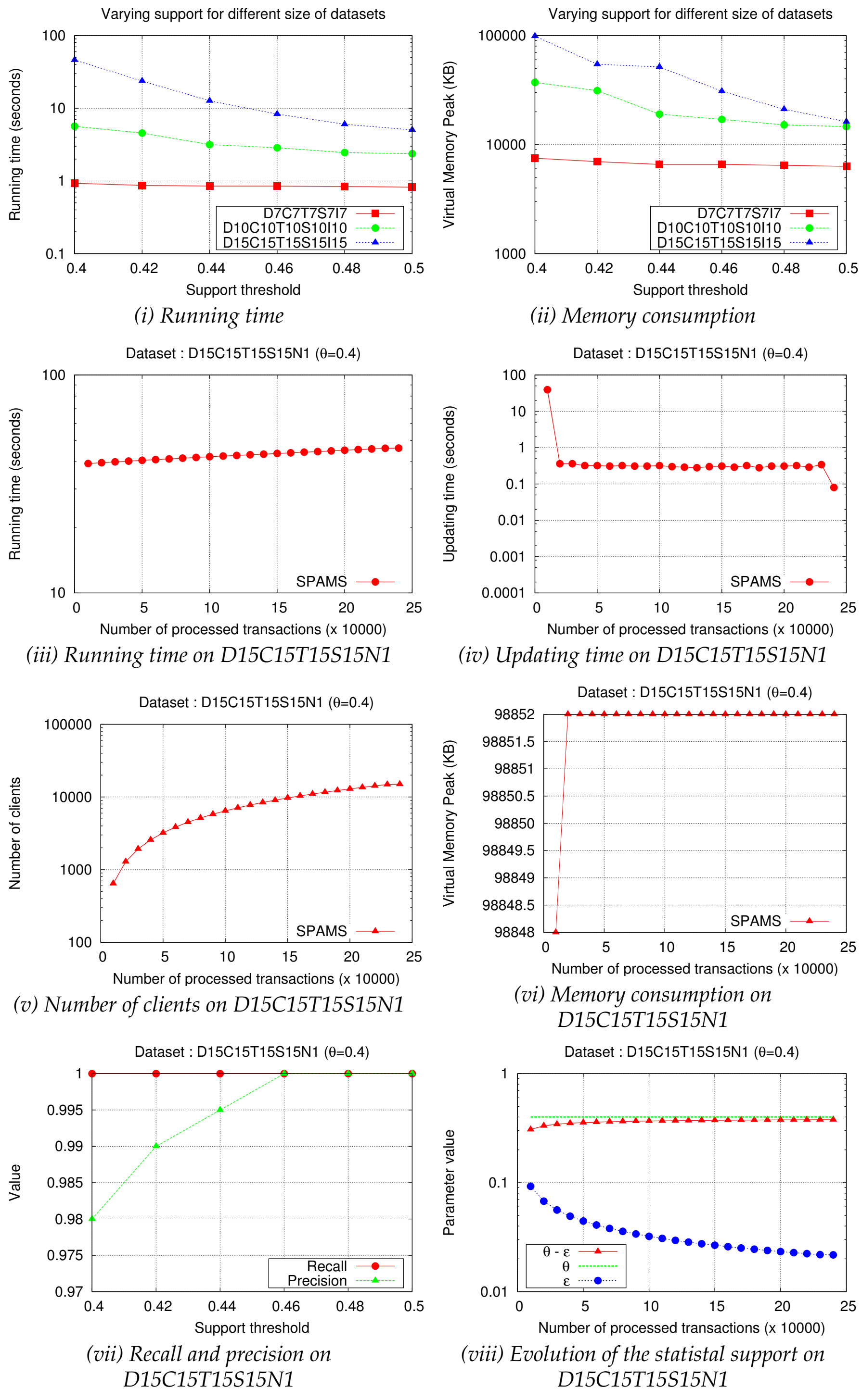
6 Conclusion

In this paper, we bring an original contribution by proposing a new one-pass algorithm, named SPAMS, enabling the building and the maintaining of an automaton data structure: the SPA, which indexes the frequent sequential patterns in a data stream. The SPA is built from scratch and is updated on the volley, as a new transaction is inserted. The current frequent patterns can be output in real time based on any user's specified thresholds. Thus, the SPA is a very informative and flexible data structure, well-suited for mining frequent sequential patterns in data streams. With the SPA, our contribution opens a promising gateway, by using an automaton as a data structure for mining frequent sequential patterns in data streams. Furthermore, taking into account the characteristics of data streams, we propose a well-suited method, said to be approximate, since we can provide near optimal results with a high probability, while maintaining satisfactory performances of the SPAMS algorithm. Experimental studies show the scalability and the applicability of the SPAMS algorithm. In the future, we will examine how to extend this work to mine closed sequential patterns on sliding windows.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: 20th VLDB Conf. (1994)
2. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation (2002)
3. Chang, J.H., Lee, W.S.: Efficient mining method for retrieving sequential patterns over online data streams. J. Inf. Sci. **31**(5), 420–432 (2005). DOI http://dx.doi.org/10.1177/0165551505055405
4. Garofalakis, M.N., Gehrke, J., Rastogi, R.: Querying and mining data streams: you only get one look a tutorial. In: M.J. Franklin, B. Moon, A. Ailamaki (eds.) SIGMOD Conference, p. 635. ACM (2002). URL `http://dblp.uni-trier.de/db/conf/sigmod/sigmod2002.html#GarofalakisGR02`
5. Gouda, K., Hassaan, M., Zaki, M.J.: Prism: A primal-encoding approach for frequent sequence mining. In: ICDM, pp. 487–492. IEEE Computer Society (2007). URL `http://dblp.uni-trier.de/db/conf/icdm/icdm2007.html#GoudaHZ07`
6. Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., Hsu, M.: Freespan: frequent pattern-projected sequential pattern mining. In: KDD, pp. 355–359 (2000)
7. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, Reading, Massachusetts (1979)
8. Laur, P.A., Symphor, J.E., Nock, R., Poncelet, P.: Statistical supports for mining sequential patterns and improving the incremental update process on data streams. Intell. Data Anal. **11**(1), 29–47 (2007)
9. Li, H., Chen, H.: Graseq : A novel approximate mining approach of sequential patterns over data stream. In: R. Alhajj, H. Gao, X. Li, J. Li, O.R. Zaane (eds.) ADMA, *Lecture Notes in Computer Science*, vol. 4632, pp. 401–411. Springer (2007). URL `http://dblp.uni-trier.de/db/conf/adma/adma2007.html\#LiC07`
10. Maout, V.L.: Tools to implement automata, a first step: Astl. In: D. Wood, S. Yu (eds.) Workshop on Implementing Automata, *Lecture Notes in Computer Science*, vol. 1436, pp. 104–108. Springer (1997)
11. Masseglia, F., Cathala, F., Poncelet, P.: The psp approach for mining sequential patterns. In: J.M. Zytkow, M. Quafafou (eds.) PKDD, *Lecture Notes in Computer Science*, vol. 1510, pp. 176–184. Springer (1998)
12. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Prefixspan: Mining sequential patterns by prefix-projected growth. In: ICDE, pp. 215–224. IEEE Computer Society (2001)
13. Raïssi, C., Poncelet, P.: Sampling for sequential pattern mining: From static databases to data streams. In: ICDM, pp. 631–636 (2007)
14. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: P.M.G. Apers, M. Bouzeghoub, G. Gardarin (eds.) EDBT, *Lecture Notes in Computer Science*, vol. 1057, pp. 3–17. Springer (1996)
15. Zaki, M.J.: Spade: An efficient algorithm for mining frequent sequences. Machine Learning **42**(1/2), 31–60 (2001)

Fig. 12 Self performance evaluation of SPAMS over small, medium and large datasets.



(i) Running time

(ii) Memory consumption

(iii) Running time on D15C15T15S15N1

(iv) Updating time on D15C15T15S15N1

(v) Number of clients on D15C15T15S15N1

(vi) Memory consumption on D15C15T15S15N1

(vii) Recall and precision on D15C15T15S15N1

(viii) Evolution of the statistal support on D15C15T15S15N1