Mach Learn (2017) 106:799–836
DOI 10.1007/s10994-016-5617-1

Memory-adaptive high utility sequential pattern mining over data streams

Morteza Zihayat[1] · **Yan Chen**[1] · **Aijun An**[1]

Received: 29 April 2016 / Accepted: 18 November 2016 / Published online: 2 February 2017

Abstract High utility sequential pattern (HUSP) mining has emerged as an important topic in data mining. A number of studies have been conducted on mining HUSPs, but they are mainly intended for non-streaming data and thus do not take data stream characteristics into consideration. Streaming data are fast changing, continuously generated unbounded in quantity. Such data can easily exhaust computer resources (e.g., memory) unless a proper resource-aware mining is performed. In this study, we explore the fundamental problem of how limited memory can be best utilized to produce high quality HUSPs over a data stream. We design an approximation algorithm, called *MAHUSP*, that employs memory adaptive mechanisms to use a bounded portion of memory, in order to efficiently discover HUSPs over data streams. An efficient tree structure, called *MAS-Tree*, is proposed to store potential HUSPs over a data stream. *MAHUSP* guarantees that all HUSPs are discovered in certain circumstances. Our experimental study shows that our algorithm can not only discover HUSPs over data streams efficiently, but also adapt to memory allocation with limited sacrifices in the quality of discovered HUSPs. Furthermore, in order to show the effectiveness and efficiency of *MAHUSP* in real-life applications, we apply our proposed algorithm to a web clickstream dataset obtained from a Canadian news portal to showcase users' reading behavior, and to a real biosequence database to identify disease-related gene regulation sequential patterns. The results show that *MAHUSP* effectively discovers useful and meaningful patterns in both cases.

Editors: Nathalie Japkowicz and Stan Matwin.

✉ Aijun An
aan@cse.yorku.ca

Morteza Zihayat
zihayatm@cse.yorku.ca

Yan Chen
ychen@cse.yorku.ca

1 Department of Computer Science and Engineering, York University, 4700 Keele Street, Toronto, ON M3J 1P3, Canada

Keywords High utility sequential pattern mining · Data streams · Approximation algorithms

1 Introduction

Sequential pattern mining is an important task in data mining and has been extensively studied by many researchers (Mooney and Roddick 2013). Given a dataset of sequences, each containing a list of itemsets, sequential pattern mining is to discover sequences of itemsets that frequently appear in the dataset. For example, in market basket analysis, each sequence in the dataset represents a list of transactions made by a customer over time and each transaction contains a set of items purchased by the customer. Mining sequential patterns from this dataset finds the sequences of itemsets that are frequently purchased by customers in the time order. Despite its usefulness, sequential pattern mining has the limitation that it neither considers the frequency of an item within an itemset nor the importance of an item (e.g., the profit of an item). Thus, some infrequent sequences with high profits may be missed. For example, selling a TV is much more profitable than selling a bottle of milk, but a sequence containing a TV is much more infrequent than the one with a bottle of milk. These profitable patterns address several important questions in business decisions such as how to maximize revenue or minimize marketing or inventory costs. Obviously, identifying sequences with high profits is important for businesses. Recently, high utility sequential pattern mining has been studied to address this limitation (Ahmed et al. 2011; Yin et al. 2012; Zihayat et al. 2015). In high utility sequential pattern mining, each item has a *global weight* (representing e.g., unit price/profit) and a *local weight* in a transaction (representing e.g., purchase quantity). A sequence is a *high utility sequential pattern* (*HUSP*) if and only if its utility (calculated based on local and global weights of the items it contains) in a dataset is no less than a minimum utility threshold. HUSP mining is very desirable to many applications such as *market analysis*, *web mining*, *bioinformatics* and *mobile computing*.

Although some preliminary works have been conducted on this topic, existing studies (Ahmed et al. 2011; Wang et al. 2014; Yin et al. 2013; Shie et al. 2013) mainly focus on mining high utility sequential patterns in static databases and do not consider the real-world applications that involve *data streams*. In many applications, such as *web clickstream mining*, *network traffic analysis*, *intrusion detection* and *online analysis of user behavior*, streaming data are continually generated in a high speed. Mining algorithms need to process the data in one scan of data. The data in the data stream may be analyzed when a new instance arrives or when a new *batch* forms. A batch is a set of M instances that occur during a period of time. In the instance-based approaches, when a new instance is generated in the stream, it will be analyzed immediately. In contrast, batch-based approaches process the stream when a new batch forms. Batch processing is more efficient when the system can wait to form a batch of instances. Generally, there are three main types of stream-processing models: *damped window based*, *sliding window based* and *landmark window based*.

In the *damped* model (also called *time-fading* model), each record (e.g., an instance/a batch of instances) is assigned with a weight that decreases over time. Therefore, in this model, recent data are more important than old ones. However, it is difficult for the users who are not domain experts to choose an appropriate *decay function* or *decay rate* for this model. The *sliding window* model captures a fixed number of most recent records (e.g., instances/batches of instances) in a window, and it focuses on discovering the patterns within the window. When a new record flows into the window, the oldest one is expelled from the window. In this model, the effect of the expired old data is eliminated, and the patterns are mined from the recent data in the window. The first two models place more importance on recent data than old ones (e.g., by assigning a weight to each data record which decreases over time in the *damped window* or using a fixed amount of recent data in the *sliding window*). This characteristic is very useful for the applications where users are more interested in finding patterns or interesting phenomenon hidden in the most recent data. Moreover, focusing on recent data can detect new characteristics of the data or changes in data distributions quickly. However, in some applications long-term monitoring is necessary and users may want to treat all data elements starting from a past time point equally and discover patterns over a long period of time in the data stream (Manku and Motwani 2002). The *landmark window* is used for such a purpose, which treats all data records starting from a past time point (called *Landmark*) until the current time equally and discovers patterns over a long period of time in the data stream. There are many applications where finding patterns over a long period of time in the data stream is useful. For example, in *web mining*, we may want to discover user's browsing behavior since the website changed its layout; in *healthcare*, in a medical dataset which contains medical treatments for a certain disease for different patients, doctors may want to monitor the sequence of side-effects of a treatment since it started to be used; in *DNA data analysis*, a domain expert may want to find disease-related gene regulation sequential patterns after a treatment is started; in *retail businesses*, we may want to detect important buying sequences of customers since the beginning of a year; and in an *energy network*, we may want to find important event sequences since a new set of equipments was installed to monitor the quality of equipments over its life-time. Monitoring only recent data (e.g, in sliding windows) may miss such sequences that are important for decision making over a long term.[1]

Considering all these applications, in this paper we aim at finding high utility sequential patterns (HUSPs) over a landmark window. In particular, the landmark in our method is the beginning of the data stream. Thus, we are dealing with the problem of incrementally learning HUSPs over the entire data stream, which is more difficult than the sliding window-based mining. In this paper, we assume that the data in the data stream arrive very fast and in a large volume, hence mining algorithms need to process the arriving data in real time with one scan of data. In this case, a complete re-scan of a long portion of the data stream is usually impossible or prohibitively costly. Moreover, data can be huge and the available memory to keep the information we need to find patterns, is limited.

Compared with other data stream mining tasks, there are unique challenges in discovering HUSPs using a landmark window. First, HUSP mining needs to search a large search space due to a combinatorial number of possible sequences. Second, the utility of a sequence does not have the *downward closure property* (Shie et al. 2013; Yin et al. 2012), which would allow efficient pruning of search space. That is, the utility of a sequence may be higher than, equal to or lower than those of its super-sequences and sub-sequences (Shie et al. 2013; Yin et al. 2012). This prevents the direct application of frequent sequential pattern mining algorithms to HUSP mining. Consequently, keeping up the pace with high speed data streams can be very hard for a HUSP mining task. A more important issue is the need of capturing the information of data over a potentially long period of time. Data can be huge such that the amount of information we need to keep may exceed the size of available memory. Thus, to avoid memory thrashing or crashing, memory-aware data processing is needed to ensure that the size of the data structure does not exceed the available memory, and at the same time accurate approximation of the information needed for the mining process is necessary.

[1] Note that landmark models may not be suitable for some applications such as *network traffic monitoring* and *intrusion detection*, in which users are more interested in the information that reflect recent data rather than old ones.

In this paper, we tackle these challenges and propose a memory-adaptive approach to discovering HUSPs from a dynamically-increasing data stream. To the best of our knowledge, this is the first piece of work to mine high utility sequential patterns over data streams in a memory adaptive manner. Our contributions are summarized as follows. First, we propose a novel method for incrementally mining HUSPs over a data stream. Our method can not only identify recent HUSPs but also HUSPs over a long period of time (i.e., since the start of the data stream). Second, we propose a novel and compact data structure, called *MAS-Tree*, to store potential HUSPs over a data stream. The tree is updated efficiently once a new potential HUSP is discovered. Third, two efficient memory adaptive mechanisms are proposed to deal with the situation when the available memory is not enough to add a new potential HUSPs to *MAS-Tree*. The proposed mechanisms choose the least promising patterns to remove from the tree to guarantee that the memory constraint is satisfied and also that all true HUSPs are maintained in the tree under certain circumstances. Fourth, using *MAS-Tree* and the memory adaptive mechanisms, our algorithm, called *MAHUSP*, efficiently discovers HUSPs over a data stream with a high recall and precision. The proposed method guarantees that the memory constraint is satisfied and also all true HUSPs are maintained in the tree under certain circumstances. Fifth, we conduct extensive experiments and show that *MAHUSP* finds an approximate set of HUSPs over a data stream efficiently and adapts to memory allocation without sacrificing much the quality of discovered HUSPs.

The paper is organized as follows. Section 2 provides relevant definitions and a problem statement. Section 3 proposes the data structures and algorithms. Experimental results and two applications of our proposed algorithms are shown in Sect. 4.1. In Sect. 5, we present the related work. We conclude the paper in Sect. 6.

2 Definitions and problem statement

Let $I^* = \{I_1, I_2, \ldots, I_N\}$ be a set of items. An itemset X is a set of items. An itemset-sequence S (or sequence in short) is an ordered list of itemsets $\langle X_1, X_2, \ldots, X_Z\rangle$, where $X_i \subseteq I^*$ and Z is the *size* of S. The *length* of S is defined as $\sum_{i=1}^{Z} |X_i|$, where $|X_i|$ is the number of items in itemset X_i. A sequence of length L is called *L-sequence*. In this paper, each itemset X_d in sequence S_r is denoted as S_r^d.

In a data stream environment, sequences come continuously over time and they are usually processed in batches (Manku and Motwani 2002; Mendes et al. 2008). A *batch* $B_k = \{S_i, S_{i+1}, \ldots, S_{i+M-1}\}$ is a set of M sequences that occur during a period of time t_k. The number of sequences can differ among batches. A *sequence data stream* $DS = \langle B_1, B_2, \ldots, B_k, \ldots\rangle$ is an ordered and unbounded list of batches where $B_i \bigcap B_j = \emptyset$ and $i \neq j$. Figure 1 shows a sequence data stream with 2 batches $B_1 = \{S_1, S_2\}$ and $B_2 = \{S_3, S_4, S_5\}$.

	SID	Sequence Data
B_1	S_1	S_1^1:{(a,2)(b,3)(c,2)}; S_1^2:{(b,1)(c,1)(d,1)}; S_1^3:{(c,3)(d,1)}
	S_2	S_2^1:{(b,4)}; S_2^2:{(a,4)(b,5)(c,1)}
B_2	S_3	S_3^1:{(b,3)(d,1)}; S_3^2:{(a,4)(b,5)(c,1)}; S_3^3:{(a,2)(c,3)}
	S_4	S_4^1:{(a,2)(b,5)(e,2)}
	S_5	S_5^1:{(c,4)}

Item	Profit
a	2
b	3
c	1
d	4
e	3

Fig. 1 An example of a data stream of itemset-squences

Definition 1 (*External utility and internal utility*) Each item $I \in I^*$ is associated with a positive number $p(I)$, called its *external utility* (e.g., price/unit profit). In addition, each item I in itemset X_d of sequence S_r (i.e., S_r^d) has a positive number $q(I, S_r^d)$, called its internal utility (e.g., quantity) of I in X_d or S_r^d.

Definition 2 (*Super-sequence and Sub-Sequence*) Sequence $\alpha = \langle X_1, X_2, \ldots, X_i\rangle$ is a *sub-sequence* of $\beta = \langle X_1^{'}, X_2^{'}, \ldots, X_j^{'}\rangle$ $(i \leq j)$ or equivalently β is a *super-sequence* of α if there exist integers $1 \leq e_1 < e_2 < \cdots e_i \leq j$ such that $X_1 \subseteq X_{e_1}^{'}, X_2 \subseteq X_{e_2}^{'}, \ldots, X_i \subseteq X_{e_i}^{'}$ (denoted as $\alpha \preceq \beta$).

For example, if $\alpha = \langle\{ac\}\{d\}\rangle$ and $\beta = \langle\{abc\}\{bce\}\{cd\}\rangle$, α is a sub-sequence of β and β is the super-sequence of α.

Definition 3 (*Utility of an item in an itemset of a sequence S_r*) The utility of an item I in an itemset X_d of a sequence S_r is defined as $u(I, S_r^d) = f_u(p(I), q(I, S_r^d))$, where f_u is the function for calculating utility of item I based on internal and external utility. For simplicity, without loss of generality, we define the utility function as $f_u(p(I), q(I, S_r^d)) = p(I) \cdot q(I, S_r^d)$.

Definition 4 (*Utility of an itemset in an itemset of a sequence S_r*) Given itemset X, the utility of X in the itemset X_d of the sequence S_r where $X \subseteq X_d$, is defined as $u(X, S_r^d) = \sum_{I\in X} u(I, S_r^d)$.

For example, in Fig. 1, the utility of item b in the first itemset of S_1 (i.e., S_1^1) is $u(b, S_1^1) = p(b)\cdot q(b, S_1^1) = 3 \times 3 = 9$. The utility of the itemset $\{bc\}$ in S_1^1 is $u(\{bc\}, S_1^1) = u(b, S_1^1) + u(c, S_1^1) = 9 + 2 = 11$.

Definition 5 (*Occurrence of a sequence α in a sequence S_r*) Given a sequence $S_r = \langle S_r^1, S_r^2, \ldots, S_r^n\rangle$ and a sequence $\alpha = \langle X_1, X_2, \ldots, X_Z\rangle$ where S_r^i and X_i are itemsets, α occurs in S_r iff there exist integers $1 \leq e_1 < e_2 < \cdots < e_Z \leq n$ such that $X_1 \subseteq S_r^{e_1}, X_2 \subseteq S_r^{e_2}, \ldots, X_Z \subseteq S_r^{e_Z}$. The ordered list of itemsets $\langle S_r^{e_1}, S_r^{e_2}, \ldots, S_r^{e_Z}\rangle$ is called an *occurrence of α in S_r*. The set of all occurrences of α in S_r is denoted as $OccSet(\alpha, S_r)$.

Definition 6 (*Utility of a sequence α in a sequence S_r*) Let $\tilde{o} = \langle S_r^{e_1}, S_r^{e_2}, \ldots, S_r^{e_Z}\rangle$ be an occurrence of $\alpha = \langle X_1, X_2, \ldots, X_Z\rangle$ in the sequence S_r. The utility of α w.r.t. $\tilde{o}$ is defined as $su(\alpha, \tilde{o}) = \sum_{i=1}^{Z} u(X_i, S_r^{e_i})$. The utility of α in S_r is defined as $su(\alpha, S_r) = \max\{su(\alpha, \tilde{o}) \mid \tilde{o} \in OccSet(\alpha, S_r)\}$.

Consequently, the *utility of a sequence S_r* is defined and denoted as $su(S_r) = su(S_r, S_r)$.

For example, in Fig. 1, the set of all occurrences of the sequence $\alpha = \langle\{bd\}\{c\}\rangle$ in S_3 is $OccSet(\langle\{bd\}\{c\}\rangle, S_3) = \{\tilde{o}_1{:}\, \langle S_3^1, S_3^2\rangle, \tilde{o}_2{:}\, \langle S_3^1, S_3^3\rangle\}$. Hence $su(\alpha, S_3) = \max\{su(\alpha, \tilde{o}_1), su(\alpha, \tilde{o}_2)\} = \{14, 16\} = 16$.

Definition 7 (*Utility of a sequence α in a dataset D*) The utility of a sequence α in a dataset D of sequences is defined as $su(\alpha, D) = \sum_{S_r\in D} su(\alpha, S_r)$, where D can be a batch or a data stream processed so far.

The *total utility of a batch B_k* is defined as $U_{B_k} = \sum_{S_r\in B_k} su(S_r, S_r)$. The *total utility of a data stream $DS_i = \langle B_1, B_2, \ldots, B_i\rangle$* is defined as $U_{DS_i} = \sum_{B_k\in DS_i} U_{B_k}$.

Table 1 Summary of notations

Notation	Description
$u(X, S_r^d)$	Utility of item/itemset X in the itemset X_d of S_r
$\alpha \preceq \beta$	α is a subsequence of β, or α occurs in β
$OccSet(\alpha, S_r)$	Set of all the occurrences of α in sequence S_r
$su(\alpha, S_r)$	Utility of a sequence α in sequence S_r
U_{B_k}	Total utility of a batch B_k
U_{DS_i}	Total utility of a data stream DS_i
$\alpha \lesssim \beta$	α is a prefix sub-sequence of β
$rsu(\alpha, D)$	Rest utility of sequence α in dataset D
LBMA	Leaf Based Memory Adaptation
SBMA	Sub-Tree Based Memory Adaptation

Definition 8 (*High utility sequential pattern*) Given a utility threshold δ in percentage, a sequence α is a *high utility sequential pattern* (HUSP) in data stream *DS*, iff $su(\alpha, DS)$ is not less than $\delta \cdot U_{DS}$.

Problem statement Given a utility threshold δ (in percentage), the maximum available memory *availMem*, and a dynamically-changing data stream $DS = \langle B_1, B_2, \ldots, B_i, \ldots\rangle$ (where batch B_i contains a set of sequences of itemsets at time period t_i), our problem of online memory-adaptive mining of high utility sequential patterns over data stream *DS* is to discover, at any time t_i ($i \geq 1$), all sub-sequences of itemsets whose utility in DS_i is no less than $\delta \cdot U_{DS_i}$ where $DS_i = \langle B_1, B_2, \ldots, B_i\rangle$ under the following constraints: (1) the memory usage does not exceed *availMem*, and (2) only one pass of data is allowed in total.

For convenience, Table 1 summarizes the concepts and notations we define in this paper.

3 Memory adaptive high utility sequential pattern mining

In this section, we propose a single-pass algorithm named *memory adaptive high utility sequential pattern mining over data streams* (*MAHUSP*) for incrementally mining an approximate set of HUSPs over a data stream. Below we first present an overview of *MAHUSP* and then propose a novel tree-based data structure, called *memory adaptive high utility sequential pattern tree* (*MAS-Tree*), to store the essential information of HUSPs over the data stream. Finally, we propose two memory adaptive mechanisms and use them in the tree construction and updating process.

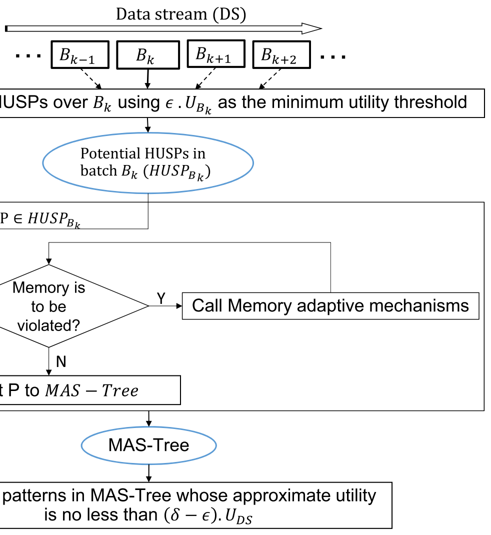
3.1 Overview of MAHUSP

Algorithm 1 represents an overview of *MAHUSP*. Given a utility threshold δ ($0 \leq \delta \leq 1$) and a significance threshold ϵ ($0 \leq \epsilon \leq \delta$), as a new batch B_k forms, *MAHUSP* first applies an existing HUSP mining algorithm on static data [e.g., USpan (Yin et al. 2012)[2]] to find a set of HUSPs over B_k using ϵ as the utility threshold. We consider this set of HUSPs as *potential* HUSPs since they have the potential to become HUSPs later. What *MAHUSP* exports via ϵ to the user is a tradeoff between accuracy and run time. That is, if the user does not want to miss any *HUSPs* over the data stream, *MAHUSP* should consider all patterns in the batch as

[2] Note that USpan finds HUSPs with one-pass over data but is not an incremental learning algorithm.

Algorithm 1 *MAHUSP*

```
Input: $B_k$, $\delta$, $\epsilon$, availMem, mechanismType
Output: MAS-Tree, appHUSPs
1: $HUSP_{B_k} \leftarrow$ HUSPs returned by USpan on $B_k$ using $\epsilon \cdot U_{B_k}$ as minimum utility threshold
2: if MAS-Tree is empty (i.e. $B_k$ is the first batch) then
3:     Initialize MAS-Tree by creating root node
4: end if
5: Call Algorithm 2 to insert the patterns in $HUSP_{B_k}$ into MAS-Tree using availMem and mechanismType
6: if user requests for HUSPs over current data stream then
7:     appHUSPs $\leftarrow$ potential HUSPs in MAS-Tree whose approximate utility is no less than $(\delta - \epsilon) \cdot U_{DS}$
8: end if
9: return MAS-Tree and appHUSPs if requested
```



Data stream (DS)
B_{k-1}
B_k
B_{k+1}
B_{k+2}
Apply USpan to find HUSPs over B_k using $\epsilon . U_{B_k}$ as the minimum utility threshold
Potential HUSPs in batch B_k ($HUSP_{B_k}$)
For each P ∈ $HUSP_{B_k}$
Memory is to be violated?
Y
Call Memory adaptive mechanisms
N
Insert P to $MAS-Tree$
MAS-Tree
Return all patterns in MAS-Tree whose approximate utility is no less than $(\delta - \epsilon). U_{DS}$

Fig. 2 MAHUSP workflow

potential *HUSPs* over the data stream. Due to the combinatorial explosion among items, the task of finding all the patterns may not even finish in a promising time. Using ϵ, *MAHUSP* allows users to consider the patterns with utility no less than a reasonable threshold, set by the user, in the batch as potential HUSPs over the data stream. Therefore, a smaller ϵ value leads to more accurate results but longer run time.

Given set of potential HUSPs in B_k, *MAHUSP* then calls Algorithm 2 to insert these potential HUSPs into the *MAS-Tree* structure. Algorithm 2 calls a memory-adaptive mechanism (to be described) to ensure that the memory constraint is satisfied and the most potential HUSPs are kept in the tree. The *MAS-Tree* contains the potential HUSPs and their approximated utilities over the data stream. Finally, if users request to find HUSPs from the stream so far, *MAHUSP* returns the set of all the patterns (i.e., *appHUSPs*) in *MAS-Tree* with approximate utility more than $(\delta - \epsilon) \cdot U_{DS_k}$, where $DS_k = \langle B_1, B_2, \ldots, B_k \rangle$. In Sect. 3.6, we will explain why we use $(\delta - \epsilon) \cdot U_{DS}$ as the utility threshold.

The aforementioned workflow is presented in Fig. 2. Below we first describe how *MAS-Tree* is structured. Then the proposed memory adaptive mechanisms and tree construction will be presented.

3.2 MAS-Tree structure

We propose a novel data structure memory adaptive high utility sequential tree (MAS-Tree) to store potential HUSPs in a data stream. This tree allows compact representation and fast update of potential HUSPs generated in the batches, and also facilitates the pruning of unpromising patterns to satisfy the memory constraint. In order to present *MAS-Tree*, the following definitions are provided (Pei et al. 2004).

Definition 9 (*Prefix itemset of an itemset*) Given itemsets $X_1 = \{I_1, I_2, \ldots, I_i\}$ and $X_2 = \{I_1^{'}, I_2^{'}, \ldots, I_j^{'}\}$ $(i < j)$, where items in each itemset are listed in a lexicographic order, X_1 is a prefix itemset of X_2 iff $I_1 = I_1^{'}, I_2 = I_2^{'}, \ldots,$ and $I_i = I_i^{'}$ (denoted as $X_1 \lesssim X_2$). Note that items in an itemset are arranged in the lexicographic order.

Definition 10 (*Suffix itemset of an itemset*) Given itemsets $X_1 = \{I_1, I_2, \ldots, I_i\}$ and $X_2 = \{I_1^{'}, I_2^{'}, \ldots, I_j^{'}\}$ $(i \leq j)$, such that $X_1 \lesssim X_2$. The suffix itemset of X_2 w.r.t. X_1 is defined as: $X_2 - X_1 = \{I_{i+1}^{'}, I_{i+2}^{'}, \ldots, I_j^{'}\}$.

For example, itemset $X_1 = \{ab\}$ is a prefix itemset $X_2 = \{abce\}$ and $X_2 - X_1 = \{ce\}$.

Definition 11 (*Prefix sub-sequence and Prefix super-sequence*) Given sequences $\alpha = \langle X_1, X_2, \ldots, X_i\rangle$ and $\beta = \langle X_1^{'}, X_2^{'}, \ldots, X_j^{'}\rangle$ $(i \leq j)$, α is a prefix sub-sequence (or *prefix-SUB* in short) of β or equivalently β is a prefix super-sequence (or *prefixSUP* in short) of α iff $X_1 = X_1^{'}, X_2 = X_2^{'}, \ldots, X_{i-1} = X_{i-1}^{'}, X_i \lesssim X_i^{'}$ (denoted as $\alpha \lesssim \beta$).

Definition 12 (*Suffix of a sequence*) Given a sequence $\alpha = \langle X_1, X_2, \ldots, X_i\rangle$ as a *prefixSUB* of $\beta = \langle X_1^{'}, X_2^{'}, \ldots, X_j^{'}\rangle$ $(i \leq j)$, sequence $\gamma = \langle X'_i - X_i, X'_{i+1}, \ldots, X'_j\rangle$ is called the suffix of β w.r.t. α.

For example, $\alpha = \langle\{abc\}\{b\}\rangle$ is a prefixSUB of $\beta = \langle\{abc\}\{bce\}\{cd\}\rangle$ and β is the *prefixSUP* of α. Hence, suffix of β w.r.t. α is $\langle\{ce\}\{cd\}\rangle$.

In an MAS-Tree, each node represents a sequence, and a sequence S_P represented by a parent node P is a prefixSUB of the sequence S_C represented by P's child node C. The child node C stores the suffix of S_C with respect to its parent sequence S_P. Thus, the sequence represented by a node N is the "concatenation" of the subsequences stored in the nodes along the path from the root (which represents the empty sequence) to N. There are two types of nodes in an MAS-Tree: C-nodes and D-nodes.

A *C-node* or *Candidate node* uniquely represents a potential HUSP found in one of the batches processed so far. For example, there are 6 C-nodes in Fig. 3a representing 6 potential HUSPs (i.e., $\langle\{ab\}\rangle$, $\langle\{abc\}\rangle$, $\langle\{b\}\{ab\}\rangle$, $\langle\{b\}\{abc\}\rangle$, $\langle\{b\}\{b\}\rangle$, and $\langle\{b\}\{bc\}\{b\}\rangle$).

A *D-node* or *Dummy* node is a non-leaf node with at least two child nodes, representing a sequence that is not a potential HUSP but is the longest common prefixSUB of all the potential HUSPs represented by its descendent nodes. In Fig. 3a, there is one D-node representing $\langle\{b\}\rangle$, which is the longest common prefixSUB of four C-node sequences $\langle\{b\}\{ab\}\rangle$, $\langle\{b\}\{abc\}\rangle$, $\langle\{b\}\{b\}\rangle$ and $\langle\{b\}\{bc\}\{b\}\rangle$. The reason for having D-nodes in the tree is to use shared nodes to store common prefixes of HUSPs to save space. Note that D-nodes are created only for storing the longest common prefixes (not every prefix) of potential HUSPs to keep the number of nodes minimum.

The MAS-Tree is different from the prefix tree used to represent sequences for frequent sequence mining where all the sub-sequences of a frequent sequence are frequent and are represented by different tree nodes. In a MAS-Tree, we do not store all subsequences of

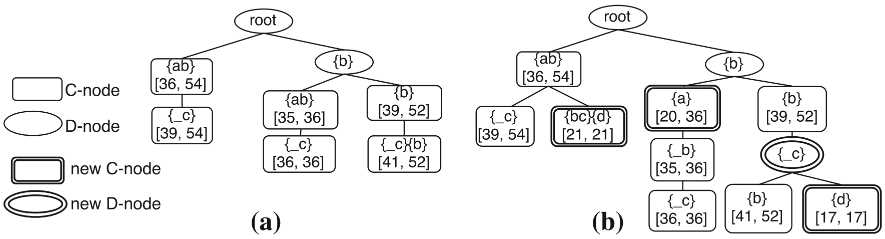


Fig. 3 **a** An example of *MAS-Tree* for B_1 in Fig. 1. Note that an underscore in a node name $\{\_c\}$ means that the last itemset in the pattern of its parent, such as $\{ab\}$, belongs to the first itemset of the pattern of this node. That is $\{ab\}$ and $\{\_c\}$ forms $\{abc\}$, **b** *MAS-Tree* after inserting three patterns: $\langle\{ab\}\{bc\}\{d\}\rangle$, $\langle\{b\}\{a\}\rangle$ and $\langle\{b\}\{bc\}\{d\}\rangle$

potential *HUSP*s since a subsequence of a *HUSP* may not be a *HUSP*. For example, in Fig. 1, given support threshold *minSup* $= 2$ and minimum utility threshold *minUtil* $= 40$, sequence $\alpha = \{b\}\{b, c\}$ is a frequent pattern and its frequency in B_1 is 2. Since frequency has the *downward closure property*, all of its prefix subsequences including $\alpha_1 = \{b\}$ and $\alpha_2 = \{b\}\{b\}$ are frequent patterns. Hence, all the subsequences are inserted as separate nodes in the tree. On the other hand, although α is a *HUSP* in B_1 ($su(\alpha, B_1) = 41$), its prefix subsequences (e.g., $su(\alpha_1, B_1) = 24$, $su(\alpha_2, B_1) = 39$) are not *HUSPs* in the batch B_1. Therefore, we do not create nodes for α_1 and α_2 in MAS-Tree to consume less memory. However, if we use the prefix trees used to represent frequent sequences, we have to create nodes for α_1 and α_2, which is not memory efficient.

Let S_N denote the sequence represented by a node N. A C-node N contains 3 fields: *nodeName*, *nodeUtil* and *nodeRsu*. *nodeName* is the suffix of S_N w.r.t. the sequence represented by the parent of N. *nodeUtil* is the approximate utility of S_N over the part of the data stream processed so far. *nodeRsu* holds the *rest utility* value (to be defined in the next section and used in memory adaptation) of S_N. For example, in Fig. 3a, the leftmost leaf node corresponds to pattern $\{abc\}$. Its *nodeName* is $\{\_c\}$ (which is the suffix of $\{abc\}$ w.r.t. its parent node sequence $\{ab\}$) and its *nodeUtil* and *nodeRsu* are 39 and 54, respectively. A D-node has only one field *nodeName*, storing the suffix of sequence it represents w.r.t. its parent sequence.

3.3 Rest utility: a utility upper bound

Before we present how a MAS-Tree is built and updated, we first define the *rest utility* of a sequence and prove that it is an upper bound on the true utilities of the sequence and all of its prefix super-sequences (*prefixSUPs*).

Definition 13 (*First occurrences of a sequence α in a sequence S_r*) Given a sequence $S_r = \langle S_r^1, S_r^2, \ldots, S_r^n\rangle$ and a sequence $\alpha = \langle X_1, X_2, \ldots, X_Z\rangle$, $\tilde{o} \in OccSet(\alpha, S_r)$ is the first occurrence of α in S_r, iff the last itemset in $\tilde{o}$ occurs sooner than the last itemset of any other occurrence in $OccSet(\alpha, S_r)$.

Definition 14 (*Rest sequence of S_r w.r.t. sequence α*) Given sequences $S_r = \langle S_r^1, S_r^2, \ldots, S_r^n\rangle$ and $\alpha = \langle X_1, X_2, \ldots, X_Z\rangle$, where $\alpha \preceq S_r$. The rest sequence of S_r w.r.t. α, is defined as: $restSeq(S_r, \alpha) = \langle S_r^m, S_r^{m+1}, \ldots, S_r^n\rangle$, where S_r^m is the last itemset of the first occurrences of α in S_r.

Definition 15 (*Rest utility of a sequence α in a sequence S_r*) The rest utility of α in S_r is defined as $rsu(\alpha, S_r) = su(\alpha, S_r) + su(restSeq(S_r, \alpha))$.

For example, given $\alpha = \langle\{ac\}\{c\}\rangle$ and S_1 in Fig. 1, $restSeq(S_1, \alpha) = \langle\{(b, 1)(c, 1)(d, 1)\}\{(c, 3)(d, 1)\}\rangle$. Hence, $su(restSeq(S_1, \alpha)) = 8 + 7 = 15$, then $rsu(\alpha, S_1) = su(\alpha, S_1) + 15 = \max\{7, 9\} + 15 = 24$.

Definition 16 (*Rest utility of a sequence α in dataset D*) The rest utility of a sequence α in a dataset D of sequences is defined as $rsu(\alpha, D) = \sum_{S_r \in D} rsu(\alpha, S_r)$.

Theorem 1 *The rest utility of a sequence α in a data stream DS is an upper-bound of the true utilities of all the prefixSUPs of α in DS. That is, $\forall \beta \gtrsim \alpha, su(\beta, DS) \leq rsu(\alpha, DS)$.*

Proof We prove that $rsu(\alpha, S_r)$ is an upper-bound of the true utilities of all the prefixSUPs of α in sequence S_r. The proof can be easily extended to batch B_k and data stream DS. Given sequence $\alpha = \langle X_1, X_2, \ldots, X_M\rangle$, and $\beta = \langle X_1, X_2, \ldots, X_M^{'}, X_{M+1}, \ldots, X_N\rangle$, where $X_M \lesssim X_M^{'}$. According to Definition 6:

$$su(\beta, S_r) = \max\{su(\beta, \tilde{o}) \mid \tilde{o} \in OccSet(\beta, S_r)\}$$

Thus,

$$\exists \tilde{o}, su(\beta, S_r) = su(\beta, \tilde{o}) \tag{1}$$

Sequence β can be partitioned into two sub-sequences: $\alpha = \langle X_1, X_2, \ldots, X_M\rangle$ and $\beta^{'} = \langle X_M^{'} - X_M, X_{M+1}, \ldots, X_N\rangle$. The Eq. 1 can be rewritten as follows:

$$\begin{aligned} &\exists \tilde{o}_\alpha \in OccSet(\alpha, S_r) \text{ and } \exists \tilde{o}_{\beta'} \in OccSet(\beta^{'}, S_r - \tilde{o}_\alpha), \\ &su(\beta, S_r) = su(\alpha, \tilde{o}_\alpha) + su(\beta^{'}, \tilde{o}_{\beta'}) \end{aligned} \tag{2}$$

$$\text{Also, } \forall \tilde{o}_\alpha \in OccSet(\alpha, S_r), su(\alpha, \tilde{o}_\alpha) \leq su(\alpha, S_r) \tag{3}$$

Similarly,

$$\forall \tilde{o}_{\beta'} \in OccSet\left(\beta^{'}, S_r - \tilde{o}_\alpha\right), su(\beta^{'}, \tilde{o}_{\beta'}) \leq su\left(\beta^{'}, S_r - \tilde{o}_\alpha\right) \tag{4}$$

where $S_r - \tilde{o}_\alpha$ is a sequence consisting of all itemsets in S_r which occur after the last itemset in $\tilde{o}_\alpha$. Since $S_r - \tilde{o}_\alpha \preceq restSeq(S_r, \alpha)$, hence:

$$\begin{aligned} su(\beta^{'}, \tilde{o}_{\beta'}) &\leq su(\beta^{'}, S_r - \tilde{o}_\alpha) \leq su(\beta^{'}, restSeq(S_r, \alpha)) \\ &\leq su(restSeq(S_r, \alpha)) \end{aligned} \tag{5}$$

From (3) and (5):

$$\begin{aligned} su(\beta, S_r) &= su(\alpha, \tilde{o}_\alpha) + su(\beta^{'}, \tilde{o}_{\beta'}) \\ &\leq su(\alpha, S_r) + su(restSeq(S_r, \alpha)) = rsu(\alpha, S_r). \end{aligned}$$

3.4 MAS-Tree construction and updating

The tree starts empty. Once a potential HUSP is found in a batch, it is added to the tree. Given a potential HUSP S in batch B_k, the first step is to find node N whose corresponding sequence S_N is either S or the longest *prefixSUB* of S in *MAS-Tree*. Let $su(S, B_k)$ be the exact utility value of S in the batch B_k and $rsu(S, B_k)$ be the rest utility value of S in the batch B_k. If S_N is S and N is a C-node, then $nodeUtil(N)$ and $nodeRsu(N)$ are updated by adding $su(S, B_k)$ and $rsu(S, B_k)$ respectively. If N is a D-node, it is converted to a C-node and $nodeUtil(N)$ and $nodeRsu(N)$ are initialized by $su(S, B_k)$ and $rsu(S, B_k)$ respectively.

If S_N is the longest *prefixSUB* of S, new node(s) are created to insert S into the tree. In this situation, there are three cases:

1. Node N has a child node *CN* where $S \lesssim S_{CN}$: for example, in Fig. 3a, if pattern $S = \langle\{b\}\{a\}\rangle$, node N with $S_N = \{b\}$ is found. N has a child node *CN* where $S_{CN} = \langle\{b\}\{ab\}\rangle$ and $S \lesssim S_{CN}$. In this case, a new C-node C is created as child of N and parent of *CN* where $nodeName(C)$ is the suffix S w.r.t. S_N. Then $nodeUtil(C)$ and $nodeRsu(C)$ are initialized by $su(S, B_k)$ and $rsu(S, B_k)$. Also $nodeName(CN)$ is updated w.r.t. S_C. In our example, a new node is created with $\{a\}$, 20, and 36 as *nodeName*, *nodeUtil* and *nodeRsu*, respectively (see Fig. 3b).
2. Node N has a child node *CN* where S_{CN} *contains* (but not exactly is) a longer *prefixSUB* (i.e., S_{prefix}) of S than S_N: For example, in Fig. 3b, given pattern $S = \langle\{b\}\{bc\}\{d\}\rangle$, $su(S, B_1) = 17$ and $rsu(S, B_1) = 17$, node N with $S_N = \langle\{b\}\{b\}\rangle$ is found. Its child node *CN* where $S_{CN} = \langle\{b\}\{bc\}\{b\}\rangle$ contains a longer *prefixSUB* of S, $S_{prefix} = \langle\{b\}\{bc\}\rangle$. In this case, since S_{prefix} is the longest common *prefixSUB* of S and S_{CN}, a new D-node D corresponding to S_{prefix} is created as child of N and parent of *CN*. Then a new C-node C is created as child of D where $nodeName(C)$ is the suffix of S w.r.t. S_{prefix}. Its *nodeUtil* and *nodeRsu* are initialized by su(S,B_k) and rsu(S,B_k) respectively. Also $nodeName(CN)$ is updated w.r.t. S_D. In the example, node D with $nodeName(D) = \langle\{\_c\}\rangle$ is added as child of N and parent of *CN*, and also node C where $nodeName(C) = \langle\{d\}\rangle$ is created as child of D.
3. None of the above cases: for example in Fig. 3b, given pattern $S = \langle\{ab\}\{bc\}\{d\}\rangle$ whose utility is 21 and rest utility is 21, node N with $S_N = \{ab\}$ is found. Its child node does not contain S or a longer *prefixSUB* of S. In this case, a new C-node C is created as child of N where $nodeName(C)$ is the suffix of S w.r.t. S_N. Also, $nodeUtil(C)$ and $nodeRsu(C)$ are initialized by $su(S, B_k)$ and $rsu(S, B_k)$. In the example, node C where $nodeName(C) = \langle\{bc\}\{d\}\rangle$, $nodeUtil(C) = 21$ and $nodeRsu(C) = 21$ is created as child of N.

Figure 3b shows the updated tree after inserting three patterns $\langle\{b\}\{a\}\rangle$, $\langle\{ab\}\{bc\}\{d\}\rangle$ and $\langle\{b\}\{bc\}\{d\}\rangle$ to MAS-Tree presented in Fig. 3a.

Algorithm 2 shows the complete procedure for inserting the potential HUSPs found in batch B_k to the tree. It first updates the tree using the patterns in $HUSP_{B_k}$ that already exist in the tree. This is to avoid the memory adaption procedure from pruning nodes that will be inserted again soon in the same batch. For each pattern S in $HUSP_{B_k}$, Algorithm 2 finds node N where S_N is either S or the longest *prefixSUB* of S in the tree. If S_N is S, Algorithm 2 updates values of $nodeRsu(N)$ and $nodeUtil(N)$ accordingly. If $nodeName(N)$ is the longest *prefixSUB* of S, the pattern S and node N are inserted into $newPatSet_{B_k}$. Each pair in $newPatSet_{B_k}$ consists of a new pattern and a pointer to the node associated to the longest *prefixSUB* of the pattern in the tree. After the tree is updated using the existing patterns, for each pair $\langle S, N\rangle$ in $newPatSet_{B_k}$, the pattern S is inserted into the tree, in which Algorithm 3 is called to create a node for the tree in a memory adaptive manner described below.

3.5 Memory adaptive mechanisms

When inserting a new node in *MAS-Tree*, if the memory constraint is to be violated, our algorithm will remove some tree nodes to release memory. An intuitive approach to releasing memory is to blindly eliminate some nodes from the tree. However, this approach could remove nodes representing high quality HUSPs and make the mining results highly inaccurate. Below we propose two memory adaptive mechanisms to cope with the situation when memory space is not enough to insert a new potential HUSP in the tree. Our goal is to effi-

Algorithm 2 *Insert potential HUSPs into MAS-Tree*

```
Input: $MAS\text{-}Tree$, $HUSP_{B_k}$, $mechanismType$, $availMem$
Output: $MAS\text{-}Tree$, $currMem$
1: $newPatSet_{B_k} \leftarrow \emptyset$
2: for $\forall S \in HUSP_{B_k}$ do
3:   $N \leftarrow$ The node with the longest $prefixSUB$ of $S$ in MAS-Tree
4:   if $S_N$ is the same as $S$ then
5:     if $N$ is C-node then
6:       $nodeRsu(N) \leftarrow nodeRsu(N) + rsu(S, B_k)$
7:       $nodeUtil(N) \leftarrow nodeUtil(N) + su(S, B_k)$
8:     else
9:       Convert $N$ to C-node
10:      $nodeRsu(N) \leftarrow rsu(S, B_k) + maxUtil$
11:      $nodeUtil(N) \leftarrow su(S, B_k) + maxUtil$
12:    end if
13:  else
14:    Add pair $\langle S, N\rangle$ to $newPatSet_{B_k}$
15:  end if
16: end for
17: for $\forall\langle S, N\rangle \in newPatSet_{B_k}$ do
18:   $CN \leftarrow$ A child of node $N$ with longer common prefixSUB (i.e.,$S_{prefix}$) of $S$ than $S_N$
19:   if $CN$ does not exist then
20:     $S_C \leftarrow$ suffix of $S$ w.r.t. $S_N$
21:     Call Algorithm 3 to create C-node $C$ as a child of $N$
22:   else
23:     if $S_{prefix}$ is $S$ then
24:       $S_C \leftarrow$ suffix of $S$ w.r.t. $S_N$
25:       Call Algorithm 3 to create C-node $C$ as a child of $N$
26:     else
27:       $S_D \leftarrow$ suffix of $S_{prefix}$ w.r.t. $S_N$
28:       Call Algorithm 3 to create D-node $D$ as a child of $N$
29:       $S_C \leftarrow$ suffix of $S$ w.r.t. $S_{prefix}$
30:       Call Algorithm 3 to create C-node $C$ as a child of $D$
31:     end if
32:   end if
33: end for
34: return $MAS\text{-}Tree$
```

ciently determine the nodes for pruning, without sacrificing too much the accuracy of the discovered HUSPs.

Mechanism 1 [*Leaf Based Memory Adaptation* (*LBMA*)] Given a *MAS-Tree*, a pattern S and the available memory *availMem*, if the required memory to insert S is not available, *LBMA* iteratively prunes the leaf node N with minimum $nodeUtil(N)$ among all the leaf nodes until the required memory is released.

Rationale (1) A leaf node is easily accessible and we do not need to scan the whole tree to find a node with low utilities. (2) A leaf node does not have a child, so it can be pruned easily without reconnecting its parent to its children. (3) In case a great portion of nodes in the tree are leaf nodes, leaf nodes with minimum utilities have low likelihood to become a HUSP. Later we prove that *LBMA* is an effective mechanism so that all true HUSPs stay in the tree under certain circumstances.

The second mechanism releases memory by pruning a sub-tree from *MAS-Tree*.

Mechanism 2 [*Sub-Tree Based Memory Adaptation* (*SBMA*)] Given a *MAS-Tree*, a pattern S and available memory *availMem*, if the required memory to insert S is not available, *SBMA* iteratively finds node N with minimum rest utility (*nodeRsu*(N)) in *MAS-Tree* and prunes the sub-tree rooted at N from *MAS-Tree* till the required memory is released.

Rationale Since in *MAS-Tree* a descendant of a node N represents a prefix super-sequence (*prefixSUP*) of the pattern represented by N, according to Theorem 1, the rest utility of N (*nodeRsu*(N)) is an upper bound of the true utilities of all its descendants. Therefore, if node N has a minimum rest utility, not only the pattern represented by N is less likely to become HUSP, but also all of its descendants are less likely to become HUSPs. Thus, we can effectively remove all the nodes in the subtree rooted at N. Similar to LBMA, with this mechanism there is no need to reconnect N's parent with its children.

Algorithm 3 shows how the proposed memory adaptive mechanisms are incorporated into node creation. It removes some nodes based on either LBMA or SBMA mechanism when there is not enough memory for a new node. In addition, the following two issues are addressed in this procedure.

Approximate utility When some C-nodes are removed from the tree, the potential HUSPs represented by the removed nodes are discarded. If a removed pattern is a potential HUSP in the new batch, the pattern will be added into the tree again. But its utility value in the previous batches is not recorded due the node removal. To compensate this situation, we keep track of the maximum value of the *nodeUtil* or *nodeRsu* of all the removed nodes, and add it to the *nodeUtil* and *nodeRsu* of a new C-node. The maximum value is denoted as *maxUtil* in Algorithms 2 and 3.

Node merging If the parent of a removed leaf node or subtree is a D-node and the parent has a single child left after the node removal, the parent and its child are merged into a single node (Lines 16–17 in Algorithm 3). This is to make the tree compact and maintain the property of MAS-Tree (i.e., each node represents the longest common prefixSUB of its descendants). Note that our strategy to remove either a subtree or a leaf node allows us to maintain the MAS-Tree structure using such minimum adjustments.

Let L_{avg} and *NumPot* be the average length and the number of potential HUSPs respectively. The time complexity to find the node N to insert a pattern as its child is $O(NumPot \times L_{avg})$. For *LBMA*, the time complexity for initializing and updating is $O(NumPot)$. The time complexity to apply *SBMA* is $O(NumPot \times L_{avg})$.

3.6 Mining HUSPs from MAS-Tree

As the data stream evolves, when the user requests to find HUSPs on the stream so far, MAHUSP traverses the MAS-Tree once and returns all the patterns represented by a node whose *nodeUtil* is no less than $(\delta - \epsilon) \cdot U_{DS_k}$, where $DS_k = \langle B_1, B_2, \ldots, B_k \rangle$ is the stream processed so far. The reason for using this threshold is that a potential HUSP in a batch B_i may not be a potential pattern in batch B_j and thus its utility in batch B_j is not recorded in the tree. However, since when we mine B_j for potential HUSP, $\epsilon \cdot U_{B_j}$ is used as the threshold, the true utility of a non-potential pattern in B_j can not be higher than $\epsilon \cdot U_{B_j}$. Thus, $nodeUtil(N) + \epsilon \cdot U_{DS_k}$ is an over-estimate for the approximate utility of the pattern represented by node N. Finding nodes whose $nodeUtil(N) + \epsilon \cdot U_{DS_k} \geq \delta \cdot U_{DS_k}$ is equivalent to finding those with $nodeUtil(N) \geq (\delta - \epsilon) \cdot U_{DS_k}$.

3.7 Correctness

Given a data stream DS, a sequence α and a node $N \in$ *MAS-Tree* where S_N is α, let $su_{tree}(\alpha, DS)$ be $nodeUtil(N)$ when *availMem* is infinite and there is no pruning, and let $su_{apprx}(\alpha, DS)$ be $nodeUtil(N)$ when *availMem* is limited and pruning occurs.

Algorithm 3 *Memory Adaptive Node Creation*

Input: S, $su(S, B_k)$, $rsu(S, B_k)$, $nodeType$, $mechanismType$, P (parent of the node to be created)
Output: node N, $maxUtil$

```
1: $currMem \leftarrow$ current memory usage
2: $reqMem \leftarrow$ memory usage for a node of $nodeType$ for pattern $S$
3: while $currMem + reqMem \geq availMem$ do
4:   if mechanismType is LBMA then
5:     Remove the leaf node $node$ with minimum $nodeUtil(node)$
6:     $maxUtil \leftarrow nodeUtil(node)$;
7:     Adjust the amount of current available memory $currMem$
8:   end if
9:   if mechanismType is SBMA then
10:     Remove the subtree rooted by $node$ with minimum $nodeRsu(node)$
11:     $maxUtil \leftarrow nodeRsu(node)$
12:     Adjust the amount of current available memory $currMem$
13:   end if
14:   if parent $P$ of $node$ has a single child $C$ then
15:     Merge $P$ and $C$ into one node
16:     Adjust the amount of current available memory $currMem$
17:   end if
18: end while
19: if parent $P$ of $node$ has been removed then
20:   Call Algorithm 2 to get a new parent node and exit
21: end if
22: Create node $N$ with pattern $S$
23: if $nodeType$ is a C-node then
24:   $nodeRsu(N) \leftarrow rsu(S, B_k) + maxUtil$
25:   $nodeUtil(N) \leftarrow su(S, B_k) + maxUtil$
26: end if
27: currMem $\leftarrow currMem + reqMem$
28: return $N$, $maxUtil$
```

Lemma 1 *Given a potential HUSP α, the difference between the exact utility of α and its utility in MAS-Tree is bounded by $\epsilon \cdot U_{DS}$ when availMem is infinite. That is, $su(\alpha, DS) - su_{tree}(\alpha, DS) < \epsilon \cdot U_{DS}$.*

Proof According to Definition 7, $su(\alpha, DS) = \sum_{B_j \in DS} su(\alpha, B_j)$. Given batch $B_k \in DS$, if $su(\alpha, B_k) < \epsilon \cdot U_{B_k}$, then α is not returned by *USpan*. In this case $\epsilon \cdot U_{B_k}$ is an upper bound on utility of α in the batch B_k. Hence, $su(\alpha, DS) - su_{tree}(\alpha, DS) = \sum_{B_m \in BSet} su(\alpha, B_m) < \epsilon \cdot \sum_{B_k \in DS} U_{B_k} \leq \epsilon \cdot U_{DS}$, where *BSet* is the set of all batches that α is not returned by *USpan*.

Lemma 2 *Given potential HUSP α, the current MAS-Tree and C-node C where S_C is α, $su_{tree}(\alpha, DS) \leq su_{apprx}(\alpha, DS)$.*

Proof If node C is never pruned, then $su_{apprx}(\alpha, DS) = su_{tree}(\alpha, DS)$ which is $nodeUtil(C)$. Otherwise, since we have a node with pattern α in the tree, C has been added back to the tree after removal. Assume that, when C was pruned from MAS-Tree, the value of *maxUtil* was denoted as $maxUtil_1$, and when C with pattern α was re-inserted, the value of *maxUtil* was denoted as $maxUtil_2$. According to Algorithm 2, the value of *maxUtil* never gets smaller. Hence $maxUtil_1 \leq maxUtil_2$. Once C was re-inserted into the tree, $nodeUtil(C)$ was incremented by $maxUtil_2$ which is bigger than or equal to $maxUtil_1$. Since $su_{apprx}(\alpha, DS) = nodeUtil(C)$, $su_{apprx}(\alpha, DS) \geq su_{tree}(\alpha, DS)$.

Lemma 3 *For any potential HUSP α, if $su_{tree}(\alpha, DS) > maxUtil$, α must exist in MAS-Tree.*

Proof We prove it by contradiction. Assume that there is a HUSP β, where $maxUtil < su_{tree}(\beta, DS)$, and node N with $nodeName(N) = \beta$ does not exist in the tree. Since $0 \leq maxUtil < su_{tree}(\beta, DS)$, at some point, β was inserted to the tree. Otherwise, $su_{tree}(\beta, DS) = 0$. Since N does not exist in MAS-Tree, it must have been pruned afterwards. Let $util_{old}$ and rsu_{old} denote $nodeUtil(N)$ and $nodeRsu(N)$ respectively when N was last pruned. Based on Lemma 2, $su_{tree}(\beta, DS) \leq su_{apprx}(\beta, DS)$, where $su_{apprx}(\beta, DS) = util_{old}$, at the time N was pruned. So $su_{tree}(\beta, DS) \leq util_{old}$. Based on the memory adaptive mechanisms, $util_{old} \leq maxUtil$. Hence, $su_{tree}(\beta, DS) \leq util_{old} \leq maxUtil$, which contradicts the assumption. Thus, if $maxUtil < su_{tree}(\alpha, DS)$, α is in the tree.

Theorem 2 *Once the user requests HUSPs over the data stream DS, if* $maxUtil \leq (\delta - \epsilon) \cdot U_{DS}$*, all the high utility sequential patterns will be returned.*

Proof Given a high utility sequential pattern α, according to Definition 7, $su(\alpha, DS) \geq \delta \cdot U_{DS}$. On the other hand, based on Lemma 1, $\epsilon \cdot U_{DS} > su(\alpha, DS) - su_{tree}(\alpha, DS)$. Thus, $\epsilon \cdot U_{DS} + su_{tree}(\alpha, DS) > su(\alpha, DS)$.
According to Definition 7, $\epsilon \cdot U_{DS} + su_{tree}(\alpha, DS) > \delta \cdot U_{DS}$. Hence:

$$su_{tree}(\alpha, DS) > (\delta - \epsilon) \cdot U_{DS} \geq maxUtil.$$

According to Lemma 3, α must exist in the tree. On the other hand, based on Lemma 2, $su_{apprx}(\alpha, DS) \geq su_{tree}(\alpha, DS) > (\delta - \epsilon) \cdot U_{DS}$. Hence, α will be returned by the algorithm.

While this theory only guarantees the perfect recall in certain situations, in the next section we will show that our algorithm will return HUSPs with both high recall and high precision in practice.

4 Experiments

In this section, the proposed algorithms are evaluated. All the algorithms are implemented in Java. The experiments are conducted on an Intel(R) Core(TM) i7 2.80 GHz computer with 16 GB of RAM.

4.1 Datasets

To evaluate the performance of our proposed algorithms, experiments have been conducted on two synthetic datasets and two real datasets:

1. *DS1:D10K-C10-T3-S4-I2-N1K* this dataset is generated by the IBM data generator (Agrawal and Srikant 1995). The parameters in DS1 mean that the number of sequences in the dataset is 10K, the average number of transactions in a sequence is 10, the average number of items in a transaction is 3, the average length of a maximal pattern consists of 4 itemsets and each itemset is composed of 2 items average. The number of items in the dataset is 1k.
2. *DS2:D100K-C8-T3-S4-I2-N10K* the second synthetic dataset is also generated by the IBM data generator (Agrawal and Srikant 1995). In this dataset, the number of sequences is 100K and the average number of transactions in a sequence is 8. Similar to DS1, the average number of items in a transaction is 3, the average length of a maximal pattern consists of 4 itemsets and each itemset is composed of 2 items average. In this dataset, the number of items in the dataset is 10k.

Table 2 Dataset characteristics

Name	#Seq(K)	Avg. Trans. Length	Type	batchSize(K)	availMem (MB)
DS1	10	10	Dense	1	100
Kosarak	25	3	Sparse, large	5	200
DS2	100	8	Dense, large	10	400
ChainStore	400	7.26	Dense, large	50	800

3. *Kosarak* this dataset contains web click-stream data of a Hungarian on-line news portal (Fournier-Viger et al. 2013), which is a sparse dataset. The average length of transaction in this dataset is 3.
4. *ChainStore* this dataset is a real-life dataset acquired from Pisharath et al. (2012), which already contains internal and external utilities. In order to use this dataset as a sequential dataset, we randomly grouped transactions into sequences so that each group represents a sequence of transactions and each sequence contains 4–8 transactions.

We follow previous studies (Ahmed et al. 2010; Zihayat et al. 2015) to generate internal and external utilities of items in the first three datasets. The external utility of each item is generated between 1 and 100 by using a log-normal distribution and the internal utilities of items in a transaction are randomly generated between 1 and 100.

Table 2 shows dataset characteristics and parameter settings in the experiments. We set *availMem* heuristically based on the average memory to store the data structures used by *USpan* and the average memory used by *MAS-Tree* over the datasets. The significance threshold (i.e., ϵ) is set as $0.5 \times \delta$. For example, in DS2, when $\delta = 0.0009$, $\epsilon = 0.00045$. We will later change these parameters (i.e., *availMem*, *batchSize* and significance threshold) to show the performance of the algorithms under different parameter values.

In Sect. 4.8, to demonstrate the effectiveness and efficiency of *MAHUSP* on real-life applications, we first conduct an analysis on a real web clickstream dataset obtained from a Canadian news portal. Then, we apply our method to identify gene regulation sequential patterns correlated with a specific disease from to a publicly available time course microarray dataset.

4.2 Methods in comparison

To the best of our knowledge, no method was proposed to mine HUSPs over a data stream in a memory adaptive manner. Therefore, the following methods are implemented to compare with the proposed methods:

1. *NaiveHUSP* this method is a fast method to approximate the utility of a sequence over the past batches using the utility of items in the sequence. That is, the utility of each item over a data stream is tracked. Once a new batch arrives, the utility value of each item in the new batch is updated. If the user requests HUSPs, the algorithm runs *USpan* to find all HUSPs in the current batch B_i. Then for each pattern α, the utility of α over the data stream is calculated as follows: $su(\alpha, DS_i) = su(\alpha, B_i) + \sum_{I \in \alpha} u(I, DS_{i-1})$, where $su(\alpha, B_i)$ is the utility of α in the batch B_i and $u(I, DS_{i-1})$ is the utility of item I over the data stream before batch B_i arrives.
2. *RndHUSP* this method is similar to *MAHUSP* and is a memory-adaptive *HUSP* mining approach over data streams. It runs *USpan* to find HUSPs in each batch and inserts them to MAS-Tree. When a new node is inserted in *MAS-Tree*, if the memory constraint is to

be violated, *RndHUSP* will remove some nodes to release memory. The mechanism to releasing memory is to randomly eliminate some nodes from the tree.
3. *USpan* once a user requests HUSPs, *USpan* is run on the whole data stream (i.e., DS_i) seen so far using $\delta \cdot U_{DS_i}$ as the utility threshold to find the true set of HUSPs (i.e., eHUSP).

Moreover, in order to see the effect of different memory adaptive mechanisms, we evaluate two versions of *MAHUSP*, named *MAHUSP_S* (which uses the *SBMA* mechanism) and *MAHUSP_L* (which uses the *LBMA* mechanism).

4.3 Performance measures

Given *eHUSP* as the true set of HUSPs and *appHUSPs* as the approximate set of HUSPs returned by a method, we use the following performance measures:

– *Precision* the average precision over data streams:

$$precision = \frac{|appHUSPs \bigcap eHUSP|}{|appHUSPs|}$$

– *Recall* the average recall values over data streams:

$$recall = \frac{|appHUSPs \bigcap eHUSP|}{|eHUSP|}$$

– *F-Score* the harmonic mean of *precision* and *recall*:

$$F\text{-}Score: 2 \times \frac{precision \times recall}{precision + recall}$$

– *AvgTrueUtil* the average true utility of patterns returned by a method.
– *Relative Utility Error* The relative utility error is computed as the average utility error compared with the exact utility of patterns returned by a method:

$$Relative\ Utility\ Error = \frac{\sum_{P \in appHUSPs} \frac{ApproxUtil(P) - TrueUtil(P)}{TrueUtil(P)}}{|appHUSPs|},$$

where $TrueUtil(P)$ is the true utility of P in the dataset and $ApproxUtil(P)$ is the approximate utility of P returned by the method.
– *AvgLength* the average number of items in patterns returned by a method is another quality measure that we use in our experiments. Since *MAHUSP_L* iteratively prunes the leaf nodes in the MAS-Tree, it may imply that patterns with long length may be missed. Using this measure, we show that the average length of the patterns returned by our methods is close to that of obtained by the exact method.
– *Run Time* the total execution time of a method over the input data stream.
– *Memory Usage* the memory consumption of a method.

4.4 Effectiveness of MAHUSP

In this section, the effectiveness of the methods is evaluated. For consistency across datasets, in all figures presented in this section the minimum threshold is shown as a percentage (i.e., δ) of the total utility of the current data stream in the dataset.

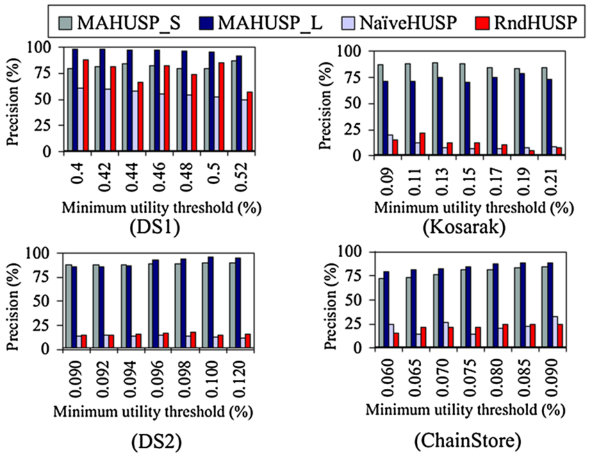


Fig. 4 Precision performance on the different datasets

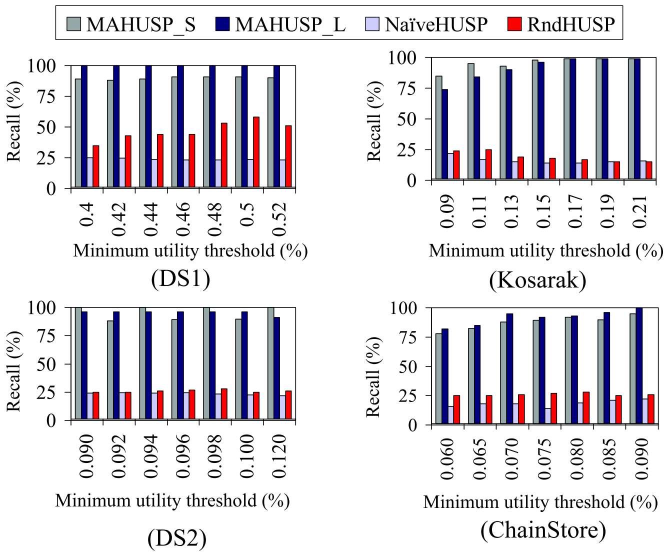


Fig. 5 Recall performance on the different datasets

4.4.1 Precision, Recall and F-Score

Figure 4 shows the *precisions* of the methods on the three datasets. The proposed methods are more effective than *NaiveHUSP* and *RndHUSP* in terms of precisions. *MAHUSP_L* outperforms *MAHUSP_S* in the most of the cases in *DS*1 and *DS*2. This is because the approximate utility by *LBMA* is usually tighter than the one by *SBMA*, and thus there are fewer false positives in the results.

Figure 5 shows the *recalls* of the methods, which indicate that our proposed methods are much more effective than the other methods in all the datasets. Indeed, on *DS*1, *MAHUSP_L* returns all the true patterns for each threshold value. Also, *MAHUSP_S* returns all the true patterns for most threshold values on *DS*2 and *Kosarak*. *MAHUSP_L* and *MAHUSP_S* could

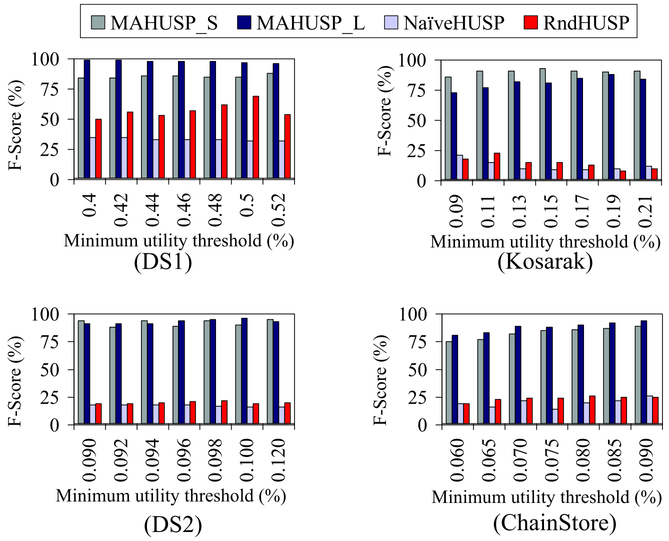


Fig. 6 F-Score performance on the different datasets

also achieve better performance than *NaiveHUSP* and *RndHUSP* on *ChainStore* dataset. The results imply that the condition presented in Theorem 2 happens often and the proposed memory adaptive mechanisms prune the nodes effectively. The average recall values of *MAHUSP_S* on *DS*1, *DS*2, *Kosarak* and *ChainStore* datasets are 87, 95, 92 and 87% respectively. These values for *MAHUSP_L* are 100, 94, 92 and 91% respectively.

Figure 6 shows the *F-Score* values for the four methods with different δ values on the 4 datasets. In most of the cases, *NaiveHUSP* is the worst among the four methods. This is because it estimates the utility of a sequence based on the utility of each item over the data stream which is not an accurate approximation. Both proposed methods outperform the other methods with an average F-score value of 90% over the *DS*1, *DS*2 and *Kosarak* datasets and 87% over the *ChainStore* dataset.

4.4.2 Average pattern length

Figure 7 shows the average length of patterns returned by the methods on different datasets. As it is shown, the *AvgLength* values obtained by the *MAHUSP_L* and *MAHUSP_S* are similar to those of returned by USpan. Since *MAHUSP_L* prunes leaf nodes to release memory, its *AvgLength* is generally lower than *MAHUSP_S* for different utility threshold values. Moreover, *NaiveHUSP* usually returns long patterns, since more items in the sequence leads higher approximate utility, thus longer patterns are usually returned as HUSPs. In most of the cases, *RndHUSP* returns patterns whose length is longer than those of returned by the exact method. However, since the method prunes patterns randomly, in some cases (e.g., in DS1 when the threshold is set to 0.44%) the average length of patterns is lower than the exact method's.

4.4.3 Average true utility

In this section, we evaluate the quality of patterns returned by the methods in terms of the true utility of the patterns. Since the methods are approximate methods, the output set usually includes some or all of the true HUSP and some patterns which may not returned

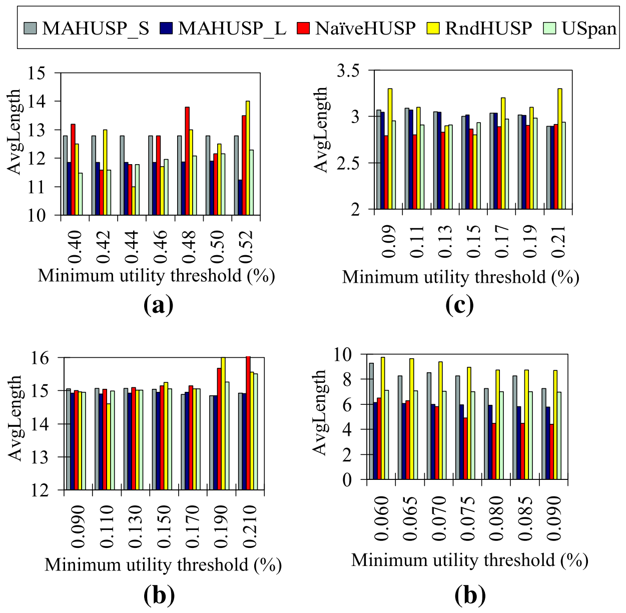


Fig. 7 Average length of patterns returned by the methods on the different datasets. **a** DS1, **b** DS2, **c** Kosarak, **d** ChainStore

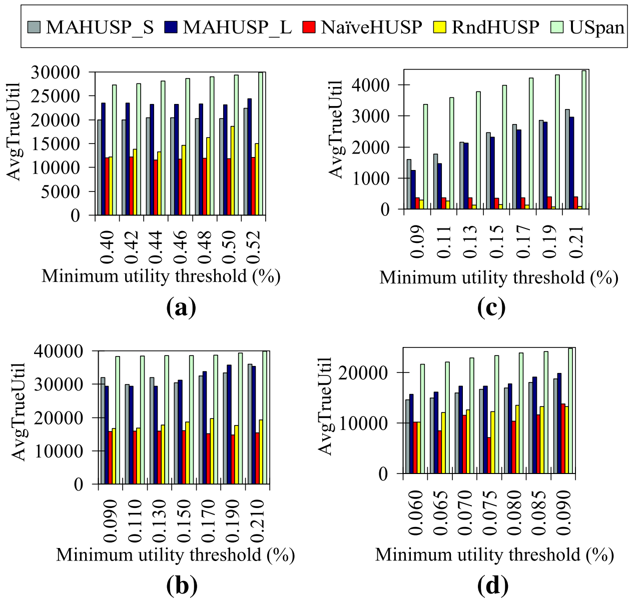


Fig. 8 Average true utility of patterns returned by the methods on the different datasets. **a** DS1, **b** DS2, **c** Kosarak, **d** ChainStore

by an exact method (e.g., USpan). In this section, we show that the average true utility of patterns returned by MAHUSP is relatively high and is comparable to those of discovered by USpan. Figure 8 shows the results in terms of *AvgTrueUtil*. In this figure, *MAHUSP_L*

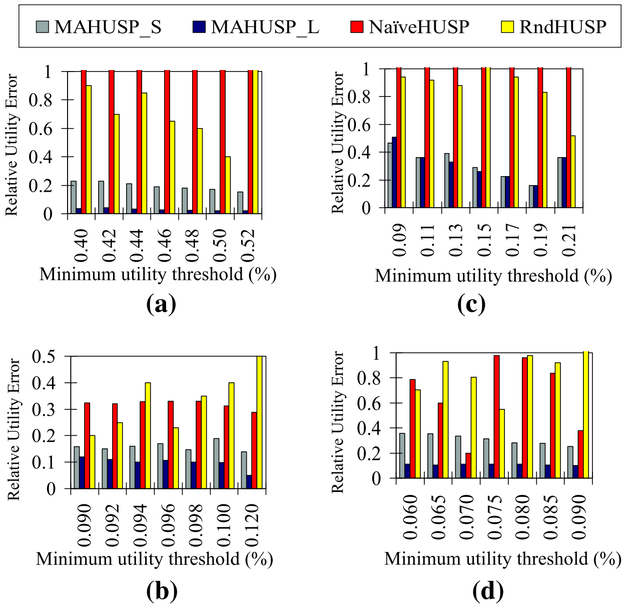


Fig. 9 Relative utility error performance on the different datasets. **a** DS1, **b** DS2, **c** Kosarak, **d** ChainStore

and *MAHUS_S* outperform *NaiveHUSP* and *RndHUSP*. For example in DS1, the average true utility of patterns returned by USpan is 25,000, while this value for *MAHUSP_L* and *MAHUSP_S* is 23,000 and 20,000 respectively. This value for *NaiveHUSP* and *RndHUSP* is 10,000 and 13,000 respectively.

4.4.4 Relative utility error

In this section, we evaluate the effectiveness of the approximate utility approach against the exact results and provide the *relative utility error* of our methods on the four datasets. The *relative utility error* values are provided for different minimum utility threshold values and different datasets, in Fig. 9. In the figure, all values greater than 1 (which result from very inaccurate approximate utility values) are shown as 1 to keep the figures more readable. Since *MAHUSP_L* releases the memory based on the utilities stored in the leaf nodes, its approximation is usually tighter than *MAHUSP_S* which approximate the utility based on rsu values. As expected, the proposed methods outperform *NaiveHUSP* and *RndHUSP* and the average values for *MAHUSP_L* on $DS1$, $DS2$, *Kosarak* and *ChainStore* are 0.03, 0.1, 0.4 and 0.1 respectively and for *MAHUSP_S* are 0.2, 0.15, 0.3 and 0.38 respectively.

4.5 Efficiency of MAHUSP

4.5.1 Running time

Figure 10 shows the execution time of each method with different threshold values. Since *NaiveHUSP* only stores and updates the utility of each item over data streams, it is the fastest method. However, it generates a high rate of false positives due to its poor utility approximation. *MAHUSP_L* is slower than *MAHUSP_S*, since it prunes the tree node by

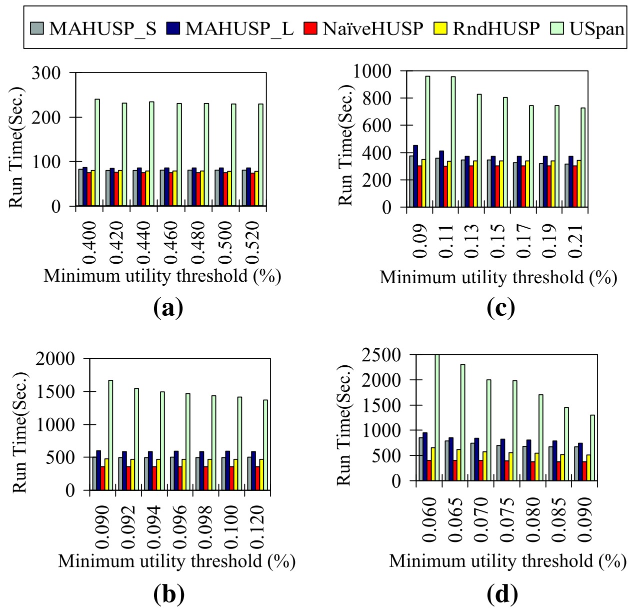


Fig. 10 Execution time on different datasets. **a** DS1, **b** DS2, **c** Kosarak, **d** ChainStore

node. *RndHUSP* is slower than *NaiveHUSP* since it updates the tree instead of utility of items. But, it is a bit faster than the proposed method because the pruning strategy is a random approach. Although these methods are faster than *MAHUSP_L* and *MAHUSP_S*, they produce high rate of false positive and false negative patterns. *USpan* is the slowest, whose run time indicates the infeasibility of using a static learning method on data streams although it returns the exact set of HUSPs. The results show the unreasonable run time of the exact method and its inapplicability in practice. Moreover, *MAHUSP* methods are only a bit slower than random pruning method (*RndHUSP*). Considering the big difference between them in precision and recall, it is very worthwhile to use the pruning strategies proposed in this paper.

4.5.2 Memory usage

Figure 11 shows the memory consumption of the methods on different datasets for different values of δ. *USpan* is the most memory consuming method since it needs to keep whole sequences in the memory. The memory usage of *NaiveHUSP* depends on the number of promising items in the dataset. For example, *NaiveHUSP* uses more memory than *MAHUSP_S* on *Kosarak* because this dataset is a sparse dataset and *NaiveHUSP* stores a huge list of items and their utilities into the memory. Regardless of the threshold value and the type of dataset, *MAHUSP_S* and *MAHUSP_L* guarantee that memory usage is bounded by the given input parameter *availMem*.

4.6 Run time performance over incoming batches

Figure 12 shows the performance of the methods over the batches as the data stream evolves. In this figure, the x-axes presents the number of batches in the current data stream and the y-

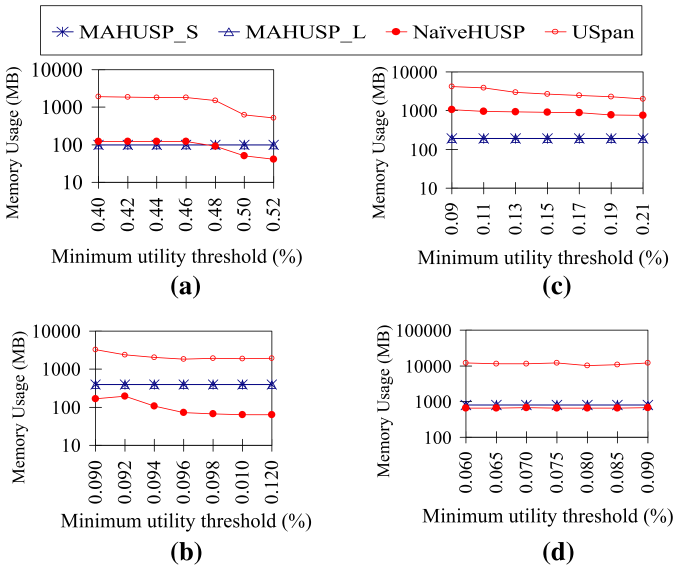


Fig. 11 Memory usage of the algorithms (shown in logarithmic scale). **a** DS1, **b** DS2, **c** Kosarak, **d** ChainStore

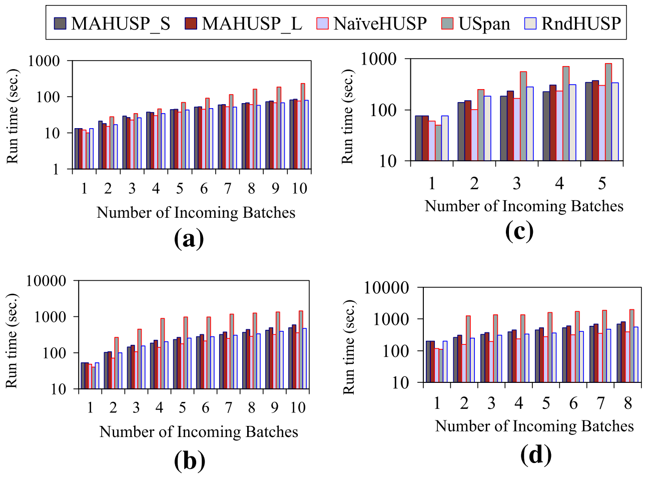


Fig. 12 Run time performance over incoming batches (shown in the logarithmic scale). **a** DS1, **b** DS2, **c** Kosarak, **d** ChainStore

axes presents the accumulated run time for different methods. The minimum utility threshold is set to 0.46, 0.96, 0.15 and 0.75% on *DS1*, *DS2*, *Kosarak* and *ChainStore* respectively. *MAHUSP_S*, *MAHUSP_L* and *RndHUSP* have the same run time on the first batch since they all run USpan to find potential HUSPs and insert them to MAS-Tree. Their performances are different in the subsequent batches since they use different memory adaptive mechanisms. In the first batch, *NaiveHUSP* and *USpan* are faster than the other methods since they do not insert patterns to the tree. However, as the stream evolves, *USpan* become slower since it runs on the whole current stream to discover HUSPs. In general, *NaiveHUSP* is the fastest

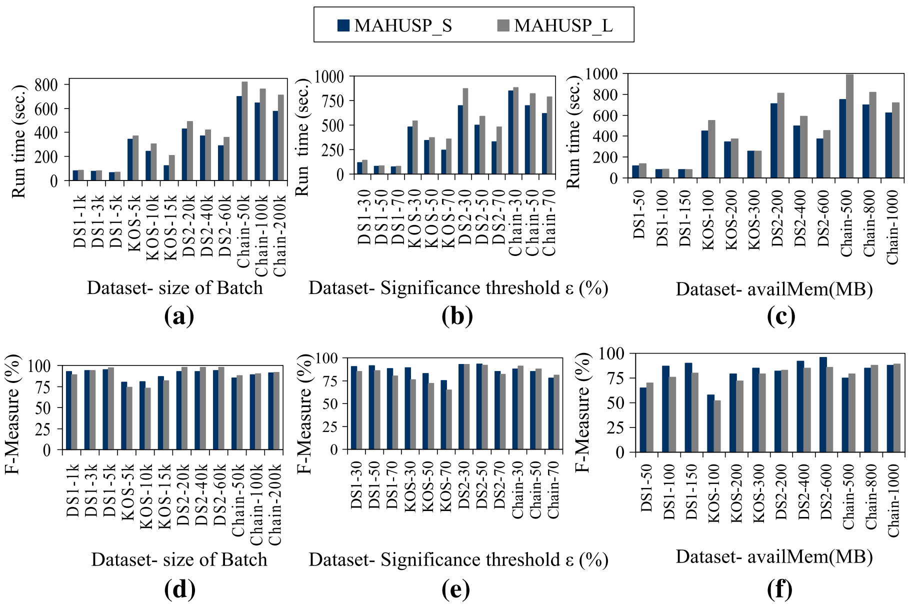


Fig. 13 Parameter sensitivity on different datasets

method. Moreover, since *MAHUSP_S*, *MAHUSP_L* and *RndHUSP* apply memory adaptive mechanisms in the consecutive batches they are slower than *NaiveHUSP*. *MAHUSP_S* works a bit faster than *MAHUSP_L* since it prunes a subtree in each iteration. In this figure, we can also observe that the total run time incurred by *MAHUSP_S* and *MAHUSP_L* increases almost linearly as the stream evolves.

4.7 Parameter sensitivity analysis

In this section we evaluate the performance of *MAHUSP_L* and *MAHUSP_S* by varying the batch size (*batchSize*), the significance threshold (ϵ) and the amount of available memory (*availMem*). In all the experiments, δ is set to 0.46, 0.096, 0.15 and 0.075% for $DS1$, $DS2$, *Kosarak* and *ChainStore* respectively. Figure 13a, d present the results on *DS1*, *DS2*, *Kosarak* and *ChainStore* when the number of sequences in the batch varies. The x-axes in each graph represents the combination of the dataset name and the number of sequences in the batch (i.e., *batchSize*). Figure 13a shows the trend in the execution time with different batch sizes. In all the datasets, the run time decreases as *batchSize* increases since increasing the batch size leads to generating less number of intermediate potential HUSPs. Figure 13d shows F-Scores on different datasets. From Fig. 13d, we can observe that the F-Score of the methods increases slowly with increasing batch sizes. Figure 13b, e show the results on *Run time* and *F-Score* for different values of ϵ. Each bar in the graphs is assigned to each dataset and value of ϵ is a percentage of δ. As it is observed, a higher value of ϵ leads to a lower number of HUSPs returned by *USpan* in each batch and thus the *F-Score* value decreases. On the other hand, when the value of ϵ increases the processing time decreases since the number of HUSPs returned by *USpan* decreases.

Figure 13c, f present the results on different datasets for different values of *availMem*. In the graphs, the x-axes represents the combination of the dataset name and the input parameter *availMem*. Figure 13c shows the execution time with different values of *availMem*. A higher value of *availMem* enables *MAS-Tree* to store more potential HUSPs, hence *LBMA* or *SBMA* is called less frequently to release the memory. Therefore, the execution time decreases when the available memory increases. Figure 13f shows the results on F-Score. When the available memory is small (e.g., 50 MB in DS1), there are fewer HUSPs in the memory and usually F-Score is lower. However, after a certain value of *availMem*, the performance of the proposed methods is much higher.

4.8 Real-life applications

In this section, we demonstrate the effectiveness and efficiency of our method on real-life applications. We first conduct an analysis on a real web clickstream dataset, called *Globe*, obtained from a Canadian news portal to extract patterns in web users' reading behavior. Then, we deploy our method to a publicly available time course microarray dataset, called *GSE6377* (McDunn et al. 2008), to identify gene regulation sequential patterns correlated with a specific disease.

4.8.1 A utility based users' reading behavior mining

News recommendation plays an important role in helping users find interesting pieces of information. A major approach to news recommendation focuses on modeling web users' reading behavior. This approach discovers users' access patterns from web clickstreams using various data mining techniques such as *frequent pattern mining*. Nonetheless, there are some common deficiencies in the frequent pattern based approaches to web users' reading behavior mining. First, they discover users' reading patterns based on the *frequency* of the news being viewed by users, which may not accurately capture users' interests. Second, the news domain is a dynamic environment. When users visit a news website, they are usually looking for important and up-to-the-minute information. However, the frequency-based approaches do not consider the *importance* (e.g., recency) of a news article.

As the first application, we analyze a real-world web clickstream dataset, called *Globe*, obtained from a Canadian news web portal (*The Globe and Mail*[3]). The dataset was created based on a random sample of users visiting *The Globe and Mail* during a 6-month period in 2014. It contains 116,000 sequences and 24,770 news articles. Each sequence in the dataset corresponds to the list of news articles read by a subscribed user in a visit.

News utility model Our goal is to take both news importance and interestingness into account when discovering behavioral patterns related to users' interests. Hence, we define the utility model such that it considers both users' interest in an article and the freshness of an article. Here, we assume that more recent news are more important[4] and the time user spends on a news article reflects his/her interest in the news, that is if the user is not interested in the news, he/she does not spend much time reading it and vice versa.

Given news *nw* and user *usr*, the *internal utility* of *nw* with respect to *usr* is defined as browsing time (in seconds) that *usr* spent on *nw*.[5] In addition, since the importance of *nw* is

[3] http://www.theglobeandmail.com/.

[4] Other importance measures can be used. In this experiment we chose to use recency to measure the importance of a news article.

[5] We consider the time interval between two consecutive visited news articles as time spent of the former article. The last visited news article is removed from the sequence since we cannot calculate its time spent. We also consider the maximum time spent 15 min for news articles whose time spent is more than 15 min.

Table 3 Top-4 HUSPs versus Top-4 FSPs with respect to time spent and support

Algorithm	ID	Pattern (title of the news in the pattern)	Time spent (min)	Support
MAHUSP	$HUSP_1$	Retiree, 60, wonders how long her money will last	**1474**	152
		Which is better,a RRIF or an annuity? You may be surprised		
	$HUSP_2$	Robin Williams warp-speed improvisation was almost…	**1471**	121
		CBC lays off veteran sportscasters amid budget cuts		
	$HUSP_3$	Israel prepares to ‘significantly’ expand campaign as UN chief...	**1212**	116
		MH17: disaster ratchets up Russia-Ukraine tensions		
	$HUSP_4$	Massive explosive decompression’ downed MH17: Kiev	**994**	86
		Canada should learn from Ireland’s housing crash		
PrefixSpan	FSP_1	CBC lays off veteran sportscasters amid budget cuts	576	**286**
		Celine Dion takes indefinite break to focus on health, family		
	FSP_2	La Prairie, Quebec mayor dies from wasp stings	380	**254**
		Duffy billed taxpayers for attending funerals, RCMP allege		
	FSP_3	Supreme Court sides with Ottawa in multibillion-dollar EI case	536	**247**
		MH17: disaster ratchets up Russia-Ukraine tensions		
	FSP_4	Controversial First Nation chiefs salary raises concern	830	**220**
		Harper sticks to hard line on Hamas; U.S. condemns Israel’s deadly...		

The bold values show that our method works better than PrefixSpan in terms of time spent. However, PrefixSpan could find the patterns with highest support

dynamic and varying from time to time, the *external utility* of *nw* is defined as:

$$p(nw) = \frac{1}{accessDate(nw) - releasedDate(nw) + 1},$$

where *accessDate* is the date that *usr* clicks on *nw* and *releasedDate* is the released date of *nw*. Note that 1 in the denominator is added to avoid zero division.

Given news nw and a visit S in the web clickstream dataset D, the utility of nw in S is defined as: $NUM(nw, S) = timeSpent(nw, S) \times p(nw)$. Note that the utility model can be plugged in as desired. The use of more sophisticated the utility model may further improve the quality of the results.

We apply *MAHUSP* to discover HUSPs based on the above utility model. We also applied PrefixSpan algorithm implemented by Fournier-Viger et al. (2013) to discover frequent sequential patterns (i.e., *FSPs*) from the *Globe* dataset. Table 3 presents top-4 HUSPs and top-4 FSPs of length 2, sorted by time spent and support respectively. Table 3 suggests that the pattern with high support is not necessarily a pattern of users’ interest if we use time-spent as the interestingness measure. It is because there exist less frequent patterns (e.g., *HUSP1*, *HUSP2*), which have higher time-spent than highly frequent patterns (e.g., *FSP1*, *FSP2*). Moreover, Fig. 14 shows the sum of browsing time of two groups of patterns returned by *Prefixspan* and *MAHUSP*. In this figure, the x axis refers to the top-k FSPs/HUSPs, while the

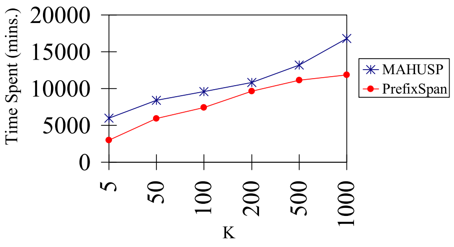


Fig. 14 The sum of the browsing time of top-k HUSPs versus top-k FSPs on the *Globe* dataset

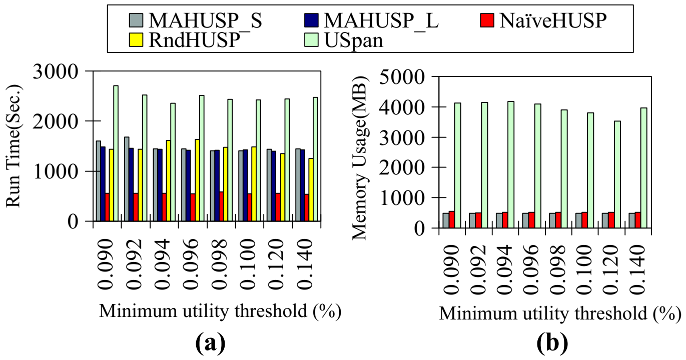


Fig. 15 **a** Run time, **b** memory usage on the *Globe* dataset

y axis shows the sum of the time spent of the top-k patterns. The results show that *MAHUSP* identifies the patterns with higher time spent even though they might not be as frequent as the ones returned by *PrefixSpan*. These patterns can be directly used to produce recommendations to navigate users based on a semantic measure (e.g., news freshness and interestingness) rather than a statistical measure (e.g., support). For example, in Table 3, if a user reads the first article of $HUSP_1$ whose title is *retiree, 60, wonders how long her money will last*, we can recommend the second article in $HUSP_1$ with title *which is better, a RRIF or an annuity? You may be surprised*. These patterns are also useful for the portal designers to understand users' navigation behavior and improve the portal design and e-business strategies.

We also evaluate the performance of *MAHUSP* in comparison to the other methods on this dataset. Figure 15 shows the results in terms of *run time* and *memory usage*. *NaiveHUSP* is the fastest method due to the fact that it only keeps the utility of each item over data streams. However, its utility approximation is inaccurate and causes a high rate of false positives. *USpan* is the slowest, since it re-runs the whole mining process on the current data stream to discover HUSPs. Figure 15b shows the memory usage of the methods. Since *RndHUSP*, *MAHUSP_L* and *MAHUSP_S* consume the same amount of memory, we only present the results of *MAHUSP_S*, *NaiveHUSP* and *USpan*. *NaiveHUSP* uses more memory than *MAHUSP_S* on *Globe* since it needs to keep a huge list of items and their utilities into the memory. Figure 16 shows the *Precision*, *Recall* and *F-Score* values for the four methods with different δ values on the *Globe* dataset. *MAHUSP_S* and *MAHUSP_L* outperform the other methods significantly with an average Precision, Recall and F-score value of 95, 75 and 83% respectively, over the *Globe* dataset.

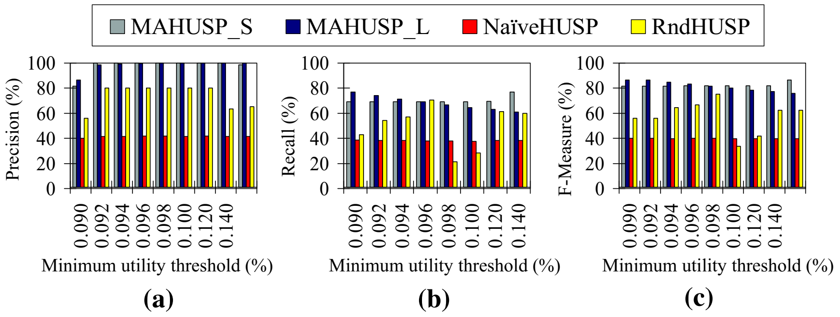


Fig. 16 **a** Precision, **b** Recall and **c** F-Measure performance on the *Globe* dataset

Table 4 (a) An example of a time course microarray dataset, (b) Fold changes of gene/probe values

(a)						(b)					
Patient IDs	Genes	TP_1	TP_2	TP_3	TP_t	Patient IDs	Genes	TP_1	TP_2	TP_3	TP_t
P_1	G_1	240	546	100	50	P_1	G_1	1	2.2	−2.4	−4.8
	G_2	321	98	454	974		G_2	1	−3.2	1.4	3.0
	G_3	410	350	251	243		G_3	1	−1.1	−1.6	−1.6
P_2	G_1	128	786	135	344	P_2	G_1	1	6.1	1.0	2.6
	G_2	253	820	482	90		G_2	1	3.2	1.9	−2.8
	G_3	290	150	256	864		G_3	1	−1.9	−1.1	2.9
P_3	G_1	600	188	99	40	P_3	G_1	1	−3.1	−6.6	−15
	G_2	500	555	510	80		G_2	1	1.1	1.0	−6.2
	G_3	200	400	350	450		G_3	1	2	1.7	2.2

4.8.2 Finding disease-related gene regulation sequential patterns from time course microarray datasets

One of the most important problems arising from bio-applications is to discover *meaningful sequences* of genes in microarray datasets (i.e., DNA or protein sequence databases). Microarray is a powerful tools to provide information on relative levels of expression of thousands of genes among samples. Table 4(a) shows an example of time course microarray dataset consists of three patients whose IDs are P_1, P_2 and P_3. In this table, the gene expression values of three genes G_1, G_2 and G_3 are presented over four time points samples TP_1, TP_2, TP_3 and TP_4.

In recent years, frequency-based sequential pattern mining methods (Bringay et al. 2010; Cheng et al. 2013; Salle et al. 2009) have been extensively conducted on microarray datasets as a promising approach to identify relationships between gene expression levels. In these approaches (Bringay et al. 2010; Cheng et al. 2013; Salle et al. 2009), a gene regulation sequential pattern is chosen as an interesting pattern based on the *frequency/support* framework. However, frequency alone may not be informative enough for biologists in many situations. For example, some genes are more important than others in causing a particular disease and some genes are more effective than others in fighting diseases. The sequences contain these more important/effective genes may not be discovered by the frequency-based approaches, because these approaches neither consider the importance of each gene within biological processes, nor temporal properties of genes under biological treatments.

As the second application, we propose a new approach to identifying disease-related gene regulation sequential patterns by taking the importance of genes with respect to a specific disease and their temporal properties under biological treatments into account. We conduct an analysis on a time course gene expression microarray dataset, called GSE6377 (McDunn et al. 2008), downloaded from the GEMMA database.[6] Our aim is to mine cross-timepoint gene regulation sequential patterns with respect to a specific disease. Below we first define a utility model to discover disease-related gene regulation sequential patterns effectively and then we present how a time-course microarray dataset is converted to a utility-based sequential database. Finally, we apply MAHUSP to find disease-related gene regulation sequential patterns from the dataset.

Gene utility model In each time point, a real value is assigned to each gene that specifies the relative abundance of that gene in the time point. Each expression value in the dataset is usually transformed as *up-regulated* (representing by $+$ meaning that values are greater than a threshold), *down-regulated* (representing by-meaning that values are less than a threshold), or *normal* (neither expressed nor repressed) and only the gene expressions that are up-regulated or down-regulated are preserved (Creighton and Hanash 2003). Hence, each gene (i.e., G_x) in a sample can be thought of as being two *items*, one item referring to the gene being up (i.e., G_{x+}), the other referring to the gene being down (i.e., G_{x-}) (Cheng et al. 2013). The absolute expression value after this process is considered as *internal utility* since it determines expression level of one gene in the sample. On the other hand, as mentioned before, some genes are known by biologists and literature to be more important with respect to a particular disease. The external utility of genes should represent the importance of genes with respect to the disease. Hence, as the *external utility* of each gene, we use the *gene-disease association score* proposed by *DisGeNET*.[7] This score takes the number and type of sources (level of curation, organisms), and the number of publications supporting the association into account to rank genes with respect to a specific disease. The score value is between 0 and 1. Hence, the utility of a gene in a sample is defined as the gene-disease association score multiplied by its expression level in the sample.

Converting microarray dataset into utility-based sequential database in order to mine HUSPs from a microarray dataset, the dataset should be converted into a utility based sequential database. Similar to Cheng et al. (2013), we consider the first time point as a baseline to derive the importance of each gene at each time point. Hence, the values in the table are divided by the first time point values. Table 4(b) shows the divided values as a fold change matrix. Then, we transform genes as up-regulated, down-regulated as follows. If the absolute fold change value is more than a threshold, the gene is defined as an eligible gene. Similar to Chang et al. (2008), the threshold is set as 1.5 and only eligible genes are preserved as new items. Table 5 shows the converted dataset. For example, in patient P_1, up-regulated G_{1+} (2.2) and down-regulated G_{2-} (3.2) occur at the second time point and are considered to occur within the same transaction. In this dataset, each gene is an item and each time point represents as an itemset. The set of time points (i.e., TPs) for each patient forms a sequence. The absolute fold change value represents the internal utility for each gene in each time point. In this study, the importance of G_x represents the importance of both G_{x+} and G_{x-} gene items.

In order to evaluate our proposed gene utility model and also the performance of MAHUSP to find disease-related gene regulation sequential patterns from a time course gene expression dataset, we mine the *GSE6377* dataset. McDunn et al. (2008) attempted to detect 8,793 transcriptional changes in 11 ventilator-associated pneumonia patients leukocytes across 10

6 http://www.chibi.ubc.ca/Gemma/home.html.

7 http://www.disgenet.org/web/DisGeNET/menu.

Table 5 Converted utility-based sequential dataset from time course microarray dataset in Table 4(a)

(a) Patient IDs	Sequence
P_1	$\{G_{1+}(2.2)G_{2-}(3.2)\}_2\{G_{1-}(2.4)G_{3-}(1.6)\}_3\{G_{1-}(4.8)G_{2+}(3.0)G_{3-}(1.6)\}_4$
P_2	$\{G_{1+}(6.1)G_{2+}(3.2)G_{3-}(1.9)\}_2\{G_{1+}(1.0)G_{2+}(1.9)G_{3-}(1.1)\}_3\{G_{1+}(2.6)G_{2-}(2.8)G_{3+}(2.9)\}_4$
P_3	$\{G_{1-}(3.1)G_{2+}(1.1)G_{3+}(2.0)\}_2\{G_{1-}(6.6)G_{2+}(1.0)G_{3+}(1.7)\}_3\{G_{1-}(15)G_{2-}(6.2)G_{3+}(2.2)\}_4$

(b) Gene	G_1	G_2	G_3
Score	0.8	0.6	0.1

Table 6 Top-20 genes related to pneumonia

Rank	Gene name	Rank	Gene name	Rank	Gene name	Rank	Gene name
1	*CAT*	6	*SFTPD*	11	*CYP2J2*	16	*HMGB1*
2	*SFTPA2*	7	*TLR2*	12	*F2*	17	*CR1*
3	*PECAM1*	8	*TLR6*	13	*CXCL3*	18	*FCGR2A*
4	*SFTPB*	9	*PDPN*	14	*CXCL2*	19	*MASP2*
5	*SFTPC*	10	*ITGB3*	15	*MBL2*	20	*IL17A*

time points. Our goal is to decipher *pneumonia-related* gene regulation sequential patterns. We have also downloaded disease-gene association scores from *DisGenet* which contains 40 genes.[8] Table 6 shows top-20 genes most related to *pneumonia*.

We apply *MAHUSP* to extract HUSPs based on the proposed utility model. We also run a frequency-based algorithm, *PrefixSpan* (Pei et al. 2004), to discover frequent gene regulation sequential patterns (i.e.,*FGSs*) from the dataset. Table 7 shows top-4 HUSPs (e.g., disease-related gene regulation sequential patterns) extracted by MAHUSP and top-4 FGSs extracted by *PrefixSpan*, sorted by the utility value and support respectively. Given a gene regulation sequential pattern S_i and disease *dis*, we evaluate the quality of the results using *popularity of a sequence score* (Bringay et al. 2010) which is defined as follows: $Pop(\alpha, dis) = \frac{\sum_{i\in\alpha} w(i,dis)}{|\alpha|}$, where $w(i, dis)$ is the importance of popular gene i for disease dis. We consider the genes presented in Table 6 as popular genes and $w(i, dis) = 20 - rank(i, dis) + 1$. For the genes not presented in the list, $w(i, dis) = 1$. Table 7 suggests that the frequent gene regulation sequential patterns are not necessarily popular w.r.t. the disease even though their support value is more than 90%. This is due to the fact that these patterns are discovered based on their frequency in the dataset which is not informative enough. On the other hand, MAHUSP returns the patterns whose popularity is relatively high. These patterns help biologists select relevant sequences according to a specific disease and also identify the relationships between important genes and the other genes.

Figure 17 shows the performance of different methods on the *GSE6377* dataset in terms of run time and memory usage. The dataset is a dense dataset and as we expected the number

[8] For genes in the dataset which are not in the list, the minimum score in the list is considered as their score value.

Table 7 Top-4 HUSPs versus Top-4 FGSs with respect to utility and support

Algorithm	ID	Sequence of genes (e.g., α)	$Pop(\alpha, Pneumonia)$	$Utility$ (%)
MAHUSP	$HUSP_1$	(F2, SFTPC)(F2, TLR6, SFTPC)	15.75	22
	$HUSP_2$	(TLR6, SFTPC)(TLR6, SFTPC)	14.5	21
	$HUSP_3$	(SFTPC)(TLR6, SFTPC)(SFTPC)	14.5	21
	$HUSP_4$	(SFTPC)(CRP)(SFTPC)	10.5	17
Algorithm	**ID**	**Sequence of genes (e.g., α)**	**$Pop(\alpha, Pneumonia)$**	**$Support$(%)**
PrefixSpan	FSP_1	(DAD1) (RTCB, CAPNS1, ZNF146)	1	91
	FSP_2	(RTCB) (SRSF9, SLC25A3)	1	91
	FSP_3	(DAD1) (CAPNS1, ZNF146, RPS11)	1	91
	FSP_4	(DAD1, RPS11)(CAPNS1)	1	91

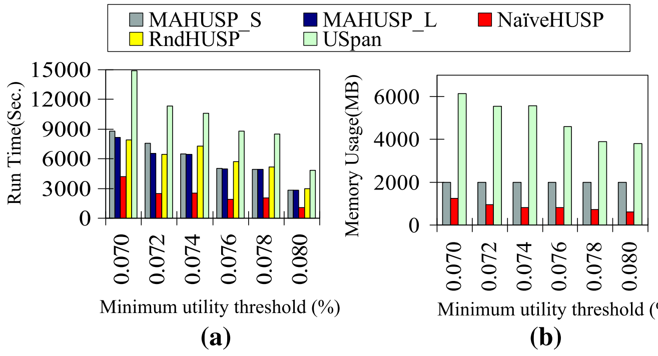


Fig. 17 **a** Run time, **b** memory usage on the *GSE6377* dataset

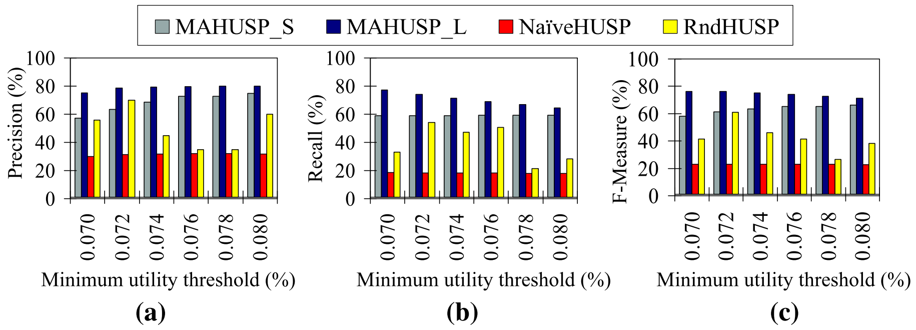


Fig. 18 **a** Precision, **b** Recall and **c** F-Measure performance on the *GSE6377* dataset

of HUSPs is huge. For example, given threshold 0.07, there are 56,542,360 HUSPs in the dataset. In this experiment *availMem* is set to 2 GB. The proposed methods are more efficient than *USpan*. *NaiveHUSP* is the fastest method since it works based on item utilities in the dataset to find HUSPs. But, its false positive rate is high due to its inaccurate approximate utility. Figure 18 shows *Precision*, *Recall* and *F-Score* values for the four methods. In general, both proposed methods outperform the other methods with an average Precision, Recall and F-score values of 75, 67 and 71% over the *GSE6377* dataset.

4.9 Discussion

In reality, datasets can be sparse and/or dense in different domains. Considering the rationale behind the proposed memory-adaptive mechanisms and the experimental results, the question of what are the situations in which one mechanism is more effective and efficient than the other one is addressed as follows.

Generally speaking, if data sharing is common such as in dense databases, the potential HUSPs found in batches will have more overlaps in the tree. In this case, if a subtree is mistakenly pruned due to memory violation (e.g., a subtree whose root has minimum *rsu* and it does not present a HUSP but its subtrees contain HUSPs), several high utility sequential patterns will be missed concurrently. Therefore, *SBMA* strategy may downgrade the performance on dense datasets. In such cases, pruning leaf nodes might be more effective (e.g., LBMA strategy), since only one pattern, represented by the path from root to the leaf node with minimum utility, is pruned. On the other hand, if data sharing is rare such as in sparse databases, potential HUSPs are generally stored as different branches and share less number of nodes in the tree. In this case, pruning sub-tree to release memory might be more effective than pruning leaf nodes. Our experiments assert such claims. For example, in the sparse dataset *Kosarak*, *MAHUSP_S* outperforms *MAHUSP_L* in terms of running time and F-Score. On the other hand, in the dense datasets, we noticed that *MAHUSP_S* did not perform as well as *MAHUSP_L*.

Moreover, there is no precise relationship between ϵ and the type of dataset. Generally, a lower value of ϵ creates a larger tree and thus the memory adaptive mechanisms may be triggered more often to release the memory. Hence, the algorithm takes longer to find HUSPs. In this case, if the selected memory adaptive mechanism (LBMA or SBMA) releases the memory effectively (e.g., all (or most of) high utility sequential patterns remain in the tree), then a lower value of ϵ increases the effectiveness of the methods. Otherwise, not only it takes longer to find the patterns, but also several true HUSPs will be missed. On the other hand, a high value of ϵ usually downgrades the effectiveness of the methods, since USpan returns less number of patterns in the batch as potential HUSPs. According to the experiments, the effectiveness of the algorithms on the sparse dataset *Kosarak* is more sensitive than dense datasets (e.g., DS2) to the value of ϵ. This is due to the fact that, in a sparse dataset, the number of items compared to the number of sequences in the dataset is relatively high and thus a HUSP occurs less often in different batches. Therefore, if ϵ is set to a high value such that a HUSP is not returned as a potential HUSP in a batch, it is less likely the same pattern appears in subsequent batches again. In contrast, in dense datasets, a HUSP may appear more often in different batches. In this case, the chance of finding the pattern as a potential HUSP in several batches is higher, thus the results are less sensitive to the ϵ value.

5 Related work

5.1 Sequential pattern mining over data streams

Sequential pattern mining plays an important role in data mining and several related algorithms have been proposed such as AprioriAll (Agrawal and Srikant 1995), GSP (Srikant and Agrawal 1996), FreeSpan (Han et al. 2010), PrefixSpan (Marascu and Masseglia 2005), SPADE (Zaki 2001) and SPAM (Ayres et al. 2002). These algorithms can be generally categorized as using a horizontal database (e.g., AprioriAll, GSP, FreeSpan and PrefixSpan) or a vertical database (e.g., SPADE, SPAM and ClaSP). The AprioriAll and GSP algorithms use candidate-generation-and-test methodology for mining sequential patterns. FreeSpan and PrefixSpan discover sequential patterns by the pattern-growth methodology. The SPADE and SPAM algorithms use different vertical representations for mining sequential patterns. A vertical representation provides the advantage of calculating frequencies of patterns without performing costly database scans. This allows vertical mining algorithms to perform better on datasets having dense or long sequences than algorithms using the horizontal format.

Data stream mining is one of the most important and challenging topics in data mining. The desired feature of stream mining algorithms is that when records are inserted into or deleted from the database, the algorithms can incrementally update the patterns based on previous mining results, which is much more efficient than re-running the whole mining process on the updated database. Several studies such as Chen et al. (2005), Marascu and Masseglia (2005), Ho et al. (2006), Raissi et al. (2006) and Chang et al. (2008) have been performed for mining sequential patterns over data streams. MILE (Chen et al. 2005) is an efficient algorithm to mine sequential patterns over data streams. The proposed algorithm recursively uses the knowledge of existing patterns to mine the new patterns efficiently. One unique feature of *MILE* is when some prior knowledge of the data distribution in the data streams is available, it can be incorporated into the mining process to further improve the performance of MILE. According to the experiments, MILE consumes more memory than *PrefixSpan*, so the authors designed a solution to balance the memory usage and time efficiency in memory limited environments. SMDS (Marascu and Masseglia 2005) extracts sequential patterns over data streams and it has two main features: First, using a sequence alignment method, *SMDS* summarizes batches of transactions. This process works based on a greedy clustering algorithm. The algorithm provides distinct clusters of sequences by considering the main features of web usage sequences. Second, frequent sequences found by *SMDS* are stored in a prefix tree structure. This data structure enables *SMDS* to calculate the real support of each proposed sequence. *IncSpam* (Ho et al. 2006) is single-pass algorithm to mine the set of sequential patterns over a stream of itemset-sequences with a sliding window. The authors proposed a bit-sequence representation to reduce the memory requirement of the online maintenance of sequential patterns generated so far. The model receives the transactions from the data stream and uses a bit-vector data structure, called *Customer Bit-Vector Array with Sliding Window* (CBASW), to store the information of items for each sequence. Lastly, a weight function is adopted in this sliding window model. The weight function can judge the importance of a sequence and ensure the correctness of the sliding window model. SPEED (Raissi et al. 2006) identifies frequent maximal sequential patterns in a data stream. SPEED is based on a new structure and on strict valuation of edges. Such a valuation is very useful either when considering the pruning phase or when comparing sequences since we only have to consider sequences embedded into the lattice sharing same valuations. At any time, users can issue requests for frequent maximal sequences over an arbitrary time interval. Furthermore, SPEED produces an approximate support answer with an assurance that it will not bypass a user-defined frequency error threshold. SeqStream (Chang et al. 2008) mines closed sequential patterns in stream windows incrementally. It works based on various novel strategies to prune search space aggressively. SeqStream works based on a synopsis structure called Inverse closed Sequence Tree (*IST*) to keep inverse closed sequential patterns in current window. There are two main tasks in SeqStream. The first one is to efficiently update supports of tree nodes under when an element is inserted or removed. The second one is to remove nodes that do not need to update or extend to eliminate the related support counting and node extension costs as much as possible. *SeqStream* uses an insertion database and removal database to update the support information of current IST. These two databases are usually much smaller than the whole database in the sliding window. Therefore the support update task can be completed quickly. For the second task, SeqStream adopts various pruning strategies to safely skip nodes that do not need to be handled.

On the other hand, several studies (Lin et al. 2013; Manku and Motwani 2002; Mendes et al. 2008) have been conducted to use approximate approaches to discover frequent patterns over the entire data stream. One of the first methods to find frequent pattern over the entire data stream is proposed by Manku and Motwani (2002). In Manku and Motwani (2002), authors presented algorithms for computing frequency counts of items exceeding a user-specified threshold over data streams. They also proposed an extended version of the algorithm to find frequent itemsets over data streams. Metwaly et al. (2005) proposed an algorithm which returns a list of the most frequent items from a skewed data stream. Mendes et al. (2008) proposed two methods (i.e., SS-BE and SS-MB) inspired by Manku and Motwani (2002) for finding frequent sequential patterns over data streams. However, all these methods are for finding frequent patterns and they do not have memory adaptive mechanisms. Although the above algorithms are pioneers in sequential pattern mining, they treat all items as having the same importance/utility and assumes that an item appears at most once at a time point, which do not reflect the characteristics in the scenario of several real-life applications and thus the useful information of sequences with high utilities such as high profits is lost. To address the issue, high utility sequential pattern (HUSP) mining (Ahmed et al. 2011, 2010; Shie et al. 2013; Yin et al. 2012) has emerged as a novel research topic in data mining recently.

5.2 High utility sequential pattern mining

UMSP (Shie et al. 2013) was designed for mining high utility mobile sequential patterns. Each itemset in a sequence is associated with a location identifier. With this feature, the utility of a mobile sequential pattern is also a single value. The authors integrate mobile sequential pattern mining with utility mining for finding high-utility mobile sequential patterns in this paper. Two different types of methods, namely level-wise and tree-based ones, are proposed for this problem. For level-wise method, an algorithm called $UMSP_L$ (mining high Utility Mobile Sequential Patterns by a Level-wise method) is proposed. Not only supports but also utilities of patterns are considered in the level-wise mining processes. For tree-based methods, two algorithms $UMSP_{T(DFG)}$ (mining high Utility Mobile Sequential Patterns by a Tree-based method with a Depth First Generation strategy) and $UMSP_{T(BFG)}$ (mining high Utility Mobile Sequential Patterns by a Tree-based method with a Breadth First Generation strategy) are proposed. Both of the two tree-based algorithms use a tree structure named *MTS-Tree* (Mobile Transaction Sequence Tree) to summarize the corresponding information, such as locations, items, paths and utilities, in mobile transaction databases. *UMSP* searches for patterns within *MTS-Tree*, which is efficient. However, due to the specific constraint on the sequences, this algorithm can only handle specific sequences with simple structures (single item in each sequence element, and a single utility per item). In Ahmed et al. (2011), an algorithm is specifically designed for utility web log sequences. The utility of a pattern can have multiple values, and the authors choose the utility with maximal values to represent a patterns utility with two tree structures, i.e. *UWAS-tree* and *IUWAS-tree*. The proposed approach can handle both forward and backward references, static and dynamic data, avoids the level-wise candidate generation-and-test methodology, does not scan databases several times and considers both internal and external utilities of a web page. However, sequence elements with multiple items such as $\langle(c, 2)(b, 1)\rangle$ can not be supported, and the scenarios considered are rather simple, which limit the algorithm's applicability for complex sequences. *UI* and *US* (Ahmed et al. 2010) extend traditional sequential pattern mining. For mining high-utility sequential patterns, they propose two new algorithms. (1) *Utility Level* which is a high-utility sequential pattern mining with a level-wise candidate generation approach, and (2) *Utility Span* which is a high-utility sequential pattern mining with a pattern growth approach. A pattern utility is calculated in two ways. The utilities of sequences having only distinct occurrences are added together, while the highest occurrences are selected from sequences with multiple occurrences and used to calculate the utilities. However, the problem definition in Ahmed et al. (2010) is rather specific. No generic framework is proposed which has a clear process to transfer from sequential pattern mining to high utility sequence analysis. Recently, Yin et al. (2012) proposed a new definition for high utility sequential patterns mining, which aims at finding sequences having a maximum utility. Then, they proposed the *USpan* algorithm for mining such patterns. *USpan* is the current best algorithm for mining high utility sequential patterns. *USpan* is a general framework for high utility sequential pattern mining. To satisfy *Downward Closure Property*, the authors have defined and used *Sequence Weighted Utilization* (*SWU*) and *Sequence Weighted Downward Closure (SDCP)*. Based on the *SWU* of a pattern, an item is called promising if adding it to a candidate pattern results in a new pattern whose *SWU* is greater than or equal to the minimum utility threshold. Therefore, the pruning they perform before candidate generation is based on this *SDCP*. Their proposed depth pruning is a *Pruning After Candidate Generation (PACG)* mechanism. However, all of the HUSP mining methods were designed for static datasets, not for data streams.

5.3 High utility pattern mining over data streams

Several methods have been proposed to mine *high utility itemsets* (*HUIs*) from data streams (Ahmed et al. 2012; Tseng et al. 2006; Li et al. 2008). *THUI-Mine* (Tseng et al. 2006) was the first algorithm for mining temporal high utility itemsets from data streams. The underlying idea of *THUI-Mine* algorithm is to integrate the advantages of Two-Phase algorithm (Liu et al. 2005) with the incremental mining techniques for mining temporal high utility itemsets. Later, two algorithms, called *MHUI-BIT* and *MHUI-TID*, were proposed in Li et al. (2008) for mining high utility itemsets from data streams. Two effective representations of item information, i.e., *Bitvector* and *TIDlist*, were developed and used in the proposed methods to restrict the number of candidates and to reduce the processing time and memory usage. The author proposed two algorithms to mine the set of high utility itemsets based on *Bitvector* and *TIDlist*, respectively. However, the proposed representations become very inefficient when the number of distinct items become large in a window. *GUIDE* is a framework proposed in Shie et al. (2012) for mining a compact form of high utility itemsets from data streams with different models (i.e. the landmark, sliding and time fading window models). It works based on a tree structure, called *MUI-Tree*, which is constructed in one scan of the data stream. *HUPMS* (Ahmed et al. 2012) is a recent method for mining HUIs from data streams, which is based on the *TWU* model. The authors proposed a novel tree structure, called *HUS-tree* (Incremental and Interactive Utility Tree for Stream data) to keep information about patterns and their *TWU* values. *HUS-tree* has the *build once mine many* property for interactive mining. In Ryang and Yun (2016), the authors proposed an algorithm, called *SHU-Growth*, for mining high utility itemsets from a data stream. They first developed two techniques called *RGE* and *RLE*, to decrease over-estimated utilities. They also proposed a novel tree structure, called *SHU-Tree*. They designed methods to construct and update the tree in order to utilize reduced over-estimated utilities. Since, the proposed method applies the sliding window model to mine HUIs, it stores the utility information of patterns for each batch of transactions separately. Thus, it is not memory efficient to discover patterns over a long period of time. Moreover, the proposed method is not a memory adaptive mining approach. In Kim and Yun (2016), the authors proposed a tree-based algorithm for mining recent high utility itemsets over data streams. The proposed algorithm applies the time decaying model and diminishes the utilities of transactions according to their arrival-time in order to assign larger weights to recent data. The algorithm regularly updates the utility information in the tree and removes the nodes whose utility value is less than the threshold. Although the aforementioned methods are efficient to discover high utility itemsets over data streams, they are not able to discover HUSPs since they do not consider the ordering relationships between items or itemsets. Considering the sequential orders between itemsets makes the mining process much more challenging than mining high utility itemsets.

In Zihayat et al. (2015), we proposed a method called *HUSP-Stream* to discover HUSPs over a time-sensitive sliding window. *HUSP-Stream* works based on a data structure named *HUSP-Tree* which is to store the essential information for mining HUSPs over a time-sensitive sliding window. *HUSP-Tree* can be easily updated when new data arrive and old data expire in the window. When data arrive at or leave from a sliding window, *HUSP-Stream* incrementally updates *HUSP-Tree* to find HUSPs based on previous mining results. During the mining HUSPs in the current window, *HUSP-Stream* applies a new utility estimation model to more effectively prune the search space. The proposed method applies the sliding window model and is not able to find HUSPs over the entire data stream. *HUSP-Tree* (Zihayat et al. 2015) is not applicable to store the information over a long period of time. Because the main idea behind *HUSP-Tree* is to facilitate update procedure when a new transaction arrives at or leaves from a the window. Hence, the nodes in the tree is designed such that it stores the utility information per transaction. This data structure is not efficient for mining HUSPs over the entire data stream, since it consumes more memory and it is not as compact as the proposed tree structure in this paper. Moreover, HUSP-Stream applies an upper bound, called *Suffix Utility* (i.e., SFU), to prune patterns during HUSP discovery in the current window. However, SFU is different than the rest utility model proposed in this paper. *SFU* is an upper bound of the utilities of some of its super-sequences (i.e., S-Concatenate super-sequences) while rest utility is an upper bound of the utilities of all of the super-sequences. During memory adaptive mechanism, we need an upper bound regardless of the type of super-sequences, hence SFU is not applicable in our study.

6 Conclusions

We tackled the problem of memory adaptive high utility sequential pattern mining over data streams. We proposed an approximation algorithm, called *MAHUSP*, to discover HUSPs over an incrementally larger data stream. *MAHUSP* is based on a compressed tree structure and two memory adaptive mechanisms that can adapt the memory usage to the available memory by pruning the least promising part of the tree when necessary. We proved that *MAHUSP* returns all the true HUSPs under certain circumstances. Our extensive evaluation of *MAHUSP* on both real and synthetic datasets showed that *MAHUSP* effectively adjusts the memory usage over the course of HUSP mining with very little overhead, and it returns more accurate results than other methods in comparison. In order to further demonstrate the effectiveness and efficiency of our method on real-life applications, we conducted an analysis on a real web clickstream dataset, obtained from a Canadian news portal, to extract patterns in web users' reading behavior. Then, we also applied *MAHUSP* to a publicly available time course microarray dataset to identify gene regulation sequential patterns correlated with a specific disease. The evaluation results showed that our method can effectively discover meaningful patterns, which are useful in each of the applications. The results revealed the practical value of this work in real-life applications in terms of discovering high utility sequential patterns over data streams. In this study, we confine the resource to memory. As future work, we will extend our method to consider new types of resources such as CPU. As another future direction, we plan to discover top-k HUSPs over data streams to limit the search space and thus the resource consumption.

References

Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. In *ICDE* (pp. 3–14).

Ahmed, C. F., Tanbeer, S. K., & Jeong, B. (2010). A novel approach for mining high-utility sequential patterns in sequence databases. *ETRI Journal*, *32*, 676–686.

Ahmed, C. F., Tanbeer, S. K., & Jeong, B. (2011). A framework for mining high utility web access sequences. *IETE Journal*, *28*, 3–16.

Ahmed, C. F., Tanbeer, S. K., & Jeong, B. S. (2012). Interactive mining of high utility patterns over data streams. *Expert Systems with Applications*, *39*, 11979–11991.

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

Bringay, S., Bringay, S., Roche, M., Teisseire, M., Poncelet, P., Rassoul, R. A., et al. (2010). Discovering novelty in sequential patterns: Application for analysis of microarray data on alzheimer disease. *Studies in Health Technology and Informatics*, *14*(160), 1314–1318.

Chang, L., Wang, T., Yang, D., & Luan, H. (2008) Seqstream: Mining closed sequential patterns over stream sliding windows. In *Proceedings of the IEEE international conference on data mining* (pp. 83–92).

Chen, G., Wu, X., & Zhu, X. (2005) *Mining sequential patterns across data streams*, Ph.D. thesis. University of Vermont.

Cheng, C. P., Liu, Y. C., Tsai, Y. L., & Tseng, V. S. (2013). An efficient method for mining cross-timepoint gene regulation sequential patterns from time course gene expression datasets. *BMC Bioinformatics*, *14*(12), 1–12.

Creighton, C., & Hanash, S. (2003). Mining gene expression databases for association rules. *Bioinformatics*, *19*(1), 79–86.

Fournier-Viger, P., Gomariz, A., Soltani, A., & Gueniche, T. (2013) *Spmf: Open-source data mining library*. http://www.philippe-fournier-viger.com/spmf/.

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., & Hsu, M. (2010) Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 355–359).

Ho, C., Li, H., Kuo, F., & Lee, S. (2006). Incremental mining of sequential patterns over a stream sliding window. In *Proceedings of the ICDM workshops* (pp. 677–681).

Kim, D., & Yun, U. (2016). Mining high utility itemsets based on the time decaying model. *Intelligent Data Analysis*, *20*(5), 1157–1180.

Li, H. F., Huang, H. Y., Chen, Y. C., Liu, Y. J., & Lee, S. Y. (2008). Fast and memory efficient mining of high utility itemsets in data streams. In *Proceedings of the 8th IEEE international conference on data mining* (pp. 881–886).

Lin, W. Y., Yang, S. F., & Hong, T. P. (2013). Memory-aware mining of indirect associations over data streams. In *IDAM 2013*. Amsterdam: Springer.

Liu, Y., Liao, W., & Choudhary, A. (2005). A fast high utility itemsets mining algorithm. In *Proceedings of the 1st international workshop on utility-based data mining* (pp. 90–99).

Manku, G. S., & Motwani, R. (2002). Approximate frequency counts over data streams. In *Proceedings of VLDB*, (pp. 346–357).

Marascu, A., & Masseglia, F. (2005). Mining sequential patterns from temporal streaming data. In *Proceedings of the ECML/PKDD workshop on mining complex data* (pp. 355–359).

McDunn, J., Husain, K., Polpitiya, A., Burykin, A., Ruan, J., Li, Q., et al. (2008). Plasticity of the systemic inflammatory response to acute infection during critical illness: Development of the riboleukogram. *PloS ONE*, *3*(2), e1564.

Mendes, L., Ding, B., & Han, J. (2008). Stream sequential pattern mining with precise error bounds. In *ICDM '08* (pp. 941–946).

Metwaly, A., Agrawal, D., & Abadi, A. (2005). Efficient computation of frequent and top-k elements in data streams. In *ICDT* (pp. 398–412). Berlin: Springer.

Mooney, C. H., & Roddick, J. F. (2013). Sequential pattern mining approaches and algorithms. *ACM Computing Surveys*, *45*(2), 19:1–19:39.

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

Pisharath, J., Liu, Y., Ozisikyilmaz, B., Narayanan, R., Liao, W. K., Choudhary, A., & Memik, G. (2012). *Nu-minebench version 2.0 dataset and technical report*. http://cucis.ece.northwestern.edu/projects/dms/minebench.html.

Raissi, C., Poncelet, P., & Teisseire, M (2006). Speed: Mining maximal sequential patterns over data streams. In *Proceedings of the IEEE international conference on intelligent systems* (pp. 546–552).

Ryang, H., & Yun, U. (2016). High utility pattern mining over data streams with sliding window technique. *Expert Systems with Applications*, *57*, 214–231.

Salle, P., Bringay, S., & Teisseire, M. (2009). Mining discriminant sequential patterns for aging brain. In *Artificial intelligence in medicine: 12th conference on artificial intelligence in medicine, AIME 2009, Verona, Italy, Proceedings* (pp. 365–369). Berlin: Springer.

Shie, B. E., Yu, P. S., & Tseng, V. S. (2012). Efficient algorithms for mining maximal high utility itemsets from data streams with different models. *Expert Systems with Applications*, *39*, 12947–12960.

Shie, B. E., Hsiao, H. F., & Tseng, V. S. (2013). Efficient algorithms for discovering high utility user behavior patterns in mobile commerce environments. *Knowledge and Information systems*, *37*(2), 363–387.

Srikant, R., & Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the international conference on extending database technology: Advances in database technology* (pp. 3–17).

Tseng, V. S., Chu, C. J., & Liang, T. (2006). Efficient mining of temporal high-utility itemsets from data streams. In *ACM KDD utility based data mining* (pp. 18–27).

Wang, J. Z., Yang, Z. H., & Huang, J. L. (2014). An efficient algorithm for high utility sequential pattern mining. In *Frontier and innovation in future computing and communications* (Vol. 301, pp. 49–56). Amsterdam: Springer.

Yin, J., Zheng, Z., & Cao, L. (2012). Uspan: An efficient algorithm for mining high utility sequential patterns. In *Proceedings of ACM SIGKDD* (pp. 660–668).

Yin, J., Zheng, Z., Cao, L., Song, Y., & Wei, W. (2013). Efficiently mining top-k high utility sequential patterns. In *IEEE 13th international conference on data mining (ICDM)* (pp. 1259–1264).

Zaki, M. J. (2001). Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, *42*, 31–60.

Zihayat, M., Wu, C. W., An, A., & Tseng, V. S. (2015). Mining high utility sequential patterns from evolving data streams. In *Proceedings of the ASE BigData & SocialInformatics 2015, ASE BD&SI '15* (pp. 52:1–52:6). New York, NY: ACM. doi:10.1145/2818869.2818883.