Memory-Bounded High Utility Sequential Pattern Mining over Data Streams

Morteza Zihayat, Yan Chen and Aijun An

Technical Report EECS-2015-04

October 10 2015

Department of Electrical Engineering and Computer Science
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

Memory-Bounded High Utility Sequential Pattern Mining over Data Streams

Morteza Zihayat, Yan Chen and Aijun An *

Abstract

Mining high utility sequential patterns (HUSPs) has emerged as an important topic in data mining. However, the existing studies on this topic focus on static data and do not consider streaming data. Streaming data are fast changing, continuously generated and unbounded in amount. Such data can easily exhaust computer resources (e.g., memory) unless proper resource-aware mining is performed. In this study, we explore a fundamental problem that is how the limited memory space can be well 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, to efficiently discover HUSPs over data streams. *MAHUSP* guarantees that all HUSPs are discovered under certain circumstances. Our experimental study shows that our algorithm cannot only discover HUSPs over data streams efficiently, but also adapt to memory allocation without sacrificing much the quality of discovered HUSPs. Furthermore, in order to show the effectiveness and efficiency of *MAHUSP* in real-life applications, we conduct an analysis on a web clickstream dataset obtained from a Canadian news portal. The results show that *MAHUSP* effectively discovers useful patterns that showcases users' reading behavior.

1 Introduction

Sequential pattern mining is an important task in data mining and has been extensively studied by many researchers [7]. 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 area decisions such as how to maximize revenue or minimize marketing or inventory costs. Recently, high utility sequential pattern mining has been studied to address this limitation [2, 9, 10]. In high utility sequential pattern mining, each item has a *global weight* (e.g. unit price/profit) and a *local weight* in a transaction (e.g. purchase quantity).

Although some preliminary works have been conducted on this topic, existing studies [2, 9] do not consider the real-world applications, such as web clickstream analysis and online analysis of user behavior, that involve *data streams*. A data stream is a continuous and unbounded flow of data and mining algorithms need to process the data in real time with one scan of data. In general, there are three main types of stream-processing windows: *damped window*, *sliding window* and *landmark window*. The first two windows place more importance on recent data than old ones. 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 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 [5]. For example, we may want to find important event sequences in an energy network since a new set of equipment was installed to monitor the quality of the equipment over its life-time; we may want to monitor the sequence of side-effects of a vaccine since it started to be used; or we may want to detect important buying sequences of customers since the beginning of a year or since the store changed its layout. Monitoring only recent data (e.g, in sliding window) may miss some sequences that are important for decision making over a long term. A complete re-scan of a long portion of a data stream is usually impossible or prohibitively costly. The *landmark window* is used for such a purpose, which consists of all the data from a past time point (called *landmark*) until the current time. In this paper, we aim at finding HUSPs over landmark windows, which has not been tackled before.

Compared with other data stream mining tasks, there are unique challenges in discovering HUSPs over landmark windows. First, HUSP mining needs to search a large search space due to a combinatorial number of possible sequences. Second, HUSP mining does not have *downward closure property* to prune low utility patterns efficiently. That is, the utility of a sequence may be higher than, equal to or lower than those of its

*Department of Computer Science and Engineering, York University, Toronto, Canada. {zihayatm, ychen, aan}@cse.yorku.ca

super-sequences and sub-sequences [9]. 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 so 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.

In this paper, we tackle these challenges and propose a memory-adaptive approach to discover 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 cannot only identify recent HUSPs but also high utility patterns over a long period of time. 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*. 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.

2 Definitions and Problem Statement

Let $I^* = \{I_1, I_2, \cdots, I_N\}$ be a set of items. An itemset-sequence S (or sequence in short) is an ordered list of itemsets $\langle X_1, X_2, \cdots, X_Z\rangle$, where $X_i \subseteq I^*$ and Z is the *size* of S. 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. A **batch** $B_k = \{S_i, S_{i+1}, ..., S_{i+L-1}\}$ is a set of L sequences that occur during a period of time t_k. The number of sequences can differ among batches. A **sequence data stream** $DS = \langle B_1, B_2, \cdots, B_k, \cdots\rangle$ is an ordered and unbounded list of batches where $B_i \bigcap B_j = \emptyset$ and $i \neq j$.

	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

Figure 1: An example of a data stream of itemset-squences

Figure 1 shows a sequence data stream with 2 batches $B_1 = \{S_1, S_2\}$ and $B_2 = \{S_3, S_4, S_5\}$.

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 $\alpha = \langle X_1, X_2, ..., X_i\rangle$ is a* sub-sequence *of $\beta = \langle X_1^{'}, X_2^{'}, ..., X_j^{'}\rangle$ ($i \leq j$) or equivalently β is a* super-sequence *of α if there exist integers $1 \leq e_1 < e_2 < ...e_i \leq j$ such that $X_1 \subseteq X_{e_1}^{'}, X_2 \subseteq X_{e_2}^{'}, ..., X_i \subseteq X_{e_i}^{'}$ (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 $u(I, S_r^d) = p(I) \cdot q(I, S_r^d)$.*

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 Figure 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 $S_r = \langle S_r^1, S_r^2, ..., S_r^n\rangle$ and a sequence $\alpha = \langle X_1, X_2, ..., X_Z\rangle$ where S_r^i and X_i are itemsets, α occurs in S_r iff there exist integers $1 \leq e_1 < e_2 < ... < e_Z \leq n$ such that $X_1 \subseteq S_r^{e_1}, X_2 \subseteq S_r^{e_2}, ..., X_Z \subseteq S_r^{e_Z}$. The ordered list of itemsets $\langle S_r^{e_1}, S_r^{e_2}, ..., S_r^{e_Z}\rangle$ 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 $\tilde{o} = \langle S_r^{e_1}, S_r^{e_2}, ..., S_r^{e_Z}\rangle$ be an occurrence of $\alpha = \langle X_1, X_2, ..., X_Z\rangle$ 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 $su(\alpha, S_r) = \max\{su(\alpha, \tilde{o}) | \forall \tilde{o} \in OccSet(\alpha, S_r)\}$.*

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

For example, in Figure 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 data set D)*** *The utility of a sequence α in a data set 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 $U_{B_k} = \sum_{S_r \in B_k} su(S_r)$. The **total utility of a data stream** $DS_i = \langle B_1, B_2, \cdots, B_i\rangle$ is defined as $U_{DS_i} = \sum_{B_k \in DS_i} U_{B_k}$.

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 no 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 $DS = \langle B_1, B_2, \cdots, B_i, \cdots\rangle$ (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 $DS_i = \langle B_1, B_2, \cdots, B_i\rangle$ under the following constraints: (1) the memory usage does not exceed $availMem$, and (2) only one pass of data is allowed in total.

3 Memory Adaptive High Utility Sequential Pattern Mining

In this section, we propose a single-pass algorithm named *MAHUSP (Memory Adaptive High Utility Sequential Pattern mining over data streams)* for incrementally mining an approximate set of HUSPs over a data stream. Algorithm 1 represents an overview of *MAHUSP*. Given a utility threshold δ and a significance

Algorithm 1 *MAHUSP*

Input: B_k, δ, ϵ, $availMem$, $mechanismType$
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

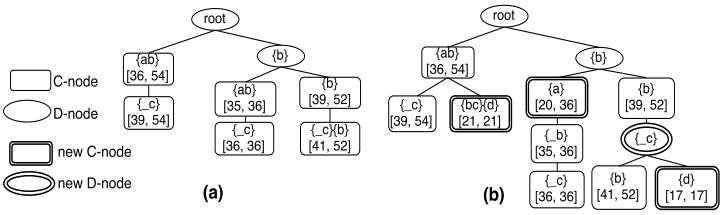

Figure 2: (a) An example of *MAS-Tree* for B_1 in Figure 1. Note that an underscore "_" in a node name $\{\_c\}$ means that the last itemset in the pattern of its parent, such as $\{ab\}$, belongs to the first itemset of the pattern of this node. (b) *MAS-Tree* after inserting three patterns:$\langle\{ab\}\{bc\}\{d\}\rangle$,$\langle\{b\}\{a\}\rangle$ and $\langle\{b\}\{bc\}\{d\}\rangle$.

threshold ϵ[1], as a new batch B_k forms, *MAHUSP* first applies an existing HUSP mining algorithm on static data (e.g., USpan [9]) [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. *MAHUSP* then calls Algorithm 2 to insert these potential HUSPs into the *MAS-Tree* structure. Algorithm 2 assures that the memory constraint is satisfied and the most potential HUSPs are kept in the tree. 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 $DS_k = \langle B_1, B_2, \cdots, B_k\rangle$. In Section 3.5, we will explain why we use $(\delta - \epsilon) \cdot U_{DS}$ as the utility threshold.

3.1 MAS-Tree Structure

We propose a novel data structure MAS-Tree (Memory Adaptive high utility Sequential 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

[1] ϵ is lower than the utility threshold δ and specifies a tradeoff between accuracy and run time.

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

to present *MAS-Tree*, the following definitions are provided.

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

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

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

DEFINITION 12. ***(Suffix of a sequence)*** *Given a sequence* $\alpha = \langle X_1, X_2, ..., X_i\rangle$ *as a* $prefixSUB$ *of* $\beta = \langle X_1^{'}, X_2^{'}, ..., X_j^{'}\rangle$ *$(i \leq j)$, sequence* $\gamma = \langle X'_i - X_i, X'_{i+1}, ..., X'_j\rangle$ *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 Figure 2(a) 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 Figure 2(a), 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[3].

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 Figure 2(a), 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.

3.2 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* $S_r = \langle S_r^1, S_r^2, ..., S_r^n\rangle$ *and a sequence* $\alpha = \langle X_1, X_2, ..., X_Z\rangle$, $\tilde{o} \in OccSet(\alpha, S_r)$ *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* $S_r = \langle S_r^1, S_r^2, ..., S_r^n\rangle$ *and* $\alpha = \langle X_1, X_2, ..., X_Z\rangle$*, where* $\alpha \preceq S_r$*. The rest sequence of* S_r *w.r.t.* α*, is defined as:* $restSeq(S_r, \alpha) = \langle S_r^m, S_r^{m+1}, ..., S_r^n\rangle$*, 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 Figure 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 $rsu(\alpha, S_1) = su(\alpha, S_1) + 15 = \max\{7, 9\} + 15 = 24$.

[3]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 is frequent and is represented by a tree node. In a MAS-Tree we do not store all subsequences of potential HUSPs since a subsequence of a HUSP may not be a HUSP.

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

THEOREM 3.1. *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 $\beta = \langle X_1, X_2, \ldots, X_M^{'}, X_{M+1}, ..., X_N \rangle$, where $X_M \lesssim X_M^{'}$. According to Definition 6:

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

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

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

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

Also, $\forall \tilde{o}_\alpha \in OccSet(\alpha, S_r), su(\alpha, \tilde{o}_\alpha) \leq su(\alpha, S_r)$ (3)
Similarly, $\forall \tilde{o}_{\beta'} \in OccSet(\beta^{'}, S_r - \tilde{o}_\alpha), su(\beta^{'}, \tilde{o}_{\beta'}) \leq$

$$su(\beta^{'}, S_r - \tilde{o}_\alpha) \quad (4)$$

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:

$$su(\beta^{'}, \tilde{o}_{\beta'}) \leq su(\beta^{'}, S_r - \tilde{o}_\alpha) \leq su(\beta^{'}, restSeq(S_r, \alpha))$$
$$\leq su(restSeq(S_r, \alpha)) \quad (5)$$

From (3) and (5):
$su(\beta, S_r) = su(\alpha, \tilde{o}_\alpha) + su(\beta^{'}, \tilde{o}_{\beta'}) \leq su(\alpha, S_r) + su(restSeq(S_r, \alpha)) = rsu(\alpha, S_r)$.

3.3 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 Figure 2(a), 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 Figure 2(b)).
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 Figure 2(b), 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 Figure 2(b), 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 2(b) 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 Figure 2(a).

Algorithm 2 shows the complete procedure for inserting the potential HUSPs found in batch B_k to

Algorithm 2 *Insert potential HUSPs into MAS-Tree*

Input: *MAS-Tree*, $HUSP_{B_k}$, *mechanismType*, *availMem*
Output: *MAS-Tree*, *currMem*

```
1: newPatSet_{B_k} ← ∅
2: for ∀S ∈ HUSP_{B_k} do
3:     N ← 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) ← nodeRsu(N) + rsu(S, B_k)
7:             nodeUtil(N) ← nodeUtil(N) + su(S, B_k)
8:         else
9:             Convert N to C-node
10:            nodeRsu(N) ← rsu(S, B_k) + maxUtil
11:            nodeUtil(N) ← su(S, B_k) + maxUtil
12:        end if
13:    else
14:        Add pair ⟨S, N⟩ to newPatSet_{B_k}
15:    end if
16: end for
17: for ∀⟨S, N⟩ ∈ newPatSet_{B_k} do
18:    CN ← 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 ← 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 ← 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 ← 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 ← 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-Tree
```

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.

3.4 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 efficiently 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 3.1, 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

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, $maxUtil$

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 Algorithm 2 and Algorithm 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 $O(NumPot \times L_{avg})$.

3.5 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 $DS_k = \langle B_1, B_2, \cdots, B_k \rangle$ 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 cannot 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 $nodeUtil(N) \geq (\delta - \epsilon) \cdot U_{DS_k}$.

3.6 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.

LEMMA 3.1. *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 3.2. *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 3.3. *For any potential HUSP α, if $su_{tree}(\alpha, DS) > maxUtil$, α must exist in MAS-Tree.*

Table 1: Dataset characteristics

Name	#Seq	Type	batchSize	availMem
DS1	10K	Dense	1K	100MB
Kosarak	25K	Sparse, Large	5K	200MB
DS2	100K	Dense, Large	10K	400MB

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 $nodeRsu(N)$ when N was last pruned, respectively. Based on Lemma 3.2, $su_{tree}(\beta, DS) \leq su_{apprx}(\beta, DS)$, 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 3.2. *Once the user requests HUSPs over data stream DS, if* $maxUtil \leq (\delta - \epsilon) \cdot U_{DS}$, *all the high utility sequential patterns will be returned.*

Proof. Suppose there is 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 3.1, $\epsilon \cdot U_{DS} > su(\alpha, DS) - su_{tree}(\alpha, DS)$. 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 3.3, α must exist in the tree. On the other hand, based on Lemma 3.2, $su_{apprx}(\alpha, DS) \geq su_{tree}(\alpha, DS) > (\delta - \epsilon) \cdot U_{DS}$. 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

To evaluate the performance of our proposed algorithm, experiments have been conducted on both synthetic datasets generated by the IBM data generator (*DS1:D10K-C10-T3-S4-I2-N1K*, *DS2:D100K-C8-T3-S4-I2-N10K*)[1], and the real-world *Kosarak* dataset [3]. Table 1 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 show the performance of the algorithms under different parameter values. In Section 4.4, we also conduct an analysis on a real web clickstream dataset obtained from a Canadian news portal.

We use the following performance measures: (1) *Precision* and *Recall*: the average precision and recall values over data streams: $precision = \frac{|appHUSPs \bigcap eHUSP|}{|appHUSPs|}$, $recall = \frac{|appHUSPs \bigcap eHUSP|}{|eHUSP|}$, where *eHUSP* is the true set of HUSPs and *appHUSPs* is the approximate set of HUSPs returned by a method. (2) *F-Score*: $2 \times \frac{precision \times recall}{precision + recall}$. (3) *Run Time*, (4)*Memory Usage*.

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 as comparison methods: (1) *NaiveHUSP*: this method is a fast method to approximate the utility of a sequence over the past batches using the utilities of items in the sequence. That is, the utility of each item over a data stream is tracked. 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: $su(\alpha, DS_i) = su(\alpha, B_i) + \sum_{I \in \alpha} u(I, DS_{i-1})$. (2) *RndHUSP*: this method is a memory adaptive HUSP mining method which adapts memory by pruning a subtree randomly, (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, we evaluate two versions of $MAHUSP$, named $MAHUSP\_S$ (which uses the $SBMA$ mechanism) and $MAHUSP\_L$ (which uses the $LBMA$ mechanism).

4.1 Effectiveness of MAHUSP Figure 3(a) shows the *precisions* of the methods on the three datasets. The proposed methods outperform *NaiveHUSP* and *RndHUSP* significantly. $MAHUSP\_L$ outperforms $MAHUSP\_S$ in the most of the cases in $DS1$ and $DS2$. 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 3(b) shows the *recalls* of the methods, which indicate that our proposed methods significantly outperform other methods in all the datasets. Indeed, on $DS1$, $MAHUSP\_L$ returns all the true patterns for each threshold value. Also, $MAHUSP\_S$ returns all the true patterns for most threshold values on $DS2$ and $Kosarak$. The results imply that the condition presented in Theorem 3.2 happens often and the proposed memory adaptive mechanisms prune the nodes effectively.

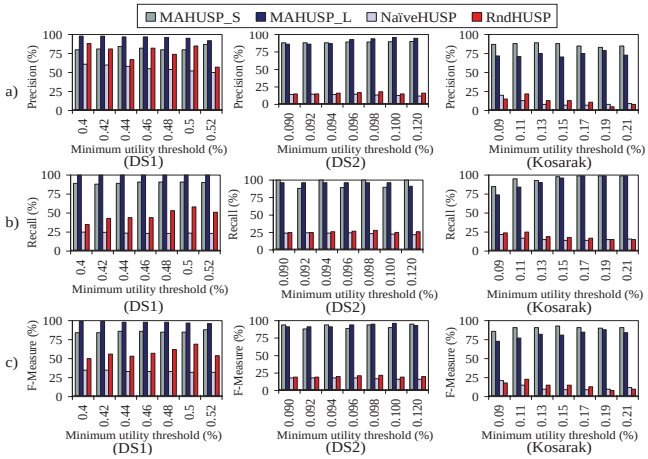

Figure 3: (a) Precision, (b) Recall and (c) F-Score performance on the different datasets

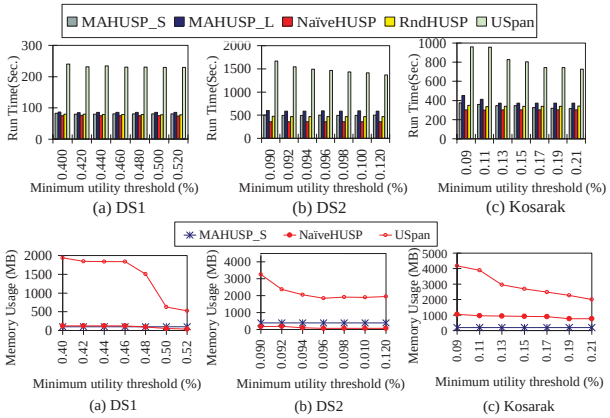

Figure 4: (a) Execution time, (b) Memory Usage on different datasets.

Figure 3(c) shows the *F-Score* values for the four methods with different δ values on the 3 datasets. Both proposed methods outperform the other methods significantly with an average F-score value of 90% over the $DS1$, $DS2$ and $Kosarak$ data sets.

4.2 Time and Memory Efficiency of MAHUSP Figure 4(a) 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 node. *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. Moreover, *MAHUSP* methods are only a bit slower than

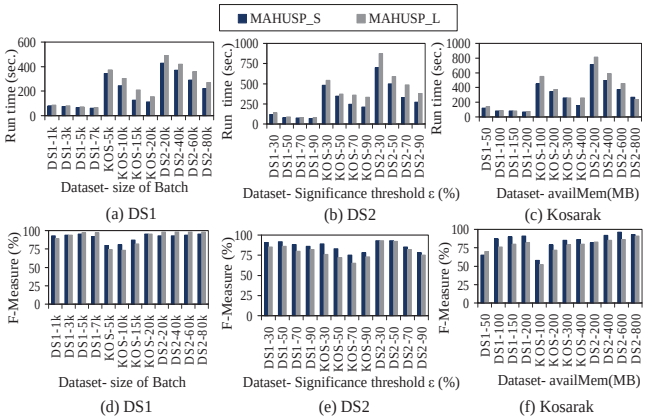

Figure 5: Parameter sensitivity on different datasets.

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.

Figure 4(b) 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*.

4.3 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% for $DS1$, $DS2$ and $Kosarak$ respectively. Figures 5(a),(d) present the results on *DS1*, *DS2* and *Kosarak* 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 5 (a) shows the trend in the execution time with different batch sizes. In all the data sets, the run time decreases as *batchSize* increases since increasing the batch size leads to generating less number of intermediate potential HUSPs. Figure 5 (d) shows F-Scores on different datasets. From Figure 5(d), we can observe that the F-score of the methods increases slowly with increasing batch sizes. Figures 5 (b)(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.

Figures 5(c),(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 5(c) 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 5 (f) 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.

4.4 A Real-life Application In this section, we analyze a real-world web clickstream dataset, called the *Globe* dataset obtained from a Canadian news web portal (*The Globe and Mail* [4]). The dataset was created based on a random sample of users visiting *The Globe and Mail* during a six months period in 2014. The dataset contains 116,000 sequences and 24,770 news. Each sequence in the dataset corresponds to news articles read by a subscribed user.

The goal is to take both news freshness and interestingness into account to discover behavioral patterns related to users' interest. We assume that the time user spends on a news 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 is defined as browsing time in seconds. In addition, since the importance of nw is 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.

We apply *MAHUSP* to discover HUSPs based on the above utility model. We also applied SPADE algorithm implemented by [3] to discover frequent sequential patterns (i.e.,*FSPs*) from the *Globe* dataset. Table 2 presents top-4 HUSPs and top-4 FSPs of length 2, sorted by time spent and support respectively. Table 2

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

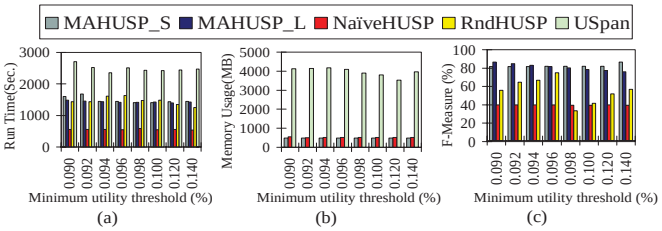

Figure 6: (a) Run time, (b) Memory Usage and (c) F-Measure Performance on the *Globe* dataset

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*). These patterns can be directly used to produce utility-based association rules to navigate users based on a semantic measure (e.g., news freshness and interestingness) rather than a statistical measure (e.g., support). They capture the user's information need more precisely, hence the generated rules can recommend more useful news to the user. They 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 6 shows *MAHUSP* outperforms the other methods significantly in terms of *run time*, *Memory usage* and *F-Measure*. The average F-measure value of *MAHUSP_S* on the *Globe* dataset is 87%.

5 Related Work

The concept of HUSP mining was first proposed by Ahmed et al [2], who defined an over-estimated sequence utility measure, SWU, which has the *downward closure property*, and proposed the UL and US algorithms for mining HUSPs which use SWU to prune the search space. Yin et al. [9] proposed the USpan algorithm for mining HUSPs. In this study, a lexicographic tree was used to extract the complete set of high utility itemset sequences. In [8], the authors proposed the HUS-Span algorithm based on two pruning strategies to identify HUSPs. However, all of the HUSP mining methods were designed for static datasets, not for data streams.

On the other hand, several studies [4, 5, 6] have been conducted to use approximate approaches to discover frequent patterns over the entire data stream. In [5], authors presented algorithms for computing frequency counts of items exceeding a user-specified threshold over data streams. Mendes et al. [6] proposed two methods

Table 2: 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(mins.)	Support
MAHUSP	HUSP1.	Retiree, 60, wonders how long her money will last Which is better,a RRIF or an annuity? You may be surprised	**1474**	152
	HUSP2.	Robin Williams warp-speed improvisation was almost too fast to be human CBC lays off veteran sportscasters amid budget cuts	**1471**	121
	HUSP3.	Israel prepares to 'significantly' expand campaign as UN chief... MH17: Disaster ratchets up Russia-Ukraine tensions	**1212**	116
	HUSP4.	Massive explosive decompression' downed MH17: Kiev Canada should learn from Ireland's housing crash	**994**	86
SPADE	FSP1.	CBC lays off veteran sportscasters amid budget cuts Celine Dion takes indefinite break to focus on health, family	576	**286**
	FSP2.	La Prairie, Quebec mayor dies from wasp stings Duffy billed taxpayers for attending funerals, RCMP allege	380	**254**
	FSP3.	Supreme Court sides with Ottawa in multibillion-dollar EI case MH17: Disaster ratchets up Russia-Ukraine tensions	536	**247**
	FSP4.	Controversial First Nation chief??s salary raises concern Harper sticks to hard line on Hamas; U.S. condemns Israel's deadly...	830	**220**

(i.e., SS-BE and SS-MB) inspired by [5] 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.

The only work on HUSP mining over data streams is proposed in [10]. The proposed method works based on the sliding window model and is not able to find HUSPs over the entire data stream. The authors proposed an upper bound, called *Suffix Utility* (i.e., SFU), to prune patterns during HUSP discovery. 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 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. Moreover, the data structure in [10] is not applicable to store the information over a long period of time.

6 Conclusions

We tackled the problem of memory adaptive HUSP mining over data streams. We proposed an approximation algorithm, called *MAHUSP*, to discover HUSPs over data streams. We proved that *MAHUSP* returns all the true HUSPs under certain circumstances. The experimental results showed that our method 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. As future work, we will extend our method to consider new types of resources such as CPU.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[2] C. F. Ahmed, S. K. Tanbeer, and B. Jeong. A novel approach for mining high-utility sequential patterns in sequence databases. *In ETRI Journal*, 32:676–686, 2010.

[3] P. Fournier-Viger, A. Gomariz, A. Soltani, and T. Gueniche. Spmf: Open-source data mining library. http://www.philippe-fournier-viger.com/spmf/, 2013.

[4] W.-Y. Lin, S.-F. Yang, and T.-P. Hong. Memory-aware mining of indirect associations over data streams. In *IDAM 2013*. Springer Netherlands, 2013.

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

[6] L. Mendes, B. Ding, and J. Han. Stream sequential pattern mining with precise error bounds. In *ICDM '08*, pages 941–946, 2008.

[7] C. H. Mooney and J. F. Roddick. Sequential pattern mining approaches and algorithms. *ACM Comput. Surv.*, 45(2):19:1–19:39, 2013.

[8] J.-Z. Wang, Z.-H. Yang, and J.-L. Huang. An efficient algorithm for high utility sequential pattern mining. In *Frontier and Innovation in Future Computing and Communications*, volume 301, pages 49–56. Springer Netherlands, 2014.

[9] J. Yin, Z. Zheng, and L. Cao. Uspan: An efficient algorithm for mining high utility sequential patterns. In *In Proc. of ACM SIGKDD*, pages 660–668, 2012.

[10] M. Zihayat, C.-W. Wu, A. An, and V. S. Tseng. Mining high utility sequential patterns from evolving data streams. In *ASE BigData*, Accepted, 2015.