Efficiently Mining High Utility Sequential Patterns in Static and Streaming Data

Morteza ZIHAYAT\textsuperscript{a,1}, Cheng-Wei WU\textsuperscript{b}, Aijun AN\textsuperscript{a} and Vincent S. TSENG\textsuperscript{b}

\textsuperscript{a}Department of Computer Science and Engineering, York University, Toronto, Canada
\textsuperscript{b}Department of Computer Science, National Chao Tung University, Taiwan, ROC

Abstract. High utility sequential pattern (HUSP) mining has emerged as a novel topic in data mining. Although some preliminary works have been conducted on this topic, they incur the problem of producing a large search space for high utility sequential patterns. In addition, they mainly focus on mining HUSPs in static databases and do not take streaming data into account, where unbounded data come continuously and often at a high speed. To efficiently deal with both problems, we propose a novel framework for mining high utility sequential patterns over static and streaming databases. In this regard, two efficient data structures named ItemUtilLists (Item Utility Lists) and HUSP-Tree (High Utility Sequential Pattern Tree) are proposed to maintain essential information for mining HUSPs in both offline and online fashions. In addition, a novel utility model called Sequence-Suffix Utility is proposed for effectively pruning the search space in HUSP mining. We propose an algorithm named HUSP-Miner (High Utility Sequential Pattern Miner) to find HUSPs in static databases efficiently. Then, a one-pass algorithm named HUSP-Stream (High Utility Sequential Pattern mining over Data Streams) is proposed to incrementally update ItemUtilLists and HUSP-Tree online and find HUSPs over data streams. To the best of our knowledge, HUSP-Stream is the first method to find HUSPs over data streams. Experimental results on both real and synthetic datasets show that HUSP-Miner outperforms the compared algorithms substantially in terms of execution time, memory usage and number of generated candidates. The experiments also demonstrate impressive performance of HUSP-Stream to update the data structures and discover HUSPs over data streams.

Keywords. High Utility Sequential Pattern Mining and Data Streams and Sliding Window.

1. Introduction

Mining useful patterns from sequential data is a challenging topic in data mining. An important task for mining from sequential data is sequential pattern mining [23,6], which discovers sequences of itemsets that frequently appear in a sequence database. For example, in market basket analysis, mining sequential patterns from sequences of customer transactions is to find the sequential lists of itemsets that are frequently purchased by customers in a time order. However, in the traditional framework of sequential pattern mining the number of occurrences of an item inside a transaction (e.g. purchased quan-
tity) is ignored in the problem setting, so is the importance (e.g., unit price/profit) of an item in the dataset. Thus, not only some infrequent patterns that bring high profits to the business may be missed, but also a large number of frequent patterns having low selling profits are discovered. For example, in retail businesses, the sale of a TV is much less frequent than the sale of a tooth brush, but it often brings much more profit than a tooth brush sale. However, patterns containing TV sales may not be discovered in frequent pattern mining due to their low frequency.

To address the issue, high utility sequential pattern (HUSP) mining [2,3,21,26] has emerged as a novel research topic in data mining recently. In HUSP mining, each item has a weight (e.g. price/profit) and can appear more than once in any transaction (e.g. purchase quantity), and the goal is to find sequences whose total utility (e.g., total profit) in the database is no less than a user-specified minimum utility threshold. Although, a few studies have been proposed for HUSP mining [2,3,21,26], they suffer from the problem of producing a large search space for high utility sequential patterns, especially when databases contain lots of long sequences or a low threshold is set. Hence, the key challenge is how to effectively prune the search space and efficiently capture all high utility sequential patterns. Moreover, the computational complexity of pattern extraction is prohibitively high. In order to lower this complexity, efficient data structures are needed to minimize the database scans in terms of both size and count.

On the other hand, the existing studies consider mainly static databases and do not take dynamic streaming data into consideration. A data stream is an ordered, continuous and unbounded sequence of data records (e.g., items, itemsets or transactions) that arrive often in a fast manner. In some applications such as network traffic monitoring and intrusion detection, users are more interested in information that reflect recent data rather than old ones. The most common data stream processing model to discover such information is based on sliding windows. Given a user-specified window size $w$, the sliding window based model captures $w$ most recent records in a window, and focuses on discovering the patterns within the window. When a new record flows into the window, if there are already $w$ data records in the window, the oldest one is removed from the window. In this model, the effect of the expired data are eliminated, and the patterns are mined from the recent data in the window.

Some studies [8,9,14,18,7,21] have been conducted on efficiently mining frequent sequences over data streams using the sliding window model. However, existing methods have the following deficiencies: (1) They are frequency-based, and did not consider the utility (e.g., profit) of an item and thus cannot be used to find HUSPs over sliding windows. (2) Most of the studies such as [9,18] focused on mining sequential patterns over a stream of items and few considered the scenario of a stream of itemsets so that the sequential relationships between itemsets are lost [14]. However, itemset-sequences are often encountered in real-life applications (e.g., market basket analysis). (3) Generally speaking, the update operations on a sliding window can be categorized into four types: (i) inserting new sequences, (ii) deleting existing sequences, (iii) appending new items/itemsets to the existing sequences and (iv) dropping items/itemsets from the existing sequences. However, very few preliminary works have been proposed for mining patterns on all the types of update in a unified framework.

In order to mine HUSPs over data streams using sliding windows, a naive approach is to apply existing (static) HUSP mining algorithms to rerun the whole mining process on the updated window whenever a data record comes into or an old one leaves from the
window. Obviously, the computational cost of this approach may be prohibitively high, especially when data records arrive at a rapid rate and the database changes quickly.

Although mining HUSPs over both static and data streams is very desirable in many real-life applications such as user behavior analysis, web mining and high-velocity big data analysis, addressing this topic is not an easy task due to the following challenges:

- A HUSP mining algorithm needs to overcome the large search space problem due to combinatorial explosion of sequences. Effectively pruning the search space for mining high utility sequential patterns is difficult, because the downward closure property does not hold for the utility of sequences. That is, the utility of a sequence may be higher than, equal to, or lower than that of its super/sub-sequences [3,21,26]. Thus, search space pruning techniques that rely on the downward closure property cannot be directly used for mining high utility sequential patterns.

- Mining high utility sequential patterns over a stream of itemset-sequences is not a trivial task. In the itemset-sequence stream, different items can occur simultaneously. This is substantially different and much more challenging than mining the patterns over a stream of item-sequences. Since items with different quantities and unit profits can occur simultaneously in any data record of itemset-sequence streams, the search space is much larger and the problem is much more challenging than mining HUSPs over streams of item-sequences.

- Streaming data usually come continuously, unbounded and at a high speed. Keeping all the data records in memory (even on disk) is infeasible and real-time processing of each new incoming record is required. On the other hand, once a data record is removed, it is impossible to backtrack over previously data records that have been expelled from the window. Hence, how to efficiently discover HUSPs over data streams by reading data records only once using limited computing and storage capabilities is a challenging problem.

- Data distribution in a stream usually changes over time such that a low (or high) utility pattern can become a high (or low) utility pattern later on and hence cannot be ignored. Comparing to mining HUSPs from a static dataset, mining HUSPs over dynamic data streams has far more information to track and far greater complexity to manage. How to efficiently discover correct HUSPs over a data stream is a challenging problem.

In this work, we address all of the above deficiencies and challenges by proposing a new framework for high utility sequential pattern mining over static and streaming data. Our framework learns HUSPs from both static data and a sliding window over data streams of itemset-sequences. The major contributions of this work are summarized as follows.

1. We propose a novel over-estimate utility model, called Sequence-Suffix Utility (SFU). We prove that SFU of a sequence is an upper bound of the utilities of some of its super-sequences, which can be used to effectively prune the search space in finding HUSPs. The experiments show that SFU is more effective in pruning the search space than the previously-proposed SWU (Sequence-Weighted Utility) model [2] for HUSP mining.

2. We propose two efficient data structures named ItemUtilLists (Item Utility Lists) and HUSP-Tree (High Utility Sequential Pattern Tree) for maintaining the essential information of high utility sequential patterns. To the best of our knowl-
edge, the `ItemUtilLists` structure is the first vertical data representation for HUSP mining over data streams that can be used to efficiently calculate the utility of sequences. These data structures can be built using one scan of data, allow easy update when the window slides, and can be used to compute sequence utilities without re-scanning the transactions in the sliding window.

3. We propose a new algorithm called `HUSP-Miner` (High Utility Sequential Pattern Miner) to find HUSPs. `HUSP-Miner` constructs `ItemUtilLists` and `HUSP-Tree` by scanning the database only once and uses both SFU and SWU to prune the size of `HUSP-Tree` efficiently.

4. We incorporate the concept of stream mining into HUSP mining and formally define the new problem of sliding window-based high utility sequential pattern mining over data streams.

5. We also propose a new one-pass algorithm called `HUSP-Stream` (High Utility Sequential Pattern Mining over Data Streams) for efficiently updating `ItemUtilLists` and `HUSP-Tree` by reading a transaction in the data stream only once. When data arrive at or leave from the window, our method incrementally updates `ItemUtilLists` and `HUSP-Tree` to find HUSPs based on previous mining results without re-running the whole mining process on the updated window. It supports four types of update in a unified framework, including (a) inserting sequences, (b) deleting sequences, (c) appending new items/itemsets to the existing sequences and (d) dropping items/itemsets from the existing sequences.

6. We conduct extensive experiments on both real and synthetic datasets to evaluate the performance of the proposed algorithms. Experimental results show that `HUSP-Miner` and `HUSP-Stream` outperform the state-of-the-art HUSP mining algorithm [26] substantially in terms of execution time, the number of generated candidates and memory usage. In particular, `HUSP-Stream` runs very well in some cases where `USpan` [26], a state-of-the-art HUSP mining algorithm, fails to complete the mining task.

The remaining of the paper is organized as follows. In Section 2, we discuss related work. Section 3 provides definitions and a problem statement. Section 4 presents the proposed algorithms and data structures. Experimental results are shown in Section 5. We conclude the paper in Section 6.

2. Related Work

In this section, we describe some existing work on sequential pattern mining, high utility sequential pattern mining, and data stream mining.

2.1. Sequential Pattern Mining

Mining sequential patterns in sequence databases is one of the most challenging problems in data mining [6,13,19,23,27], which was first introduced by Agrawal et al [1]. A subsequence is called a sequential pattern or frequent sequence if it frequently appears in a sequence database, and its frequency is no less than a user-specified minimum support threshold [1]. Sequential pattern mining plays an important role in data mining and several related algorithms have been proposed such as `AprioriAll` [6], `GSP` [23], `FreeSpan`
PrefixSpan [19], SPADE [27] and SPAM [1]. 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 and SPAM). 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 dense databases or long sequences than algorithms using the horizontal format. 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.

2.2. High Utility Sequential Pattern Mining

High utility pattern mining [2,3,21,26] considers the external utility (e.g., unit profits) and internal utility (e.g., quantity) of items such that it provides users with patterns having a high utility (e.g., profit). Some efficient algorithms such as two-phase [17], IHUP [4], UP-Growth [24], HUI-Miner [16] and FHM [12] have been proposed to find high utility itemsets (HUIs) from a transaction database, where the sequential ordering of itemsets is not considered. The addition of ordering information makes the pattern mining problem fundamentally different and much more challenging than mining high utility itemsets. The integration of sequential pattern mining and utility mining has taken place very recently. The concept of high utility sequential pattern (HUSP) mining was first proposed by Ahmed et al. [2], who defined an over-estimated sequence utility measure, $SWU$ (i.e., Sequence-Weighted Utility), which has the downward closure property, and proposed two approaches, called $UL$ and $US$, to find HUSPs based on $SWU$. $UL$ is a level-wise candidate generation-and-testing algorithm and hence involves multiple scans of the database and generates a large number of high-$SWU$ candidate sequences. They also proposed $US$ which uses a pattern-growth method inspired by PrefixSpan [19] to generate all sequences whose $SWU$ satisfies the threshold, and then scans the database again to compute the exact utilities of high-$SWU$ candidate sequences to find HUSPs. Shie et al. [21] proposed a framework for mining HUSPs in a mobile environment. Their algorithm can only handle sequences with a single item in each sequence element. Ahmed et al. proposed efficient algorithms for mining high utility access sequences from web log data [3], which also only consider single-item sequences. Recently, Yin et al. [26] proposed the USpan algorithm for mining HUSPs. They used a lexicographic tree to extract the complete set of high utility itemset-sequences and designed mechanisms for expanding the tree with two pruning strategies. One of the pruning strategy is based on $SWU$. The other pruning strategy needs to be used after candidate generation. Moreover, it needs to construct $Utility\ Matrix$ (the proposed data structure) for each generated sequence and also it traverses each element once to calculate the utility of extended sequence, which is very time consuming. In addition, all of the HUSP mining methods were designed for static datasets, not for data streams.

2.3. Stream Mining

Due to the widespread existence of data streams, stream mining has become an important and challenging topic in data mining [25,22,5,15,10,8,9,14,28,18,7,21]. Since a data
Table 1. Summary of notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u(X,S^d_t))</td>
<td>Utility of item/itemset (X) in transaction (T_d) of sequence (S^d)</td>
</tr>
<tr>
<td>(TU(S^d_t))</td>
<td>Utility of transaction (T_d) of sequence (S^d)</td>
</tr>
<tr>
<td>(\alpha \preceq \beta)</td>
<td>(\alpha) is a subsequence of (\beta), or (\alpha) occurs in (\beta)</td>
</tr>
<tr>
<td>(OccSet(\alpha,S^d))</td>
<td>Set of all the occurrences of (\alpha) in sequence (S^d)</td>
</tr>
<tr>
<td>(su(\alpha,S^d))</td>
<td>Utility of a sequence (\alpha) in sequence (S^d)</td>
</tr>
<tr>
<td>(\alpha \oplus I)</td>
<td>Itemset-extended of sequence (\alpha) and item (I)</td>
</tr>
<tr>
<td>(\alpha \otimes I)</td>
<td>Sequence-extended of sequence (\alpha) and itemset ({I})</td>
</tr>
<tr>
<td>(TSWU(\alpha,SW_i))</td>
<td>Sequence weighted utility of sequence (\alpha) in (SW_i)</td>
</tr>
<tr>
<td>(TSWU(\alpha,D))</td>
<td>Sequence weighted utility of sequence (\alpha) in database (D)</td>
</tr>
<tr>
<td>(suffix(S^d,\alpha))</td>
<td>Suffix of sequence (S^d) w.r.t sequence (\alpha)</td>
</tr>
<tr>
<td>(SFU(\alpha,D))</td>
<td>Sequence-suffix utility of sequence (\alpha) in (D)</td>
</tr>
<tr>
<td>(SFU(\alpha,SW_i))</td>
<td>Sequence-suffix utility of sequence (\alpha) in (SW_i)</td>
</tr>
</tbody>
</table>

Stream is an unbounded, fast, and dynamically-changing flow of data, a stream mining algorithm is often required to process each data record only once, and only the most recent or relevant data can be stored in memory. 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. Many studies such as [8,9,14,18,7,21] have been conducted to mine frequent sequential patterns over data streams. For example, Ho et al. proposed IncSPAM [14] to find sequential patterns over a data stream of itemset-sequences. Rassi et al. proposed the SPEED algorithm [7] for mining maximal sequential patterns over streaming data. Chang et al. proposed SeqStream [8] for mining closed sequential patterns over data streams. However, all these methods are for finding frequent sequential patterns and some useful infrequent patterns with high utility may be missed.

Although many studies have been conducted for mining sequential patterns over data streams, they may have the following deficiencies: (1) They are not developed for high utility sequential pattern mining and thus may produce too many patterns with low utility (e.g., low profit); (2) Most of these algorithms discover patterns from a stream of item-sequences instead of itemset-sequences so that the sequential relationships between itemsets or items are lost.

To the best of our knowledge, our proposed method in [29] is the only work on \(HUSP\) mining over data streams. In this paper, we extend [29] as follows. First, we propose a new algorithm (i.e., \(HUSP\)-Miner) to find \(HUSPs\) in static databases. The correctness of the proposed method is provided in Section 4.3. Second, the newly proposed algorithm is compared with the algorithm in [26] in the experiments. Third, the experimental results for \(HUSP\)-Stream are extended by (1) adding the results on a new dataset called \(DS1\), (2) evaluating the effectiveness of the proposed pruning strategy, (3) showing the scalability of \(HUSP\)-Stream and (4) investigating the parameter sensitivity of the proposed method.
3. Definitions and Problem Statement

Let $I^* = \{I_1, I_2, \ldots, I_K\}$ be a set of items. An itemset is a set of distinct items. An itemset-sequence $S$ (or sequence in short) is an ordered list of itemsets $(X_1, X_2, \ldots, X_Z)$, where $Z$ is the size of $S$. The length of $S$ is defined as $\sum |X_i|$. An L-sequence is a sequence of length $L$. A sequence database $D$ consists of a set of sequences $\{S_1, S_2, \ldots, S_K\}$, in which each sequence $S_i$ has a unique sequence identifier $r$ called SID and consists of an ordered list of transactions $(T_{d_1}, T_{d_2}, \ldots, T_{d_n})$, where each transaction $T_{d_i} \in S_r$ is an itemset and has a unique global transaction identifier $d_i$ called TID. A transaction $T_{d_i}$ in the sequence $S_r$ is also denoted as $S^d_i$.

**Definition 1 (External utility and internal utility)** Each item $I \in I^*$ is associated with a positive number $p(I)$, called its external utility, representing, e.g., the unit profit of $I$. Also, an item $I$ in transaction $T_d$ has a positive number $q(I, T_d)$, called its internal utility, representing, e.g., the quantity of $I$ in $T_d$.

Figure 1 (a) shows a sequence database with five transactions. Figure 1 (b) presents the external utility (e.g., profit) of each item in the database. The internal utility (e.g., quantity) of an item in a transaction is shown in the transaction. For example, $(c, 3)$ in $T_4$ (which belongs to sequence $S_1$) means $q(c, S^d_1) = 3$.

**Definition 2 (Super-sequence and sub-sequence)** For non-empty sequences $\alpha = \langle X_1, X_2, \ldots, X_i \rangle$ and $\beta = \langle X'_1, X'_2, \ldots, X'_j \rangle$ ($i \leq j$), $\alpha$ is a sub-sequence of $\beta$ or equivalently $\beta$ is a super-sequence of $\alpha$ if and only if there exist integers $1 \leq e_1 < e_2 < \ldots < e_i \leq j$ such that $X_1 \leq X'_{e_1}, X_2 \leq X'_{e_2}, \ldots, X_i \leq X'_{e_i}$ (denoted as $\alpha \preceq \beta$).

**Definition 3 (Utility of an item in a transaction)** The utility of an item $I$ in the transaction $T_{d_i}$ of the sequence $S_r$ is defined as $u(I, S^d_i) = p(I) \times q(I, S^d_i)$.

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

**Definition 5 (Transaction utility)** The transaction utility of transaction $S^d_i \in D$ is denoted as $TU(S^d_i)$ and computed as $u(S^d_i, S^d_i)$.

---

**Table:**

<table>
<thead>
<tr>
<th>Item</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
For example, \( u(b, S_1^1) = p(b) \times q(b, S_1^1) = 3 \times 3 = 9 \), and \( u(\{bc\}, S_1^1) = u(b, S_1^1) + u(c, S_1^1) = 9 + 2 = 11 \). Therefore, transaction utility of \( S_1^1 \) is \( TU(S_1^1) = 2 \times 2 + 3 \times 3 + 1 \times 2 = 15 \).

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

That is, the maximum utility of a sequence \( \alpha \) among all its occurrences in \( S_r \) is used as its utility in \( S_r \). This definition is consistent with the sequence utility definition in [26].

**Definition 7 (Utility of a sequence in sequence database \( D \))** The utility of a sequence \( \alpha \) in \( D \) is defined as \( su(\alpha, D) = \sum_{S_r \in D} su(\alpha, S_r) \).

**Definition 8 (High utility sequential pattern (HUSP))** A sequence \( \alpha \) is called a high utility sequential pattern (HUSP) iff \( su(\alpha, D) \) is no less than a user-specified minimum utility threshold \( \delta \).

**Problem statement 1.** Given a minimum utility threshold \( \delta \), the problem of mining high utility sequential patterns (HUSPs) over a dataset \( D \) is to discover all sequences of itemsets whose utility is no less than \( \delta \).

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

4. **High Utility Sequential Pattern Mining in Static and Streaming Data**

In this section we first propose an algorithm named \( \text{HUSP-Miner} \) (High Utility Sequential Pattern Miner) to find HUSPs in static databases. Then, we present an extended version of \( \text{HUSP-Miner} \) named \( \text{HUSP-Stream} \) (High Utility Sequential Pattern mining over data Stream) for incrementally mining the complete set of HUSPs in the current sliding window \( SW_i \) (to be defined later) of a data stream. We propose a vertical representation of a dataset called \( \text{ItemUtilLists} \) (Item Utility Lists) and a tree-based data structure, called \( \text{HUSP-Tree} \) (High Utility Sequential Pattern Tree), to model the essential information to discover HUSPs.

4.1. **HUSP-Miner: High Utility Sequential Pattern Miner**

The overview of \( \text{HUSP-Miner} \) is as follows. The algorithm includes two main phases: (1) \( \text{Construction phase} \), (2) \( \text{HUSP mining phase} \). In the construction phase, the \( \text{ItemUtilLists} \) structure is constructed for storing the utility information for every item in the input transaction \( S_r \). Then, \( \text{HUSP-Tree} \) is constructed for maintaining potential high utility sequential patterns (to be defined later) using the information stored in \( \text{ItemUtilLists} \). After the construction phase, \( \text{HUSP-Miner} \) finds all the HUSPs from the potential HUSPs stored in \( \text{HUSP-Tree} \).
4.1.1. Construction phase

In this phase, HUSP-Miner reads the transactions in the dataset one by one to construct ItemUtilLists and HUSP-Tree. Below we first introduce these two data structures and then explain how to construct them in the construction phase.

4.1.1.1. ItemUtilLists (Item Utility Lists)  The first component of the proposed algorithm is an effective representation of items to restrict the number of generated candidates and to reduce the processing time and memory usage to mine HUSPs. ItemUtilLists is a vertical representation of the transactions in the dataset. The ItemUtilLists of an item \( I \) consists of several tuples. Each tuple stores the utility of item \( I \) in the transaction \( S_v \) (i.e., transaction \( T_u \) in sequence \( S_v \)) that contains \( I \). Each tuple has three fields: SID, TID, and Util. Fields SID and TID store the identifiers of \( S_v \) and \( T_u \), respectively. Field Util stores the utility of \( I \) in \( S_v \) (Definition 3). Figure 2 shows ItemUtilLists for the dataset presented in Figure 1, which is computed easily by using the external utility of \( I \) and the internal utility of \( I \) in \( S_v \). ItemUtilLists can be implemented with an array of hash maps with item ID as the index for the array. An element of the array (i.e., the ItemUtilList of an item) can be implemented using a hash map with SID and TID as the key. Thus, a direct access can be done to a tuple in ItemUtilLists. The average time complexity for constructing ItemUtilLists is \( O(M \times L_{avg}) \), where \( M \) is the size of the dataset and \( L_{avg} \) is the average length of these transactions.

4.1.1.2. HUSP-Tree Structure  In this section, we propose an efficient tree-based data structure called HUSP-Tree, to maintain the essential information for mining HUSPs. The structure of HUSP-Tree is similar to the lexicographic tree proposed in [14] and LQS-Tree [26]. These trees are used to enumerate sequential patterns in a sequence database. The main difference among different lexicographic trees is in the node structure and the content stored in the node. The node structure and content are important as they determine what can be done during the mining process. For example, the tree proposed in [14] is used to find frequent sequential patterns, hence a node in this tree mainly stores the frequency of a sequence represented by the node.

A HUSP-Tree is a lexicographic sequence tree where each non-root node represents a sequence of itemsets. Figure 3 shows part of the HUSP-Tree for the the dataset in Figure 1, where the root is empty. Each node at the first level under the root represents a sequence of length 1, a node on the second level represents a 2-sequence, and all the child

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{ItemUtilLists for items in \( D \) in Figure 1}
\end{figure}
nodes of a parent are listed in alphabetic order of their represented sequences. There are two types of child nodes for a parent: \textit{I-node} and \textit{S-node}, which are defined as follows.

\textbf{Definition 9 (Itemset-extended node (I-node))} Given a parent node \( p \) representing a sequence \( \alpha \), an \textit{I-node} is a child node of \( p \) which represents a sequence generated by adding an item \( I \) into the last itemset of \( \alpha \) (denoted as \( \alpha \oplus I \)).

\textbf{Definition 10 (Sequence-extended node (S-node))} Given a parent node \( p \) representing a sequence \( \alpha \), an \textit{S-node} is a child node of \( p \) which represents a sequence generated by adding a 1-Itemset \( \{ I \} \) after the last itemset of \( \alpha \) (denoted as \( \alpha \odot I \)).

In Figure 3, the node for sequence \( \langle \{ abc \} \rangle \) is an \textit{I-node}, while the node for \( \langle \{ ab \} \{ c \} \rangle \) is a \textit{S-node}. Their parent is \( \{ ab \} \).

Since \textit{HUSP-Tree} is used to store information in both static and streaming data, it should address two main issues. First, it is not possible to re-construct a tree when a new transaction arrives at or leaves from the window and the data structure should be able to update itself efficiently. Second, the size of the tree can be huge since the number of possible patterns is exponential in the number of items in the database. To avoid generating such a tree, we need to design strategies to prune the tree so that only the nodes representing potential \textit{HUSPs} (to be defined later) are generated. These strategies will be presented later in this section. Moreover, we need to store summarized information regarding potential \textit{HUSPs} to prune the tree during tree construction and updating, and also to compute the exact utility of potential \textit{HUSPs} during the \textit{HUSP} mining phase. Hence, we design each non-root node of a \textit{HUSP-Tree} to have a field, called \textit{SeqUtilList}, for storing such information about the sequence represented by a node.
**Definition 11 (Sequence Utility List)** The sequence utility list (SeqUtilList) of a sequence $\alpha$ is a list of three-value tuples, where each tuple $\langle\text{SID}, \text{TID}, \text{Util}\rangle$ represents an occurrence of $\alpha$ in a sequence of the dataset and the utility of $\alpha$ with respect to the occurrence. The SID in a tuple is the ID of a sequence in which $\alpha$ occurs, TID is the ID of the last transaction in the occurrence of $\alpha$, and Util is the utility of $\alpha$ with respect to the occurrence. The tuples in a SeqUtilList are ranked first by SID and then by TID. If multiple occurrences of $\alpha$ have the same SID and TID, only the tuple with the highest Util value is kept in SeqUtilList. The SeqUtilList of $\alpha$ is denoted as SeqUtilList($\alpha$).

For example, given sequence $\alpha = \langle\{a\}\{c\}\rangle$ in Figure 1, since $\alpha$ has two occurrences, which are $\langle T_1, T_2 \rangle$ and $\langle T_1, T_4 \rangle$, the SeqUtilList of $\alpha$ is $\{\langle S_1, T_2, (4 + 1)\rangle, \langle S_1, T_4, (4 + 3)\rangle\} = \{\langle S_1, T_2, 5\rangle, \langle S_1, T_4, 7\rangle\}$.

**4.1.1.3. Major Steps in HUSP-Tree Construction** HUSP-Tree is constructed recursively in a top-down fashion using ItemUtilLists. The first level of the tree under the root is constructed by using the items in ItemUtilLists as nodes. The SeqUtilList of these nodes is the ItemUtilList of the items. Given a non-root node, its child nodes are generated using I-Step and S-Step, which generate I-nodes and S-nodes respectively.

Given a node $N$ representing sequence $\alpha$, I-Step generates all the I-nodes of $N$ (Definition 9). We define I-Set of $\alpha$ as the set of items occurring in the database (i.e., in ItemUtilLists) that are ranked alphabetically after the last item in $\alpha$. In I-Step, given an item $I$ in the I-Set of $\alpha$, for each tuple $T_p = \langle s, t, u \rangle$ in SeqUtilList($\alpha$), if there is a tuple $T_p' = \langle s', t', u' \rangle$ in ItemUtilList($I$) such that $s = s'$ and $t = t'$, then add a new tuple $\langle s, t, (u + u') \rangle$ to SeqUtilList($\beta$), where $\beta = \alpha \oplus I$, and SeqUtilList($\beta$) was initialized to empty before the I-Step. An I-node representing $\beta$ is added as a child node of $N$ if SeqUtilList($\beta$) is not empty.

For example, if $\alpha = \langle\{a\}\rangle$ and $I = b$. To construct SeqUtilList of $\beta = \alpha \oplus I = \langle\{ab\}\rangle$, we find the tuples for common transactions from SeqUtilList($\langle\{a\}\rangle$) = $\{\langle S_1, T_1, 4\rangle, \langle S_2, T_5, 8\rangle\}$ and ItemUtilList($b$) = $\{\langle S_1, T_1, 9\rangle, \langle S_1, T_3, 3\rangle, \langle S_2, T_5, 12\rangle, \langle S_2, T_5, 15\rangle\}$, which are the ones containing $\langle S_1, T_1 \rangle$ and $\langle S_2, T_5 \rangle$. Hence, SeqUtilList($\langle\{ab\}\rangle$) is $\{\langle S_1, T_1, (4 + 9)\rangle, \langle S_2, T_5, (8 + 15)\rangle\} = \{\langle S_1, T_1, 13\rangle, \langle S_2, T_5, 23\rangle\}$.

S-Step generates all the S-nodes for a non-root node. Given a node $N$ for sequence $\alpha$, the S-Set of $\alpha$ contains all the items that occur in the sliding window. The S-Step checks each item $I$ in the S-Set to generate the S-nodes of $N$ as follows. Let $\beta = \alpha \otimes I$ (i.e., a sequence by adding itemset $\{I\}$ to the end of $\alpha$). First, SeqUtilList($\beta$) is initialized to empty. For each tuple $T_p = \langle s, t, u \rangle$ in SeqUtilList($\alpha$), if there is a tuple $T_p' = \langle s', t', u' \rangle$ in ItemUtilList($I$) such that $s = s'$ and $t < t'$ (i.e., $t'$ occurs after $t$), then a new tuple $\langle s, t', (u + u') \rangle$ is added to SeqUtilList($\beta$). If SeqUtilList($\beta$) is not empty, an S-node is created under the node $N$ to represent $\beta$.

For example, if $\alpha = \langle\{ab\}\rangle$ and $I = d$. To construct SeqUtilList of $\beta = \alpha \otimes I = \langle\{ab\}\{d\}\rangle$, we need to find the tuples that satisfy the above conditions from SeqUtilList($\langle\{ab\}\rangle$) = $\{\langle S_1, T_1, 13\rangle, \langle S_2, T_5, 23\rangle\}$ and ItemUtilList($d$) = $\{\langle S_1, T_2, 4\rangle, \langle S_1, T_4, 4\rangle\}$. The tuple $\langle S_1, T_1, 13\rangle$ in SeqUtilList($\langle\{ab\}\rangle$) and two tuples $\langle S_1, T_2, 4\rangle$ and $\langle S_1, T_4, 4\rangle$ in ItemUtilList($d$) satisfy the conditions. Hence, SeqUtilList($\langle\{ab\}\{d\}\rangle$) is $\{\langle S_1, T_2, (13 + 4)\rangle, \langle S_1, T_3, (13 + 4)\rangle\} = \{\langle S_1, T_2, 17\rangle, \langle S_1, T_3, 17\rangle\}$.

Figure 4 shows the details of I-Step and S-Step to construct SeqUtilLists of $\langle\{ab\}\rangle$ and $\langle\{ab\}\{d\}\rangle$. 

4.1.1.4. Pruning Strategies  

In HUSP mining, the downward closure property does not hold for the sequence utility. Hence, the search space cannot be pruned as it is done in traditional sequential pattern mining. To effectively prune the search space, the concept of Sequence-Weighted Utility (SWU) was proposed in [2] to serve as an over-estimate of the true utility of a sequence, which has the downward closure property.

Below we integrate SWU into our proposed framework. This model is called Transaction based Sequence-Weighted Utility (TSWU) and we prove that TSWU has downward closure property.

**Definition 12** The Transaction based Sequence-Weighted Utility (TSWU) of a sequence $\alpha$ in the database $D$, denoted as $\text{TSWU}(\alpha, D)$, is defined as the sum of the utilities of all the transactions in all the sequences containing $\alpha$ in $D$:

\[
\text{TSWU}(\alpha, D) = \sum_{S \in D : \alpha \preceq S} \sum_{T \in S} TU(T)
\]

where $TU(T)$ is the utility of transaction $T$, and $\alpha \preceq S$ means $\alpha$ is a subsequence of $S$.

For example, in Figure 1, there are two sequences $S_1$ and $S_2$ contain the sequence $\langle \{b\} \{c\} \rangle$. The TSWU of $\langle \{b\} \{c\} \rangle$ in $D$ is $\text{TSWU}(\langle \{b\} \{c\} \rangle, D) = (15+8+7) + (12+24) = 66$.

Since it uses the utilities of all the transactions of all the sequences containing $\alpha$ in $D$, TSWU of sequence $\alpha$ is an over-estimate of the utility of $\alpha$ (i.e., Definition 7).

**Theorem 1** Given a sequence database $D$ and two sequences $\alpha$ and $\beta$ such that $\alpha \preceq \beta$, $\text{TSWU}(\alpha, D) \geq \text{TSWU}(\beta, D)$.

**Proof 1** Let $DS_\alpha$ be the set of sequences containing $\alpha$ in $D$ and $DS_\beta$ be the set of sequences containing $\beta$ in $D$. Since $\alpha \preceq \beta$, $\beta$ cannot be present in any sequence where $\alpha$ does not exist. Therefore, $DS_\beta \subseteq DS_\alpha$. Thus, according to Definition 12 $\text{TSWU}(\alpha, D) \geq \text{TSWU}(\beta, D)$.

Since TSWU has the downward closure property, we can use it to prune the HUSP-Tree.
Pruning Strategy 1 (Pruning by TSWU): Let $\alpha$ be the sequence represented by a node $N$ in the HUSP-Tree and $\delta$ be the minimum utility threshold. If $\text{TSWU}(\alpha, D) < \delta$, there is no need to expand node $N$. This is because the sequence $\beta$ represented by a child node is always a super-sequence of the sequence represented by the parent node. Hence $\text{su}(\beta, D) \leq \text{TSWU}(\beta, D) \leq \text{TSWU}(\alpha, D) < \delta$, meaning $\beta$ cannot be a HUSP.

Since TSWU uses the utilities of all the transactions of all the sequences containing $\alpha$, it overestimates the utility of a sequence too loosely. Below we propose another over-estimate of the utility of a sequence, called Sequence-Suffix Utility (SFU), and then develop a new pruning strategy based on SFU.

Definition 13 (First occurrence of sequence $\alpha$ in sequence $S_r$) Let $\bar{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_7} \rangle$ be an occurrence of a sequence $\alpha$ in the sequence $S_r$. $\bar{o}$ is called the first occurrence of $\alpha$ in $S_r$ if the last transaction in $\bar{o}$ (i.e., $T_{e_7}$) occurs before the last transaction of all occurrences in $\text{OccSet}(\alpha, S_r)$.

For example, there are two occurrences of $\{a\}\{c\}$ in $S_1$ in SW1 in Figure 5 (i.e., $\langle T_1, T_2 \rangle$ and $\langle T_1, T_3 \rangle$) where $\langle T_1, T_2 \rangle$ is the first occurrence because $T_2$ occurs earlier than $T_4$.

Definition 14 (Suffix of a sequence $S_r$ w.r.t. a sequence $\alpha$) Given sequence $\bar{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_7} \rangle$ as the first occurrence of $\alpha$ in $S_r$. The suffix of $S_r$ w.r.t. $\alpha$ (denoted as $\text{suffix}(S_r, \alpha)$) is the list of transactions in $S_r$ after the last transaction in $\bar{o}$ (i.e., after $T_{e_7}$).

Definition 15 (Sequence-Suffix utility of sequence $\alpha$ in sequence $S_r$) Given sequence $\alpha \preceq S_r$, the sequence-suffix utility (SFU) of $\alpha$ in $S_r$ is defined as follows:

$$\text{SFU}(\alpha, S_r) = \text{su}(\alpha, S_r) + \sum_{T \in \text{suffix}(S_r, \alpha)} \text{TU}(T)$$

where $\text{TU}(T)$ is the utility of transaction $T$.

In other words, the sequence-suffix utility of a sequence in $S_r$ is the utility of $\alpha$ in $S_r$ plus the sum of the utilities of the transactions in the suffix of $S_r$ with respect to $\alpha$.

Note that for any non-root node $N$ in the HUSP-Tree, $\text{SFU}(\alpha, S_r)$ can be computed easily using the information in the $\text{SeqUtilList}$ of $N$. According to Definition 6, $\text{su}(\alpha, S_r) = \max_{\bar{o} \in \text{OccSet}(\alpha, S_r)} \{\text{su}(\alpha, \bar{o})\}$ which can be obtained using the highest $\text{Util}$ value among all the tuples with $S_r$ as its $\text{SID}$. The $\text{TID}$ field of the first tuple stores the $\text{TID}$ of the last transaction in $\alpha$’s first occurrences in $S_r$. With this $\text{TID}$ value, we can easily get the $\text{TIDs}$ of all the transactions in $\text{suffix}(S_r, \alpha)$, and obtain their $\text{TU}$ values (which were pre-computed and stored when a transaction was scanned to build $\text{ItemUtilLists}$). For example, the sequence-suffix utility of $\alpha = \{a\}\{c\}$ in $S_1$ in Figure 5 is calculated as follows. According to $\text{SeqUtilList}(\alpha) = \{\langle S_1, T_2, 5 \rangle, \langle S_1, T_3, 7 \rangle\}$, $\text{su}(\alpha, S_1) = \max(5, 7) = 7$ and $\text{suffix}(S_1, \alpha) = \{T_3\}$. Hence, $\text{SFU}(\alpha, S_1) = 7 + \text{TU}(T_3) = 7 + 7 = 14$.

Definition 16 The $\text{SFU}$ of a sequence $\alpha$ in the dataset $D$, denoted as $\text{SFU}(\alpha, D)$, is defined as follows: $\text{SFU}(\alpha, D) = \sum_{S \in D} \text{SFU}(\alpha, S)$. 
**Property 1** The sequence-suffix utility value of α in a dataset D is an upper bound of the true utility of α in D. That is, \( su(\alpha, D) \leq SFU(\alpha, D) \).

The following theorem states that SFU is an upper bound on the utility of pattern β and any pattern prefixed with β, where β is produced by S-Step from α.

**Theorem 2** Given pattern α and sequence database D and item I, SFU(α, D) is an upper bound on:

1. The utility of pattern \( \beta = \alpha \otimes I \). That is, \( su(\beta, D) \leq SFU(\alpha, D) \).
2. The utility of any \( \beta \)'s offspring θ (i.e., any sequence prefixed with β). That is, \( su(\theta, D) \leq SFU(\alpha, D) \).

**Proof 2** Let \( \beta = \alpha \otimes I \) and \( S \in D \). We first prove \( su(\beta, S) \leq SFU(\alpha, S) \) for any sequence \( S \in D \). The conclusion can be easily extended to D. According to Definition 6, the utility of \( \beta \) can be rewritten as:

\[
su(\beta, S) = \max_{\delta \in OccSet(\beta, S)} \{ su(\alpha, \delta) + u(I, \delta) \}
\]

Assume that "I" occurs in transaction \( T_i \in \delta \) where \( \delta \) is the occurrence with the maximum utility of \( \beta \). We have \( su(\beta, S) \leq \max_{\delta \in OccSet(\beta, S)} \{ su(\alpha, \delta) + TU(T_i) \} \).

Since all occurrences of "I" are in suffix \( S, \alpha \), \( TU(T_i) \leq \sum_{T \in suffix(S,\alpha)} \).

Therefore:

\[
su(\beta, S) \leq \max_{\delta \in OccSet(\beta, S)} \{ su(\alpha, \delta) + \sum_{T \in suffix(S,\alpha)} TU(T) \}
\]

The second part is independent of \( \delta \). Thus, \( su(\beta, S) \leq \max_{\delta \in OccSet(\beta, S)} \{ su(\alpha, \delta) \} + \sum_{T \in suffix(S,\alpha)} TU(T) = SFU(\alpha, S) \).

Below we prove that the utility of any offspring of \( \beta \) is less than SFU(α, S). Assume that \( \theta = \alpha \otimes I \otimes ... \otimes IS \) where IS is the last itemset in \( \theta \) and \( \otimes \in \{ \otimes, \odot \} \). Let \( \delta_1 \) be the occurrence with maximum utility of \( \theta \) in S. The utility of \( \theta \) can be rewritten as follows:

\[
su(\theta, \delta_1) = su(\alpha, \delta_1) + \sum_{i \in \theta \otimes I \in suffix(S,\alpha)} u(i, \delta_1)
\]

Note that all items in \( \theta \) which are not in \( \alpha \) occur in suffix \( S, \alpha \). We know that \( su(\alpha, \delta_1) \leq su(\alpha, S) \). Hence:

\[
su(\theta, \delta_1) \leq su(\alpha, S) + \sum_{i \in \theta \otimes T \in suffix(S,\alpha)} u(i, T)
\]

Since the utility of each item in a transaction is no more than the utility of the transaction, \( su(\theta, \delta_1) \leq su(\alpha, S) + \sum_{T \in suffix(S,\alpha)} TU(T) = SFU(\alpha, S) \).

The conclusion can be easily extended from S to D.

**Pruning Strategy 2 (Pruning by SFU):** Let \( \alpha \) be the sequence represented by a node \( N \) in the HUSP-Tree and \( \delta \) be the minimum utility threshold. If \( SFU(\alpha, D) < \delta \), there is no need to generate S-nodes from \( N \). This is because the utility of \( \alpha \otimes I \) and any of \( \alpha \otimes I \)'s offspring is no more than \( SFU(\alpha, D) \), which is less than \( \delta \).

The pruning using SFU becomes more effective than TSWU when the length of the pattern increases. That is, it may prune more low utility patterns at each deeper level of the HUSP-Tree. This is due to the fact that overestimation using SFU decreases as
the length of the pattern increases. In other words, given a sequence \( \alpha \), to extend it using \textit{I-step} or \textit{S-step} and items in sequence \( S \), the items are added from the end of first occurrence of \( \alpha \) in \( S \). And those items in \( S \) within the first occurrence are unable to form a new extension of \( \alpha \). However, for a sequence \( \beta \) formed by an itemset or sequence extension, the utilities of those items are added to \( TSWU(\beta) \). For example in Table 1 \( SFU(\langle \{a\} \{b\} \{c\} \rangle),S_1) = 10 + 14 = 24 \) and \( TSWU(\langle \{a\} \{b\} \{c\} \rangle),S_1) = 15 + 8 + 7 + 14 = 44 \). In the evaluation section we will show the efficiency and effectiveness of using \( SFU \) in comparison to use of \( TSWU \) for pruning the HUSP-Tree.

4.1.1.5. HUSP-Tree Construction Algorithm Using the proposed pruning strategies, our tree construction process will generate only the nodes that represent potential HUSPs, defined as follows.

**Definition 17** A sequence \( \alpha \) is called potential high utility sequential pattern in \( D \) iff:

- If the node for \( \alpha \) is an I-node and \( TSWU(\alpha,D) \geq \delta \)
- If the node for \( \alpha \) is an S-node and \( SFU(\alpha,D) \geq \delta \)

The complete tree construction process is as follows. We first generate the child nodes of the root as described in Section 4.1.1.3. Then for each child node, the Tree-Growth algorithm (see Algorithm 1) is called to generate its I-nodes and S-nodes using the two pruning strategies and the I-Step and S-Step described in Section 4.1.1.3. Tree-Growth is a recursive function and it generates all potential HUSPs in a depth-first manner. Given the input node \( ND(\alpha) \), it first checks whether \( TSWU(\alpha) < \delta \). If yes, the node is pruned. Otherwise, it generates the I-nodes from \( ND(\alpha) \) using the I-Step (Lines 4-8) and recursively calls Algorithm 1 with each I-node. Then, the algorithm checks whether \( SFU(\alpha) \) satisfies the threshold \( \delta \). If yes, it generates the S-nodes of \( ND(\alpha) \) using the S-Step (Lines 11-15) and recursively calls the Algorithm 1 with each S-node.

The average time complexity for building HUSP-Tree with dataset \( D \) is \( O(\text{NumPot} \times \text{NumOcc}_{\text{avg}}) \), where \( \text{NumPot} \) is the number of potential high utility sequential patterns and \( \text{NumOcc}_{\text{avg}} \) is the average number of occurrences of a potential high utility sequential pattern. Note that \( \text{NumPot} \) depends on threshold \( \delta \).

4.1.2. HUSP Mining Phase

HUSP mining phase is straightforward. After performing the construction phase, HUSP-Tree maintains the information of the sequences in the database. When users request the mining results, the algorithm performs the mining phase by traversing the HUSP-Tree once. For each traversed node \( ND(\alpha) \), the algorithm uses the \textit{SeqUtilList} of \( ND(\alpha) \) to calculate the utility of \( \alpha \) in the database. If the utility of \( \alpha \) is no less than the minimum utility threshold, the algorithm outputs \( \alpha \) as a HUSP. After traversing the tree, all the HUSPs are returned.

4.2. HUSP-Stream: High Utility Sequential Pattern Mining over Data Streams

In the previous section, we proposed an efficient algorithm to find HUSPs in static databases. In this section, we propose an extended version of HUSP-Miner called HUSP-Stream to discover HUSPs over a transaction-sensitive sliding window. Below we first integrate the definitions presented in the previous section to the context of streaming data and then we describe HUSP-Stream.
Algorithm 1 TreeGrowth

Input: \( ND(\alpha) \): node representing sequence \( \alpha \)

Output: HUSP-Tree

1: if \( TSWU(\alpha, D) < \delta \) then
2: remove node \( ND(\alpha) \)
3: else
4: \( I_Set \leftarrow \) items in itemUtilList whose \( TSWU \geq \delta \) and whose id ranks lexicographically after the last item in the last itemset of \( \alpha \)
5: for each item \( \gamma \in I_Set \) do
6: Compute SeqUtilList(\( \alpha \oplus \gamma \)) using the I-Step
7: if SeqUtilList(\( \alpha \oplus \gamma \)) is not empty then
8: Create I-node \( ND(\alpha \oplus \gamma) \) as child of \( ND(\alpha) \)
9: Call Algorithm 1 (\( ND(\alpha \oplus \gamma) \))
10: end if
11: end for
12: if \( SFU(\alpha, D) \geq \delta \) then
13: \( S_Set \leftarrow \) items in itemUtilList whose \( TSWU \geq \delta \)
14: for each item \( \gamma \in S_Set \) do
15: Compute SeqUtilList(\( \alpha \otimes \gamma \)) using the S-Step
16: if SeqUtilList(\( \alpha \otimes \gamma \)) is not empty then
17: Create S-node \( ND(\alpha \otimes \gamma) \) as child of \( ND(\alpha) \)
18: Call Algorithm 1 (\( ND(\alpha \otimes \gamma) \))
19: end if
20: end for
21: end if
22: end if

4.2.1. Preliminaries and New Definitions

In many applications, such as web click stream analysis and customer purchase sequence mining, a new itemset may belong to an existing or a new sequence. Below, we define transaction-sensitive sliding window which not only considers new sequences, also a new element can be an item/itemset belonging to an existing sequence.

Definition 18 (Transaction data stream) A data stream of itemset-sequences (or Data stream in short) \( DS = \langle T_1, T_2, \ldots, T_M \rangle \) is an ordered list of transactions that arrive continuously in a time order. Each transaction \( T_i \in DS \) \((1 \leq i \leq M)\) belongs to a sequence of transactions. A data stream can thus also be considered as a set of dynamically-changing sequences.

Figure 5 shows a data stream \( DS = \langle S_1^1, S_2^1, S_3^1, S_1^2, S_2^2, S_3^2, S_1^3, S_2^3 \rangle \) with seven transactions, each belonging to one of three sequences: \( S_1, S_2 \) and \( S_3 \).

Definition 19 (Transaction-sensitive sliding window) Given a user-specified window size \( w \) and a data stream \( DS = \langle T_1, T_2, \ldots, T_M \rangle \), a transaction-sensitive sliding window \( SW \) captures the \( w \) most recent transactions in \( DS \). When a new transaction arrives, the oldest one is removed from \( SW \). The \( i \)-th window over \( DS \) is defined as \( SW_i = \langle T_i, T_{i+1}, \ldots, T_{i+w-1} \rangle \).

According to the definition, transactions in a sliding window can belong to different sequences. Thus, a sliding window is actually a sequence database that changes over
(a) A Data Stream of Itemset-Sequences

<table>
<thead>
<tr>
<th>SID</th>
<th>TID</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>(a,2)</td>
<td>(b,3)(c,2)</td>
<td>(b,1)(d,1)</td>
<td>(c,3)(d,1)</td>
<td>(b,1)(c,3)(d,2)</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>(b,4)</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>S3</td>
<td>(a,2)(b,5)(e,2)</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(b) Profit Table

<table>
<thead>
<tr>
<th>Item</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5. An example of a data stream of itemset-sequences

For example, in Figure 5, if the window size $w$ is set to 5, the first and the second windows over $DS$ are $SW_1 = \langle S_1^1, S_1^2, S_1^3, S_1^4, S_1^5 \rangle$ (which has two sequences) and $SW_2 = \langle S_2^1, S_2^2, S_2^3, S_2^4, S_2^5 \rangle$ (which has three sequences), respectively.

Definition 20 (Utility of a sequence in a sliding window) The utility of a sequence $\alpha$ in the $i$-th sliding window $SW_i$ over $DS$ is defined as $su(\alpha, SW_i) = \sum_{T \in S} su(\alpha, T)$.

Definition 21 (High utility sequential pattern (HUSP) over sliding window $SW_i$) A sequence $\alpha$ is called a high utility sequential pattern (HUSP) in a sliding window $SW_i$ iff $su(\alpha, SW_i)$ is no less than a user-specified minimum utility threshold $\delta$.

For example, let $\alpha = \langle \{ab\} \{c\} \rangle$. In $SW_1$ of Figure 5, $OccSet(\alpha, S_1) = \{ \langle S_1^1, S_1^2 \rangle, \langle S_1^1, S_1^3 \rangle \}$. The utility of $\alpha$ in $S_1$ is $su(\alpha, S_1) = \max \{ su(\alpha, \langle S_1^1, S_1^2 \rangle), su(\alpha, \langle S_1^1, S_1^3 \rangle) \}$ = $\max \{14, 16\} = 16$. The utility of $\alpha$ in $SW_1$ is $su(\langle \{ab\} \{c\} \rangle, SW_1) = su(\alpha, S_1) + su(\alpha, S_2) = 16 + 0 = 16$.

Definition 22 The Transaction based Sequence-Weighted Utility (TSWU) of a sequence $\alpha$ in the $i$-th transaction-sensitive window $SW_i$, denoted as $TSWU(\alpha, SW_i)$, is defined as the sum of the utilities of all the transactions in all the sequences containing $\alpha$ in $SW_i$: $TSWU(\alpha, SW_i) = \sum_{S \in SW_i} \sum_{\alpha \subseteq S} TU(T)$

where $TU(T)$ is the utility of transaction $T$, and $\alpha \subseteq S$ means $\alpha$ is a subsequence of $S$.

Definition 23 The SFU of a sequence $\alpha$ in the $i$-th window $SW_i$, denoted as $SFU(\alpha, SW_i)$, is defined as follows: $SFU(\alpha, SW_i) = \sum_{S \in SW_i} SFU(\alpha, S)$.

In this work, $w$ is defined as the number of transactions. However it could be defined as a period of time, which means that a window is formed by all the transactions arrived in a time interval.
Algorithm 2 HUSP-Stream

Input: A new transaction \( S_i \), window size \( w \), minimum utility threshold \( \delta \), ItemUtilLists, HUSP-Tree

Output: Updated ItemUtilLists, updated HUSP-Tree, HUSPs

1: if \( i \leq w \) (when \( S_i \) is a transaction in the first window) then
2: \( \forall \text{ item } \in S_i, \text{ put}(r,i,u(\text{item},S_i)) \) to ItemUtilLists(\text{item})
3: if \( i = w \) then
4: Construct HUSP-Tree using ItemUtilLists and \( \delta \)
5: \end if
6: else
7: Update ItemUtilLists and HUSP-Tree using \( S_i \), \( w \) and \( \delta \)
8: \end if
9: if the user requests to get HUSPs for the current window then
10: Find all the HUSPs from the potential HUSPs stored in HUSP-Tree using \( \delta \)
11: \end if
12: Return Updated ItemUtilLists, updated HUSP-Tree, HUSPs if requested

Property 2 The sequence-suffix utility value of \( \alpha \) in a sliding window \( SW_i \) is an upper bound of the true utility of \( \alpha \) in \( SW_i \). That is, \( su(\alpha,SW_i) \leq SFU(\alpha,SW_i) \).

Definition 24 A sequence \( \alpha \) is called a potential high utility sequential pattern in sliding window \( SW_i \) iff:

- If the node for \( \alpha \) is an I-node and \( TSWU(\alpha,SW_i) \geq \delta \)
- If the node for \( \alpha \) is an S-node and \( SFU(\alpha,SW_i) \geq \delta \)

Problem statement 2. Given a minimum utility threshold \( \delta \) and a window size \( w \), the problem of mining high utility sequential patterns over a data stream \( DS \) of transactions is to discover all sequences of itemsets whose utility is no less than \( \delta \) from the current sliding window over \( DS \).

It is very inefficient to apply existing algorithms (e.g., \([2,3,21,24]\)) designed for static databases to rerun the whole mining process on updated databases whenever a record is added to or deleted from the database. A promising solution to this problem is to design an in-memory data structure to maintain the essential information of HUSPs in \( SW_i \). When the window slides from \( SW_i \) to \( SW_{i+1} \), we incrementally update this structure to reflect the current HUSPs in \( SW_{i+1} \). Our method is proposed based on this concept and presented in the next section.

4.2.2. An Overview

The overview of HUSP-Stream is presented in Algorithm 2. The algorithm includes three main phases: (1) initialization phase, (2) update phase and (3) HUSP mining phase. The initialization phase is the same as the construction phase used by HUSP-Miner. The difference is that, the initialization phase is applied when the input transaction belongs to the first sliding window (i.e., when \( i \leq w \)). In the initialization phase (lines 1-5), the ItemUtilLists structure is constructed for storing the utility information for every item in the input transaction \( S_i \). When there are \( w \) transactions in the first window (i.e., when \( i = w \)), HUSP-Tree is constructed for the first window. If there are already \( w \) transactions in the window when the new transaction \( S_i \) arrives, \( S_i \) is added to the window and the oldest
transaction in the window is removed. This is done by incrementally updating the ItemUtilLists and HUSP-Tree structures on line 7, which is the update phase of the algorithm. The updated HUSP-Tree contains all the potential high utility sequential patterns for the new window, which makes finding HUSPs from the new window an easy job. After the updating phase, if the user requests to find HUSPs from the new window, HUSP-Stream finds all the HUSPs from the potential HUSPs stored in HUSP-Tree.

### 4.2.3. Initialization phase

In this phase, HUSP-Stream reads the transactions in the first sliding window one by one to construct ItemUtilLists and HUSP-Tree. It is done similar to the construction phase in HUSP-Miner. The average time complexity for building HUSP-Tree for the first sliding window is $O(NumPot \times NumOcc_{avg})$, where $NumPot$ is the number of potential high utility sequential patterns and $NumOcc_{avg}$ is the average number of occurrences of a potential high utility sequential pattern.

### 4.2.4. Update Phase

When a new transaction $S_i'$ arrives, if the current window $SW_i$ is full, the oldest transaction $S_i$ expires. In this scenario, the algorithm needs to incrementally update ItemUtilLists and HUSP-Tree to find the HUSPs in $SW_{i+1}$. Below, we first perform step-by-step analysis and then develop the algorithm for the update phase.

Let $H^+$ be the complete set of HUSPs in the current sliding window $SW_i$, $H^-$ be the complete set of HUSPs after a transaction removed from or added to $SW_i$, $D^+$ be the window after transaction $S_i'$ is added to $SW_i$, $D^-$ be the window after $S_i'$ is removed from $SW_i$ and $S$ be a pattern found in $SW_i$. The following lemmas state how utility of $S$ changes when a transaction is added to or removed from the window.

**Lemma 1** Given non-empty sequence $S$, after $S_i'$ is added to the window, one of the following cases is held:

1. If $S \leq S_i$ and $S \in H^-$, then $S \in H^-$ and $su(S, D^+) \geq su(S, SW_i)$.
2. If $S \leq S_i$ and $S \notin H^-$, then $su(S, D^+) \geq su(S, SW_i)$.
3. If $S \notin S_i$ and $S \in H^-$, then $S \in H^-$ and $su(S, D^+) = su(S, SW_i)$.
4. If $S \notin S_i$ and $S \notin H^-$, then $S \notin H^-$ and $su(S, D^+) = su(S, SW_i)$.

**Proof 3** Let $S_i'$ be sequence $S_i$ before transaction $S_i'$ is appended to, $OSet_{SW_i}$ be the set of occurrences of $S$ in $SW_i$ and $OSet_{SW_{i+1}}$ be the set of occurrences of $S$ in $SW_{i+1}$. Below, we prove each case separately:

1. Since $S \in H$, according to Definition 21, $su(S, SW_i) \geq \delta$. Also, $S \geq S_i$, hence $OSet_{SW_i} \subseteq OSet_{SW_{i+1}}$. In this case there is $o' \in OSet_{SW_{i+1}}$ where $o' \notin OSet_{SW_i}$. If $su(S, o') > su(S, S_i')$ then $su(S, SW_{i+1}) > su(S, SW_i)$. Otherwise, $su(S, SW_{i+1}) = su(S, SW_i)$. In both cases, since $su(S, SW_i) \geq \delta$ then $su(S, SW_{i+1}) \geq \delta$ and $S \in H^+$.

2. Since $S \geq S_i$, then $OSet_{SW_i} \subseteq OSet_{SW_{i+1}}$. In this case there is an occurrence $o' \in OSet_{SW_{i+1}}$ where $o' \notin OSet_{SW_i}$. Also, $S \notin H$, according to Definition 21, $su(S, SW_i) < \delta$. If $su(S, o') > su(S, S_i')$ then $su(S, SW_{i+1}) > su(S, SW_i)$. Otherwise, $su(S, SW_{i+1}) = su(S, SW_i)$.

3. Since $S \neq S_i$, then $OSet_{SW_i} = OSet_{SW_{i+1}}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{SW_{i+1}})$. Also, $S \in H$, according to Definition 21, $su(S, SW_i) \geq \delta$. Since the utility of $S$ is the same, $S \in H^+$. 

Deletion Operation
ItemUtilList Update
Incrementally update TSWU(β), SFU(β)
Addition Operation
Incrementally update TSWU(γ)
Update first level of HUSP-Tree using PSet
Perform HUSP-Tree Update

Figure 6. An overview of ItemUtilLists update step

(4) Since \( S \not\in S_c \), then \( OSet_{SW} = OSet_{SW_i} \). In this case \( su(S, OSet_{SW}) = su(S, OSet_{SW_{i+1}}) \). Also, \( S \not\in H \), according to Definition 21, \( su(S, SW_i) < δ \). Consequently, \( su(S, SW_{i+1}) < δ \) so \( S \not\in H^+ \).

Let \( S'^i_c \) be the sequence before transaction \( S'^i_c \) is removed from sequence \( S_c \). We have the following lemma.

**Lemma 2** Given non-empty sequence \( S \), after \( S'^i_c \) is removed from the window, one of the following cases is held:

1. If \( S \subseteq S'^i_c \) and \( S \in H^+ \), then \( su(S, D^-) \leq su(S, SW_i) \).
2. If \( S \subseteq S'^i_c \) and \( S \not\in H^+ \), then \( su(S, SW_i) < δ \). Consequently, \( su(S, SW_{i+1}) < δ \) so \( S \not\in H^+ \).
3. If \( S \not\subseteq S'^i_c \) and \( S \not\in H^+ \), then \( su(S, SW_i) < δ \).
4. If \( S \not\subseteq S'^i_c \) and \( S \not\in H^+ \), then \( su(S, SW_{i+1}) < δ \).

**Proof 4** Let \( OSet_{SW} \) be the set of occurrences of \( S \) in \( SW_i \) and \( OSet_{SW_{i+1}} \) be the set of occurrences of \( S \) in \( SW_{i+1} \):

1. Since \( S \in H \), according to Definition 21, \( su(S, SW_i) \geq δ \). Also, since \( S \subseteq S'^i_c \) and \( S \not\subseteq S'^i_c \), hence \( OSet_{SW_{i+1}} \subseteq OSet_{SW} \). In this case there is \( o' \in OSet_{SW} \) where \( o' \not\in OSet_{SW_{i+1}} \). If \( su(S, o') > su(S, S_c) \) then \( su(S, SW_{i+1}) < su(S, SW_i) \). Otherwise, \( su(S, SW_{i+1}) = su(S, SW_i) \).
2. Since \( S \subseteq S'^i_c \) and \( S \not\subseteq S'^i_c \), hence \( OSet_{SW_{i+1}} \subseteq OSet_{SW} \). In this case there is \( o' \in OSet_{SW} \) where \( o' \not\in OSet_{SW_{i+1}} \). Also, \( S \not\in H \), according to Definition 21, \( su(S, SW_i) \geq δ \). Consequently, \( su(S, SW_{i+1}) < su(S, SW_i) \). Otherwise, \( su(S, SW_{i+1}) = su(S, SW_i) \).
3. Since \( S \not\subseteq S'^i_c \), hence \( OSet_{SW_{i+1}} = OSet_{SW} \). In this case \( su(S, OSet_{SW}) = su(S, OSet_{SW_{i+1}}) \).
4. Since \( S \not\subseteq S'^i_c \), hence \( OSet_{SW} = OSet_{SW_{i+1}} \). In this case \( su(S, OSet_{SW}) = su(S, OSet_{SW_{i+1}}) \).

Also, \( S \in H \), according to Definition 21, \( su(S, SW_i) \geq δ \). Since the utility of \( S \) is the same, \( S \in H^+ \).

Also, \( S \not\in H \), according to Definition 21, \( su(S, SW_i) < δ \). Consequently, \( su(S, SW_{i+1}) < δ \) so \( S \not\in H^+ \).
Below we propose an efficient approach to update ItemUtilLists and HUSP-Tree based on Lemma 1 and Lemma 2.

Figure 6 shows the update ItemUtilLists step. For each item $\beta$ in the oldest transaction $S_d$, the algorithm removes each tuple $T_p$ whose SID and TID are $c$ and $d$ respectively from ItemUtilLists($\beta$) (i.e., Deletion operation in Figure 6). Then, TSWU($\beta$) and SFU($\beta$) are updated accordingly. The next operation is addition operation, which is performed as follows. For each item $\gamma$ in the new transaction $S_u$, the algorithm inserts new tuple $\langle S_v, T_u, u(\gamma, S_u) \rangle$ to ItemUtilLists($\gamma$). Once, TSWU($\gamma$) is updated, all the promising items (i.e., the items whose TSWU is no less than the utility threshold) are collected into an ordered set $pSet$. For each item $\gamma$ in $pSet$, if ND($\gamma$) is already under the root and its SeqUtilList has not been updated, the algorithm replaces the old SeqUtilList by the updated ItemUtilLists of item $\gamma$. If ND($\gamma$) has not been created under the root, the algorithm creates it under the root. All the unpromising nodes (i.e., the nodes whose TSWU is less than the utility threshold) in the first level of HUSP-Tree are removed from the tree.

After ItemUtilLists update step, HUSP-Tree update step is invoked. Figure 7 shows an overview of HUSP-Tree update step. For each child node ND($\alpha$) under the root, the algorithm calls the procedure UpdateTree(ND($\alpha$)) to update the sub-tree of ND($\alpha$), which is performed as follows. For each child node ND($\beta$) where $\beta$ is $\alpha \ominus \gamma$ or $\alpha \ominus \gamma$ and $\gamma \in pSet$, the algorithm checks whether ND($\beta$) is already in the current HUSP-Tree. If ND($\beta$) is not in the HUSP-Tree, the algorithm constructs $\beta$’s SeqUtilList using I-Step or S-Step and creates ND($\beta$) under ND($\alpha$). If ND($\beta$) is already in the HUSP-Tree, the algorithm incrementally updates the tuples in SeqUtilList($\beta$) related to the new and oldest transactions as follows. Given the oldest transaction $S_d^c$ and the newest transaction $S_u^v$, according to Lemma 1 and Lemma 2, the SeqUtilList($\beta$) should be updated if it has a tuple whose SID is either $S_c$ or $S_v$. These tuples (not all the tuples in SeqUtilList($\beta$)) are re-
constructed by applying I-Step (if $\beta$ is $\alpha \oplus \gamma$) or S-Step (if $\beta$ is $\alpha \otimes \gamma$) on $\text{SeqUtilList}(\alpha)$ and $\text{itemUtilLists}(\gamma)$. Then the algorithm updates $\text{TSWU}$ of $\beta$ based on the updated $\text{SeqUtilList}(\beta)$. If $\text{TSWU}$ of $\beta$ is less than the utility threshold, the algorithm removes $\text{ND}(\beta)$ and the sub-tree under $\text{ND}(\beta)$. Otherwise, if $\beta$ is $\alpha \oplus \gamma$, the algorithm calls the procedure $\text{UpdateTree}(\text{ND}(\beta))$ to update the sub-tree of $\text{ND}(\beta)$. If $\beta$ is $\alpha \otimes \gamma$, the $\text{SFU}$ of $\beta$ is updated using the updated $\text{SeqUtilList}(\beta)$. If $\text{SFU}$ of $\beta$ is less than the threshold, node $\text{ND}(\beta)$ and its subtree are removed from the tree; otherwise, it recursively calls $\text{UpdateTree}(\text{ND}(\beta))$.

For example, Figure 8 shows the updated $\text{ItemUtilLists}$ and $\text{SeqUtilList}([\text{ab}])$ when $T_1$ is removed from and $T_6$ is added to the window. Note that we do not reconstruct the whole $\text{SeqUtilList}([\text{ab}])$. Since $T_1$ belongs to $S_1$, we only need to update/ remove the first tuple and also add a new tuple for the new sequence $S_3$. The other tuples are not updated. In this figure, since $[\text{ab}]$ is not in $S_1$ any more but exists in $S_3$, $\text{SeqUtilList}([\text{ab}])$ is updated as $\text{SeqUtilList}([\text{ab}]) = \{[S_2, T_3, 23], [S_3, T_6, 19]\}$.

Since a tuple in $\text{ItemUtilLists}$ can be accessed directly and the number of tuples needed to be updated in $\text{ItemUtilLists}$ is $L_{\text{oldest}} + L_{\text{new}}$, where $L_{\text{oldest}}$ is the length of the transaction to be removed and $L_{\text{new}}$ is the length of the new transaction added to the sliding window, the average time complexity for updating $\text{ItemUtilLists}$ is $O(L_{\text{avg}})$, where $L_{\text{avg}}$ is the average length of transactions in the data stream. The average time complexity for updating $\text{HUSP-Tree}$ is $O(\text{NumPot} \times \text{NumOccAff}_{\text{avg}})$ where $\text{NumPot}$ is the number of potential high utility sequential patterns in the new sliding window, and $\text{NumOccAff}_{\text{avg}}$ is the average number of occurrences of a potential high utility sequential pattern in the sequences affected by the removal of the oldest transaction and the addition of the new transaction.

4.2.5. Mining Phase

The mining phase can be done similar to the mining phase proposed by $\text{HUSP-Miner}$. However, $\text{HUSP}$ mining phase can be combined with the update phase as well. That is, during $\text{HUSP-Tree update}$, the utility of the sequence represented by each node can be computed. If the utility is no less than the threshold, the sequence can be returned as a $\text{HUSP}$ during the update phase.
4.3. Correctness

In this section, we show the correctness of our proposed methods. We first prove that, HUSP-Miner does not miss any HUSP.

**Theorem 3** Given a sequence database \( D \) and minimum utility threshold \( \delta \), if pattern \( \alpha \) is a high utility sequential pattern over \( D \) then there is a node \( N \) in HUSP-Tree which represents \( \alpha \).

**Proof 5** We prove the theorem by contradiction. Assume that the pattern \( S \) is a HUSP but it does not exist in the tree. According to Definition 8 since \( S \) is a HUSP then \( su(S, D) \geq \delta \).

On the other hand, HUSP-Miner does not create a node for a pattern in HUSP-Tree only in two main cases:

1. If TSWU of the pattern formed by I-Step is less than \( \delta \): according to Theorem 1, for all the patterns \( \beta \) where \( \alpha \preceq \beta \), \( TSWU(\alpha, D) \geq TSWU(\beta, D) \) and also \( su(\alpha, D) \leq TSWU(\alpha, D) \) and \( \forall \beta \preceq \alpha, su(\beta, D) \leq TSWU(\alpha, D) \). If HUSP-Stream does not create a node for the pattern formed by I-Step, it means that \( TSWU(\alpha, D) \leq \delta \). Hence \( su(\alpha, D) \leq \delta \) which is a contradiction.

2. If SFU of the pattern formed by S-Step is less than \( \delta \): according to Property 2 and Theorem 2, \( su(\alpha, D) \leq SFU(\alpha, D) \). If HUSP-Stream does not create a node for the pattern formed by S-Step, it means that \( SFU(\alpha, D) \leq \delta \). Hence \( su(\alpha, D) \leq \delta \) which is a contradiction.

Considering Lemma 1 and Lemma 2 and the proposed update procedure, below we prove that HUSP-Stream keeps all the HUSPs in HUSP-Tree for the current sliding window.

**Theorem 4** Given a sliding window \( SW_i \) and minimum utility threshold \( \delta \), if pattern \( \alpha \) is a high utility sequential pattern over \( SW_i \) then there is a node \( N \) in HUSP-Tree which represents \( \alpha \).

**Proof 6** Below we show that our method for finding HUSPs (i.e., Algorithm 2) does not miss any HUSPs. There are two places where we prune the search space in Algorithm 2.

- **Initialization Phase:** when \( SW_i \) is the first window (i.e., \( i = 1 \)), HUSP-Stream constructs the data structures in the initialization phase. As mentioned before, initialization phase is similar to the construction phase in HUSP-Miner. Therefore, according to Theorem 3 no HUSPs will be missed.

- **Update Phase:** If \( SW_i \) is not the first window (i.e., \( i \geq 1 \)), given the oldest transaction \( S^d_i \) and the newest transaction \( S^u_i \), when transaction \( S^d_i \) leaves from \( SW_{i-1} \) and transaction \( S^u_i \) is added to the window and \( SW_i \) forms, HUSP-Stream updates the data structures accordingly (i.e., update phase). Let \( \delta \) be the minimum utility threshold, \( P_{SW} \) be the set of all the potential HUSPs stored in HUSP-Tree in \( SW_i \), \( H_{SW} \) be the set of all the HUSPs in \( SW_i \) and node \( \alpha \in HUSP-Tree \) represents \( \alpha \). According to Definitions 21 and 24, we conclude that \( H_{SW} \subseteq P_{SW} \). Considering Definition 24, below we prove that HUSP-Tree does not miss any PHUSPs after window slides from \( SW_{i-1} \) to \( SW_i \). There are two cases:
Table 2. Parameters of IBM data generator

<table>
<thead>
<tr>
<th>D</th>
<th>Number of sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Average number of transactions in a sequence</td>
</tr>
<tr>
<td>T</td>
<td>Average number of items in a transaction</td>
</tr>
<tr>
<td>S</td>
<td>Average number of itemsets in a potential maximal sequential pattern</td>
</tr>
<tr>
<td>I</td>
<td>Average number of items in an itemset of a potential maximal sequential pattern</td>
</tr>
<tr>
<td>N</td>
<td>Number of distinct items</td>
</tr>
</tbody>
</table>

1. \( \alpha \notin P_{SW_i} \) and \( \alpha \in P_{SW_i} \): since \( \alpha \) does not exist in \( SW_i \), HUSP-Tree does not contain \( N_\alpha \). Hence, removing \( S^c \) does not affect \( N_\alpha \). However, when \( S^c \) is added to the window, according to the proposed update procedure, SeqUtilList(\( \alpha \)) is formed by using I-Step and S-Step based on the updated itemUtilLists. Therefore, SeqUtilList(\( \alpha \)) contains all the occurrences of \( \alpha \) in \( SW \), and the node will be added if \( \alpha \) is a PHUSP according to Definition 24.

2. \( \alpha \in P_{SW_i} \) and \( \alpha \in P_{SW_{i-1}} \): according to the update phase when \( S^d \) leaves from the window, it affects all the records in SeqUtilList whose SID is \( c \). In the update phase all the occurrences of \( \alpha \) in the updated sequence \( S_c \) (i.e., after \( S^d \) is removed) are found. Hence, all the occurrences affected by \( S^d \) are eliminated from SeqUtilList(\( \alpha \)). Similarly, when \( S^u \) arrives, all the occurrences of \( \alpha \) in the updated \( S^u \) (i.e., after \( S^d \) is added) are found and inserted to SeqUtilList(\( \alpha \)). Consequently, the updated SeqUtilList(\( \alpha \)) covers all the occurrences of \( \alpha \) in \( SW \). Hence, the value calculated by SFU (if \( N_\alpha \) is S-node) or TSWU (if \( N_\alpha \) is S-node) is correct and no PHUSPs will be missed.

5. Experiments

In this section, we evaluate the performance of the proposed methods. The experiments were conducted on an Intel(R) Core(TM) i7 3.20 GHz computer with 32 GB of RAM. Both synthetic and real datasets are used in the experiments. Two synthetic datasets DS1:D10K-C10-T3-S4-I2-N10K and DS2:D100K-C8-T3-S4-I2-N1K were generated by the IBM data generator [1]. The definition of parameters used by the IBM data generator are shown in the Table 2. Chainstore is a real-life dataset acquired from [20], which already contains internal and external utilities. In order to use this dataset as a sequential dataset, we grouped transactions in different sizes so that each group represents a sequence of transactions. Another real dataset BMS is obtained from SPMF [11] which contains 3,340 distinct items and consists of 77,512 sequences of clickstream data from an e-retailer. We follow previous studies [2] to generate internal and external utility of items. 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 3 shows characteristics of the datasets and parameter settings in the experiments. The Window Size column of Table 3 shows the default window size for each dataset. We will later change the window size to show the performance of the algorithms under different window sizes.

We use the following measures to evaluate the performance of the algorithms:
Table 3. Details of parameter setting

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Seq</th>
<th>#Trans</th>
<th># Items</th>
<th>Window Size (w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>10K</td>
<td>100K</td>
<td>10,000</td>
<td>50K</td>
</tr>
<tr>
<td>BMS</td>
<td>77K</td>
<td>120K</td>
<td>3,340</td>
<td>60K</td>
</tr>
<tr>
<td>DS2</td>
<td>100K</td>
<td>800K</td>
<td>10,000</td>
<td>400K</td>
</tr>
<tr>
<td>ChainStore</td>
<td>400K</td>
<td>1000K</td>
<td>46,086</td>
<td>500K</td>
</tr>
</tbody>
</table>

- **Number of potential high utility sequential patterns (#PHUSP):** the total number of potential high utility sequential patterns produced by the algorithm in all sliding windows.
- **Total execution time (sec.):** the total execution time of the algorithms.
- **Sliding Time (sec.):** the average execution time of the algorithms to update data structures when a transaction arrives to or leaves from the window.
- **Memory Usage (MB):** the average memory consumption per window.

For consistency across datasets, the minimum threshold is shown as a percentage of the total utility of all the sequences in a dataset.

5.1. Methods in Comparison

We compare **HUSP-Miner** to **USpan** [26] which is the current best algorithm for mining high utility sequential patterns in static databases. To the best of our knowledge, no study has been proposed for mining high utility sequential patterns over data streams. Hence, we compare **HUSP-Stream** with two different versions of **USpan** [26]. Since **USpan** is not applicable to data streams, we design the following approaches to apply **USpan** over data streams: (1) We execute **USpan** on each sliding window individually, and collect the aggregated results for the performance evaluation. We call this approach **USpan_Trans** since it is ran when the window slides. The datasets used in the experiments are quite large and the window slides a large number of times, so the first approach runs very slowly. (2) To reduce the execution time of **USpan**, we modified **USpan** so that we run it per set of transactions (i.e., per batch). Once the number of incoming transactions equals to a given input parameter, **USpan** is ran to find **HUSPs**. This approach is called **USpan_Batch**. We set the size of each batch to 0.01% of whole transactions in dataset. In the next section, we investigate the efficiency of the proposed method and two versions of **USpan** in terms of update processing.

Moreover, in order to see the effect of using **SFU** to prune the tree in comparison to the other pruning strategy, **TSWU**, we implemented a basic version of **HUSP-Stream** in the experiments, called **HUSP_TSWU** which applies the **TSWU** pruning strategy for pruning I-nodes and S-nodes.

5.2. Performance Evaluation of HUSP-Miner

In this section, we show the performance of **HUSP-Miner** in comparison to **USpan** in terms of number of potential **HUSPs**, execution time and memory usage.
5.2.1. Number of Potential HUSPs

Figure 9 shows the number of PHUSPs on different datasets under varied minimum utility thresholds. As shown in this figure, HUSP-Miner generates fewer potential HUSPs than USpan. This shows the effectiveness of the proposed pruning strategies in this paper. By applying SFU, more unpromising patterns are discarded from the tree during the construction phase. In Figure 9, when the minimum utility threshold is less than 0.07% in DS1 and ChainStore datasets and 0.014% in BMS and DS2 datasets, the number of PHUSP generated by HUSP-Miner becomes smaller than HUSP-Stream clearly.

5.2.2. Time Efficiency of HUSP-Miner

Figure 10 shows the execution time of each method to find HUSPs with different minimum utility threshold values. For all the datasets, the execution time increases as the minimum utility threshold decreases. This is due to fact that, with smaller threshold values, the search space is more deeper and wider which results in producing more patterns. It can be observed that for all the datasets, HUSP-Miner runs faster than USpan. Especially for the lower minimum utility threshold values, HUSP-Miner outperforms USpan significantly, since HUSP-Miner applies SFU to prune candidates which results in a larger decrease on the execution time. On the DS1 and BMS datasets, the execution time of USpan is a bit worse than HUSP-Miner. On the DS2 and ChainStore datasets, HUSP-Miner is significantly faster than USpan.
Figure 10. Execution time of HUSP-Miner and USpan

Figure 11. Memory Usage of HUSP-Miner and USpan
5.2.3. Memory Usage

In this section, we report the memory usage taken by each method to find HUSPs. Figure 11 reports the memory consumption on the four datasets. It can be observed that for all of the datasets, USpan consumes more memory than HUSP-Miner. This is due to the large amount of memory that it needs to keep the larger search tree generated during the mining process. It is caused by the fact that USpan uses different pruning strategies which are not as effective as the pruning strategies used by HUSP-Miner.

5.3. Performance Evaluation of HUSP-Stream

In the previous section, we showed the performance of HUSP-Miner in comparison to the state-of-the-art HUSP mining algorithm. In this section, we present the effectiveness and efficiency of HUSP-Stream.

5.3.1. Performance evaluation for sliding two hundred consecutive windows

In this section, we investigate the efficiency of the update phase of the method. Since USpan is not designed for data stream environment, it is not able to return results over the whole dataset due to memory problem and long time processing. In order to compare, after the first window is processed, we only added a small portion of transactions (i.e. 200 transactions) to show the performance of the methods. Figure 12(a), Figure 12(b) and Figure 12(c) show the execution run time for the three methods on DS1, BMS and DS2 respectively. USpan_Trans is the slowest method because it is ran per transaction. USpan_Batch works more efficient than USpan_Trans, since it updates data structures.
Figure 13. Number of PHUSPs on different datasets

and results per set of transactions. However, HUSP-Stream outperforms both methods significantly. For example in DS2, HUSP-Stream is 20 times faster than USpan_Trans and around 8 times faster than USpan_Batch. We will later show the results of execution time over the whole datasets.

The second measure is average window sliding time. Figure 12(d), Figure 12(e) and Figure 12(f) show the results on DS1, BMS and DS2 respectively. For the dataset DS1, the average window sliding time of HUSP-Stream is more than 20 times faster than that of USpan_Trans and 10 times faster than that of USpan_Batch. For BMS, this ratio is 40 times and for the largest dataset DS2, HUSP-Stream is 500 times faster than USpan_Trans. As the figures presented, USpan_Trans is very inefficient even for a small number of updates. Hereafter, we do not report USpan_Trans as a compared method since the datasets are really large and it can not return results due to memory problem and long time processing.

5.3.2. Number of Potential HUSPs

In this section, we evaluate the algorithms in terms of the number of potential high utility sequential patterns (PHUSPs) produced by the algorithms. Figure 13 shows the results under different utility thresholds. As shown in Figure 13, HUSP-Stream produces much fewer PHUSPs than USpan_Batch. For example, on DS1, when the threshold is 0.06%, the number of PHUSPs generated by USpan_Batch is 10 times more than that generated by HUSP-Stream. On the larger datasets, i.e., BMS, DS2 and ChainStore, the number of PHUSPs grows quickly when the threshold decreases. For example, on BMS, when the threshold is 0.02%, the number of PHUSPs produced by USpan_Batch is 14 times larger than that generated by HUSP-Stream. The main reason why our approach produces much fewer candidates is that HUSP-Stream incrementally updates HUSP-Tree by reusing the
previous mining results. Hence, it avoids regenerating a large number of intermediate PHUSPs during the mining process. Another reason is that our pruning strategies are more effective than the ones used in USpan\_Batch.

5.3.3. Time Efficiency of HUSP-Stream

Figure 14(a), Figure 14(b), Figure 14(c) and Figure 14(d) show the total execution time of the algorithms on each of the four datasets with different minimum utility threshold. As it is shown in the figure, HUSP-Stream is much faster than USpan\_Batch. For example, HUSP-Stream runs 5 times faster on the BMS dataset and more than 10 times faster than USpan\_Batch on DS2. A reason is that USpan\_Batch re-runs the whole mining process, while HUSP-Stream performs incremental mining on each new window by efficiently updating its data structures. For example, the average execution time of HUSP-Stream on DS1 is 350 seconds, while that of USpan\_Batch on the same dataset is close to 1,200 seconds. On the BMS dataset, HUSP-Stream runs faster than USpan\_Batch by 5 times. Besides, it can be observed that HUSP-Stream is very scalable. Even under the low threshold values, it can perform well. From this experiment, the fewer number of PHUSPs of HUSP-Stream is another reason that HUSP-Stream is faster than USpan\_Batch.

We also evaluate the average window sliding time of the algorithms under different minimum utility thresholds. Figure 15 shows the average window sliding time of the algorithms on DS1, BMS, DS2 and ChainStore. For the datasets DS1 and BMS, the average window sliding time of our algorithm is below 1 second, which is 10 times faster than that of USpan\_Batch. For the largest dataset DS2, when the threshold is set to 0.06%, HUSP-Stream only spends 2.2 second, while USpan\_Batch sends more that 160 seconds. In this case, HUSP-Stream is 100 times faster than the USpan\_Batch. For the largest dataset ChainStore, when the threshold is set to 0.04%, HUSP-Stream only spends 1.1
second, while USpan_Batch sends more than 260 seconds. In this case, HUSP-Stream is 200 times faster than the USpan_Batch.

5.3.4. Memory Usage

In this section, we evaluate the memory usage of the algorithms under different utility thresholds. The results are shown in Figure 16, which indicate our approach consumes less memory than USpan_Batch. For example, for the dataset DS2, when the threshold is 0.06%, the memory consumption of HUSP-Stream is around 300 MB, while that of USpan_Batch is over 4,000 MB. A reason is that USpan_Batch produces too many PHUSPs during the mining process, which causes USpan_Batch having a larger tree than that of HUSP-Stream.

5.3.5. Performance Evaluation with Window Size Variation

In this section, we evaluate the performance of the algorithms under different window sizes. In this experiment, the minimum utility threshold is set to 0.09%, 0.03%, 0.09%, 0.04% for the datasets DS1, BMS, DS2 and ChainStore, respectively. The results are shown in Figure 17. In Figure 17(a), each bar shows the memory consumption of HUSP-Stream on a dataset under a window size. For example, the most left bar is the memory consumption of HUSP-Stream on DS1 when the window size is set to 2,000 transactions. From Figure 17(a), we observe that the memory consumption of HUSP-Stream increases very slowly with increasing window sizes. Figure 17(b) shows the execution time of HUSP-Stream under different window sizes. We also see that HUSP-Stream is also scalable in time with increasing window sizes.
5.3.6. Effectiveness of SFU Pruning

In this section, we evaluate the use of SFU (in comparison to the use of only TSWU) for pruning the tree. To show effectiveness of the proposed pruning strategy, \textit{HUSP-Stream} is compared to its basic version, \textit{HUSP-TSWU}, which only applies the TSWU pruning strategy for pruning \textit{I-nodes} and \textit{S-nodes}.

Figure 18, Figure 19 and Figure 20 illustrate the run time, the number of potential \textit{HUSPs} generated by the two methods, and their memory usage under different utility threshold values. These figures show that our new pruning strategy is more effective than using only TSWU in terms of all the three performance measures. Moreover, the figures show that the differences between the two pruning strategies in the number of \textit{PHUSPs},
Figure 18. Impact of SFU on Run Time

Figure 19. Impact of SFU on Number of PHUSPs
run time and memory usage increases in general when the utility threshold decreases.

5.3.7. Scalability

To further evaluate the scalability of HUSP-Stream, we generate a number of subsets of the DS1, BMS, DS2 and ChainStore datasets. The size of a subset ranges from 50% to 100% transactions of the dataset it is generated from. Figure 21 illustrates how the memory usage and run time of HUSP-Stream for producing HUSPs vary with different dataset sizes. We observe that the run time increases (almost) linearly when the number of transactions increases. This indicates that HUSP-Stream scales well with the size of datasets.
6. Conclusion and Future Works

We presented a novel framework for mining high utility sequential patterns over static and streaming data. We proposed two algorithms, HUSP-Miner and HUSP-Stream, to discover high utility sequential patterns in a sequence database and transaction-sensitive sliding window over an itemset-sequence stream. Two data structures named ItemUtilLists and HUSP-Tree were proposed to dynamically maintain the essential information of potential HUSPs. We also defined a new over-estimated sequence utility measure named Sequence-Suffix Utility (SFU), and used it to effectively prune the HUSP-Tree. Both real and synthetic datasets were used to show the performance of HUSP-Stream. In the experiments, we compared HUSP-Miner with USpan [26], a state-of-the-art algorithm for mining HUSPs in static databases. We also compared HUSP-Stream with two approaches that use USpan [26] to learn HUSPs over data streams. The experiments show that HUSP-Miner and HUSP-Stream substantially outperform USpan-based approaches in the number of generated potential HUSPs, run time and memory usage especially when the size of the dataset is very large (e.g., HUSP-Stream updates the data structures (window sliding time) up to three orders of magnitudes (1,500 times) faster than the USpan on DS2 dataset when the minimum utility threshold is 0.1%). The experimental results also show that our proposed SFU tree-pruning strategy is much more effective than the TSWU strategy. Our framework is scalable in both time and memory, and serves as an efficient solution to the new problem of mining high utility sequential patterns over static and stream data.

Although we first incorporate the concept of streaming mining into high utility sequential pattern mining and address the problem of mining high utility sequential patterns over data streams in this work, it still leaves ample room for exploration in the future work. For example, in this paper, we measure the utility of a sequence by its maximum utility and have not considered other ways such as [2] to measure the utility of a sequence.

References