# MINING HIGH UTILITY PATTERNS OVER DATA STREAMS

MORTEZA ZIHAYAT KERMANI

A DISSERTATION SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

JULY 2016

# Abstract

Mining useful patterns from sequential data is a challenging topic in data mining. An important task for mining sequential data is *sequential pattern mining*, which discovers sequences of itemsets that frequently appear in a sequence database. In sequential pattern mining, the selection of sequences is generally based on the frequency/support framework. However, most of the patterns returned by sequential pattern mining may not be informative enough to business people and are not particularly related to a business objective. In view of this, *high utility sequential pattern (HUSP) mining* has emerged as a novel research topic in data mining recently. The main objective of HUSP mining is to extract valuable and useful sequential patterns from data by considering the *utility* of a pattern that captures a business objective (e.g., *profit*, *user's interest*). In HUSP mining, the goal is to find sequences whose utility in the database is no less than a *user-specified minimum utility threshold*.

Nowadays, many applications generate a huge volume of data in the form of *data streams*. A number of studies have been conducted on mining HUSPs, but they are

mainly intended for non-streaming data and thus do not take data stream characteristics into consideration. Mining HUSP from such data poses many challenges. First, it is infeasible to keep all streaming data in the memory due to the high volume of data accumulated over time. Second, mining algorithms need to process the arriving data in real time with *one scan* of data. Third, depending on the minimum utility threshold value, the number of patterns returned by a HUSP mining algorithm can be large and overwhelms the user. In general, it is hard for the user to determine the value for the threshold. Thus, algorithms that can find the most valuable patterns (i.e., top-k high utility patterns) are more desirable. Mining the most valuable patterns is interesting in both static data and data streams.

To address these research limitations and challenges, this dissertation proposes techniques and algorithms for mining high utility sequential patterns over data streams. We work on mining HUSPs over both a long portion of a data stream and a short period of time. We also work on how to efficiently identify the most significant high utility patterns (namely, the top-k high utility patterns) over data streams.

In the first part, we explore a fundamental problem that is how the limited memory space can be well utilized to produce high quality HUSPs over the *entire data stream*. An approximation algorithm, called *MAHUSP*, is designed which employs memory adaptive mechanisms to use a bounded portion of memory, to efficiently discover HUSPs over the entire data streams.

The second part of the dissertation presents a new sliding window-based algorithm to discover *recent high utility sequential patterns* over data streams. A novel data structure named *HUSP-Tree* is proposed to maintain the essential information for mining recent HUSPs. An efficient and single-pass algorithm named *HUSP-Stream* is proposed to generate recent HUSPs from *HUSP-Tree*.

The third part addresses the problem of top-k high utility pattern mining over data streams. Two novel methods, named *T-HUDS* and *T-HUSP*, for finding top-k high utility patterns over a data stream are proposed. *T-HUDS* discovers top-k *high utility itemsets* and *T-HUSP* discovers top-k *high utility sequential patterns* over a data stream. *T-HUDS* is based on a compressed tree structure, called *HUDS-Tree*, that can be used to efficiently find potential top-$k$ high utility itemsets over data streams. *T-HUSP* incrementally maintains the content of top-k HUSPs in a data stream in a summary data structure, named *TKList*, and discovers top-k HUSPs efficiently.

All of the algorithms are evaluated using both synthetic and real datasets. The performances, including the running time, memory consumption, precision, recall and F-measure, are compared.

In order to show the effectiveness and efficiency of the proposed methods in real-life applications, the fourth part of this dissertation presents applications of one of the proposed methods (i.e., MAHUSP) to extract meaningful patterns from a real web click-stream dataset and a real biosequence dataset. The utility-based sequential patterns are

compared with the patterns in the frequency/support framework. The results show that high utility sequential pattern mining provides meaningful patterns in real-life applications.

# Acknowledgements

First and foremost, I would like to express my sincere appreciation to my supervisor Prof. Aijun An for her continuous support of my PhD study and research, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having had a better advisor and mentor for my PhD study.

I would like to thank my committee members, Prof. Parke Godfrey and Prof. Jarek Gryz for their great support and invaluable advice. I also show gratitude for Prof. Nick Cercone (former thesis committee member), an excellent professor and pioneer in the data mining area, who unfortunately passed away a few months before the official dissertation defense.

The members of both data mining and database labs have contributed immensely to my personal and professional time at York University. The group has been a source of friendships as well as great collaborators. I am also thankful to the administrative and technical staff of our department for their finest support.

Lastly, I would like to thank my family for all their encouragement and love. Words cannot express the feelings I have for my parents for their constant unconditional support in every aspect of my life.

# Table of Contents

# List of Tables

# List of Figures

# 1 Introduction

## 1.1 Background

Nowadays, huge volumes of data are produced in the form of sequences. A *sequence* is an ordered list of events with or without concrete notions of time [44]. A *sequence database* consists of a set of sequences. These sequential data are valuable sources of information not only to find a particular event at a specific time, but also to discover particular sequential relationships by analysing the occurrences of certain events or sets of events. Sequential data are produced by many applications such as *consumer shopping sequences*, *Web access logs*, *DNA sequences*, sequences in *financial markets*, etc. Here are three examples drawn from different domains, where sequential data are produced.

**1) Retail Business:** Table 1.1 shows an example of sequential data from a retail store. Each record is a customer transaction. In this table, the first column contains IDs assigned to the transactions. The second column shows time stamps for the transactions. The last three columns represent the information about the purchased items in each transaction, the quantity of each purchased item in the transaction and the unit profit. In this

Table 1.1: An example of sequential data in a retail store

| Transaction ID | Timestamp | Customer ID | The items | Quantities | Unit Profit |
|---|---|---|---|---|---|
| $T_1$ | 10-09-2015 10:00:00 | $C_1$ | $\{Bread, Milk\}$ | $\{2, 6\}$ | $\{\$1, \$1\}$ |
| $T_2$ | 10-09-2015 11:00:00 | $C_2$ | $\{Birthday\ Cake\}$ | $\{2\}$ | $\{\$20\}$ |
| $T_3$ | 10-09-2015 11:30:00 | $C_3$ | $\{Birthday\ Card, Egg\}$ | $\{2, 3\}$ | $\{\$10, \$2\}$ |
| $T_4$ | 10-09-2015 12:00:00 | $C_1$ | $\{Bread, Milk, Yoghurt, Tuna\}$ | $\{2, 4, 3, 5\}$ | $\{\$1, \$1, \$4, \$2.5\}$ |
| $T_5$ | 10-09-2015 12:10:00 | $C_4$ | $\{Egg, Pizza, Juice, Milk\}$ | $\{5, 4, 2\}$ | $\{\$2, \$8, \$3, , \$1\}$ |
| $T_6$ | 11-09-2015 9:00:00 | $C_4$ | $\{Bread, Yoghurt, Milk\}$ | $\{2, 4, 3\}$ | $\{\$1, \$4, \$1\}$ |

dataset, extracting customers' shopping behavior patterns can address several important questions such as how to increase revenue by recommending products based on previously observed shopping behaviours.

**2) News Portal:** Table 1.2 shows an example of a news portal clickstream dataset $D$ with 5 sessions $\{S_1, S_2, S_3, S_4, S_5\}$. Each session contains a sequence of tuples, corresponding to the list of articles that a reader read in a visit to the news portal. For each article that a user read, the article id, whether the user clicked on the like button, whether she/he shared the news in a social media and the time that the user spent on the article are recorded. For example, tuple $\langle nw_1, 1, 1, 14\rangle$ in $S_1$ means that a user visited news $nw_1$, pressed the like button, shared $nw_1$ in a social media and also spent 14 minutes to read news article $nw_1$. From such a dataset, modeling users' reading behavior is a major way to obtain a deep insight into the users. Reading behaviour patterns are useful for the portal designers to understand users' navigation behavior and improve the portal design and

Table 1.2: An example of a web clickstream dataset

| Session $ID$ | $\langle newsID, Shared, Liked, Time\ Spent\rangle$ |
|---|---|
| $S_1$ | $\langle nw_1, 1, 1, 14\rangle\langle nw_3, 1, 0, 3\rangle\langle nw_5, 1, 1, 22\rangle\langle nw_6, 0, 1, 7\rangle$ |
| $S_2$ | $\langle nw_4, 0, 0, 4\rangle\langle nw_5, 1, 0, 15\rangle\langle nw_6, 1, 1, 18\rangle$ |
| $S_3$ | $\langle nw_2, 1, 1, 14\rangle\langle nw_4, 0, 0, 1\rangle\langle nw_5, 1, 0, 19\rangle\langle nw_6, 0, 1, 3\rangle$ |
| $S_4$ | $\langle nw_1, 1, 0, 4\rangle\langle nw_3, 1, 0, 8\rangle\langle nw_5, 0, 0, 13\rangle$ |
| $S_5$ | $\langle nw_4, 1, 0, 9\rangle\langle nw_5, 1, 0, 2\rangle$ |

e-business strategies. These patterns can be also used to build a news recommendation system.

**3) Bioinformatic:** Microarray has been widely used in the biomedical field for identifying genes that are differentially expressed in different biological states (e.g. diseased versus non-diseased). Table 1.3 shows a part of a time course microarray dataset obtained from a biological investigation which consists of three patients whose IDs are $P_1$, $P_2$ and $P_3$. In this table, the gene expression values of three genes $G_1$, $G_2$ and $G_3$ are presented over four time points $TP_1$, $TP_2$, $TP_3$ and $TP_4$. Identifying potential gene regulations that occur in a period of time is important for biologists. Such patterns allow researchers to compare gene expression in different tissues, cells or conditions and provide some information on the relative levels of expression of thousands of genes among samples (usually less than a hundred).

Motivated by the above examples and many examples from other businesses that involve sequential data, mining patterns in sequential data has become an interesting research topic. The problem of *sequential pattern mining* was first introduced by Agrawal

Table 1.3: An example of a time course microarray dataset

| *Patient IDs* | *Genes* | $TP_1$ | $TP_2$ | $TP_3$ | $TP_4$ |
|---|---|---|---|---|---|
| $P_1$ | $G_1$ | 240 | 546 | 100 | 50 |
| | $G_2$ | 321 | 98 | 454 | 974 |
| | $G_3$ | 410 | 350 | 251 | 243 |
| $P_2$ | $G_1$ | 128 | 786 | 135 | 344 |
| | $G_2$ | 253 | 820 | 482 | 90 |
| | $G_3$ | 290 | 150 | 256 | 864 |
| $P_3$ | $G_1$ | 600 | 188 | 99 | 40 |
| | $G_2$ | 500 | 555 | 510 | 80 |
| | $G_3$ | 200 | 400 | 350 | 450 |

and Srinkant [2] as follows. *Given a sequence database, where each sequence consists of a list of transactions ordered by transaction time and each transaction is a set of items, sequential pattern mining is to discover all sequential patterns that frequently appear in the database.* For example in Table1.1, mining sequential patterns is to find the sequences of products that are frequently purchased by customers in a time order. In the last two decades many techniques and algorithms such as *AprioriAll* [2], *GSP* [53], *FreeSpan* [29], *PrefixSpan* [50], *SPADE* [71] and *SPAM* [9] have been proposed to mine sequential patterns. The mined patterns have been used for different purposes such as *customer acquisition*[14], web page design [12] and future location prediction of mobile users [59].

Despite the usefulness of sequential pattern mining in many applications, these approaches assume that all items are equally important and consider only binary frequency values of items in transactions. In [16], the authors showed that such approaches were not sufficiently practical for industrial needs. In their study, the patterns returned by se-

quential pattern mining methods were handed over to business people, the results showed that business people were not able to effectively take over and interpret the patterns to solve business problems. According to their study, there are three reasons for the problem. First, many patterns returned by sequential pattern mining may not be informative enough so that business people do not know which patterns are truly interesting and actionable for their business. Second, most of the patterns are not particularly related to a business need. Third, business people often do not know how to take actions on them to support business decision-making and operations. Below are three applications drawn from retail business, news portal and bioinformatics where frequency-based sequential pattern mining is not sufficiently practical.

**1) Mining profitable shopping behaviour:** In Table 1.1 the number of occurrences of an item in a transaction (e.g., quantity) and the importance of an item (e.g., unit profit) are not considered in the traditional framework of sequential pattern mining. Hence, such a framework may discover a large number of sequential patterns having low selling profits and lose the valuable information on important patterns that will contribute high profits. According to the table, selling a *Birthday Cake* is much more profitable (e.g., \$40 in total) than selling a bottle of *milk* (e.g., \$15 in total), but a pattern containing a *Birthday Cake* is much more infrequent than the one with a bottle of *milk*. Hence, the sequences contain such profitable items may not be discovered by existing approaches. However, these profitable patterns are important in making business decisions for maxi-

mizing revenue or minimizing marketing or inventory costs.

**2) Mining user reading behaviour:** There are some common deficiencies in most previous approaches to user reading behaviour mining based on frequency-based pattern mining in news domain. First, they discover patterns based on statistical measures such as frequency which do not take the user's interest into account. It is very common that not all the news opened by the user are of interest to him/her. For example, in Table 1.2, the user in session $S_3$ clicked on the news $nw_4$ to find that it is not very interesting to him/her since he/she only spent one minute to browse the news article. Second, the intrinsic characteristics of news reading make web usage mining in news domain different than other domains. The news domain is a dynamic environment. When users visit a news portal, they are looking for new information or even surprising ones. However, the traditional approaches ignore the fact that the value of a news article changes over time.

**3) Mining disease-related gene expression sequences:** Applying frequency-based sequential pattern mining approaches to Table 1.3, we are able to identify potential gene regulations that occur in a period of time frequently. These methods mostly choose important gene expression sequences based on the *frequency/support* framework. Usually, such datasets are collected to study a specific disease. However, as clinical studies have shown, the frequency alone may not be informative enough to discover gene expression sequences regarding an specific disease. For example, some genes are more important than others in causing a particular disease and some genes are more effective than others

in fighting diseases. The sequences contain these highly valuable genes may not be discovered by the frequency-based approaches because they neither consider the importance of each gene, nor temporal behavior of genes under biological investigation.

Considering these challenges, the important question is how to find sequential patterns of business interest. In view of this, *utility* was introduced into sequential pattern mining to discover patterns based on a business objective such as increasing profits, reducing costs, user's interest or a specific disease. This led to *high utility pattern mining* where the patterns are selected as interesting patterns based on their utility value. There are two main branches of high utility pattern mining which are highly related to the topic of this thesis.

### 1.1.1 High Utility Itemset Mining

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items and each item $i_j \in I$ is associated with a positive number $p(i_j)$, called its *external utility* (which can be the price or profit) of item $i_j$. Let $D$ be a set of $N$ transactions: $D = \{T_1, T_2, ..., T_N\}$ such that for $\forall T_j \in D, T_j = \{(i, q(i, T_j))|i \in I, q(i, T_j)$ is the quantity of item $i$ in transaction $T_j\}$. **Utility of an item** $i$ **in a transaction** $T_j$ is defined as: $u(i, T_j) = q(i, T_j) \times p(i)$. Hence, **the utility of an itemset** $X$ **in a transaction** $T_j$ is defined by: $u(X, T_j) = \sum_{i \in X} u(i, T_j)$. For example, in Table 1.1, utility of itemset $\{Bread, Yoghurt\}$ is $(2 \times \$1 + 3 \times \$4) + (2 \times \$1 + 4 \times \$4) = \$32$. An itemset is a *high utility itemset (HUI)* if its utility (such as the total profit that

the itemset brings) in a database is no less than a minimum utility threshold.

Mining HUIs is computationally more complex than mining frequent itemsets. This is due to the fact that the utility of an itemset does not have the *downward closure property*, which would allow effective pruning of search space during the HUI mining process. In fact, the utility of an itemset may be higher than, equal to, or lower than those of its super-itemsets and sub-itemsets [3, 52, 66]. In the last decade, several techniques and algorithms have been proposed for mining high utility itemsets. The $MEU$ (Mining with Expected Utility) [64] is the first high utility itemset mining method. The authors defined the problem of mining high utility itemsets, and proposed a theoretical model of high utility itemset mining. Specifically, two types of utilities for items, namely transaction utility (referred as internal utility in our definitions above) and external utility, were first proposed. They also proposed a utility upper bound called *Expected Utility* for the itemset to prune unpromising candidates. The *Two-Phase* method presented in [42] is one of the most cited papers in high utility itemset mining. The authors proposed an over-estimate utility (i.e., $TWU$) model for mining high utility itemsets. The main advantage of $TWU$ is its downward closure property. In the first phase, *Two-Phase* discovers all itemsets whose TWU is more than the threshold (i.e., $HTWU$ itemsets). Then in the second phase, it scans the database one more time to extract the true high utility itemsets from the HTWU itemsets. Based on the $TWU$ model, *CTU-Mine* [24] was proposed that is more efficient than *Two-phase* in dense databases when the minimum

utility threshold is very low. To reduce the number of candidates in each database scan, the isolated item discarding strategy ($IIDS$) was proposed in [37]. Applying $IIDS$, the authors proposed two efficient algorithms $FUM$ and *DCG+*. The authors of [34] proposed tighter upper bounds of utility values than $TWU$ values. In [57] a pattern growth approach (i.e., UP-Growth) was proposed to discover high utility patterns in two scans of database. The improved version of UP-Growth [57] was presented in [55]. Recently, some works have focused on mining HUIs in one scan of database. *HUI-Miner* (High Utility Itemset Miner) proposed by [40] is able to discover HUIs in one scan of database. In [33] a high utility itemset approach is proposed that discovers HUIs in a single phase without generating candidates.

### 1.1.2 High Utility Sequential Pattern mining

High utility itemset mining methods do not consider the ordering relationships between items or itemsets. Considering the sequential orders between itemsets makes the mining process much more challenging than mining high utility itemsets. To address this problem, *high utility sequential pattern (HUSP)* mining has emerged in data mining recently [3, 4, 66]. HUSP mining finds sequences of items or itemsets whose utility is higher than a user-specified utility threshold. The concept of HUSP mining was first proposed by Ahmed et al [4]. They 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. Shie et al. [52] proposed a framework for mining HUSPs in a mobile environment. Yin et al. [66] proposed the *USpan* algorithm for mining HUSPs, in which a lexicographic tree was used to extract the complete set of high utility sequential patterns.

## 1.2 Motivation

A *data stream* is an ordered and unbounded list of records (e.g., items/itemsets/sequences). Many applications generate huge volumes of data streams such as *network monitoring*, *ATM operations in banks*, *sensor networks*, *web clickstreams*, *transactions in retail chains* and many others. For example, Table 1.1 can be considered as a part of a data stream consisting of transactions ordered by their time stamps. Since transactions are done by different customers, this data stream can be also conisdered to have multiple sequences of transactions, each corresponding to a customer. A customer is likely to shop more than once in the retail store. Hence, the sequences of customers such as $C_1$ and $C_2$ become longer and longer over time and also it is very likely that new customers start shopping in the future and thus new sequences can be generated as time evolves.

An algorithm dealing with data streams need to process the data in real time with one scan of data. There are three main types of stream-processing models: *damped window based*, *sliding window based* and *landmark window based*.

In the *damped window based* (also called *time-fading* window based) model, each

data record (e.g., a transaction) is assigned with a weight that decreases over time. Therefore, in this model, recent data are more important than old ones. However, it is difficult for users who are not domain experts to choose an appropriate decay function or decay rate for this model. The *sliding window based* model captures a fixed number of most recent records in a window, and it focuses on discovering the patterns within the window. When a new record flows into the window, the oldest one is expired and its effect is eliminated from the window. 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. The *landmark window* based model is used for such a purpose, which treats all data records starting from a past time point (called *Landmark*) until the current time equally and discovers patterns over a long period of time in the data stream. Figure 1.1 summarizes the different stream processing models. In this figure, a data record can be an item, a transaction or a sequence.

All in all, mining such data poses many challenges due to its intrinsic characteristics. First, it is infeasible to keep all streaming data in the memory due to the high volume of data accumulated over time. Second, mining algorithms need to process the arriving data in real time with one scan of data. Third, the distribution of data varies over time, and hence analysis results need to be updated in real time.

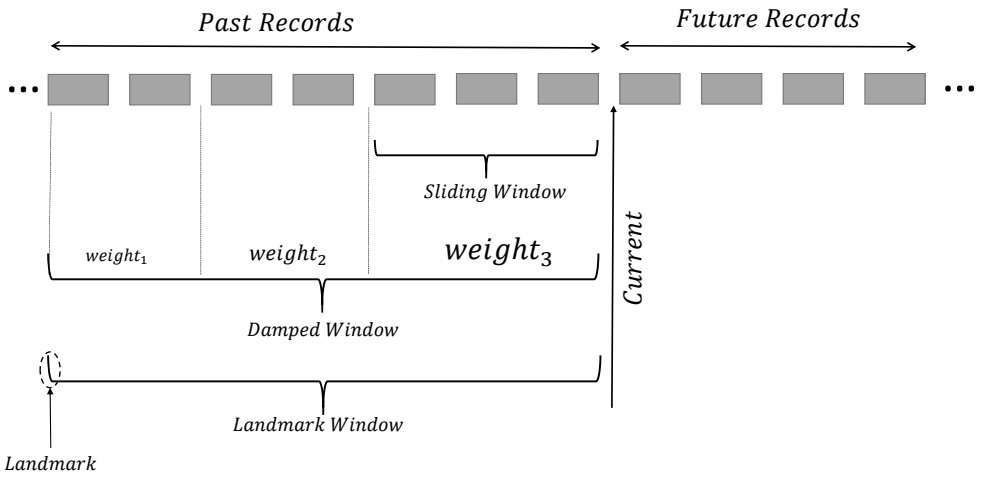



Figure 1.1: Data stream processing models

There are some recent studies on mining data streams, including approximate frequency counts over data streams [38, 45, 48], mining frequent patterns in data streams[21, 35, 65] and high utility itemset mining over data streams [7, 42, 56], but no work has been done to mine high utility sequential patterns over data streams. The advantage of mining HUSPs over mining frequent sequential patterns from a data stream is that a business objective can be considered during pattern discovery as the utility. There are unique challenges in discovering HUSPs over data streams. Below, we first present the research objectives and challenges and then we describe our contributions to address the challenges.

## 1.3 Research Objectives

The objective of this dissertation is to develop techniques and algorithms for mining high utility sequential patterns over data streams. We work on mining HUSPs over both a long portion of a data stream and a short period of time. We also work on how to efficiently identify the most significant high utility patterns (namely, the top-k high utility patterns) over data streams. In this dissertation, we assume that the data in the data stream arrive very fast and in a large volume, hence mining algorithms need to process the arriving data in real time with one scan of data. In particular, we define and address the following research issues.

### 1.3.1 Incremental High Utility Sequential Pattern Mining over the Entire Data Streams

In some applications long-term monitoring or planning is necessary and users want to discover patterns over a long period of time in the data stream. For example, the users may want to detect important buying sequences of customers since the beginning of a year or since the store changed its layout. A complete re-scan of a long portion of a data stream is usually impossible or prohibitively costly. Mining from a dynamically-increasing data streams allows incremental maintenance of patterns over a long period of time in a data stream. However, incremental mining of HUSPs over a data stream is

challenging due to the need of capturing the information of data over a potentially long period of time. Since real-time stream processing does not allow us to scan the data more than once, information about the data processed so far need to be summarized and kept in memory. However, it is possible that the amount of information we need to keep exceeds 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 the mining algorithm should be able to adapt its computation and memory usage to produce the best possible solutions under the memory constraint. That is, accurate approximation of the information needed for the mining process is necessary.

### 1.3.2 Mining Recent High Utility Sequential Patterns over Data Streams

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. That is, the mining method should capture most recent records (i.e., transactions/sequences) in a window, and focus on discovering the patterns within the window. When a new record flows into the window, the oldest one should be eliminated. To discover recent HUSPs, a naive approach is to apply an existing algorithm on static data to re-generate all HUSPs when the new record is added to the window and the oldest record is expired. However, this approach is very time-consuming because it needs to re-run the whole

mining process on the window even just a very small portion of the data is changed.

### 1.3.3 Mining Top-k High Utility Patterns over Data Streams

A threshold-based high utility pattern (i.e., itemset/sequence) mining approach enables us to discover the complete set of high utility patterns with a pre-defined minimum utility threshold. However, it is often difficult for the user to specify a minimum utility threshold, especially if the user has no background knowledge in the application domain. If the threshold is set too low, a large number of patterns can be found, which is not only time and space consuming [61], but also makes it hard to analyze the mining results. On the other hand, if the threshold is set too high, there may be very few or even no high utility patterns being found, which means that some interesting patterns are missed. Therefore, it is more reasonable to ask users to set a bound on the result size and discover top-k high utility patterns over data streams. The main challenge is that the minimum utility threshold is not given in advance to mine top-k high utility patterns. In the threshold-based high utility pattern mining approaches, the algorithms can prune the search space efficiently with the given threshold. However, in the scenario of top-k high utility pattern mining, the threshold is not provided. Therefore, the minimum utility threshold is initially set to 0. The mining task should gradually raise the threshold to prune the search space. Hence, the challenge is to design strategies to raise the threshold as high as possible to prune the search space as early as possible.

### 1.3.4 High Utility Sequential Patterns in Real Life Applications

Much of existing HUSP mining research has focused on devising techniques to discover patterns from databases efficiently. Relatively little attention has been paid to show applicability of the algorithms in real life scenarios. There are several challenges to use a HUSP mining algorithm in a real life application such as how to define the utility so that it reflects the objective of the application effectively and how to convert input sequential database to a utility-based sequential database.

## 1.4 Research Contributions

### 1.4.1 Memory Adaptive High Utility Sequential Pattern Mining over Data Streams

To solve the problem of *mining high utility sequential patterns over the entire data streams*, we propose a memory-adaptive high utility sequential pattern mining over data streams. In particular, our method is based on a specific type of the landmark window in which the *landmark* is the beginning of the data stream. Our contributions are summarized as follows.

- We propose a novel method for incrementally mining HUSPs over data streams. Our method identifies high utility sequential patterns over a long period of time.

- 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 candidate is discovered.

- 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*.

- 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.

### 1.4.2 Sliding Window-based High Utility Sequential Pattern Mining over Data Streams

To address the problem of *mining recent high utility sequential patterns over data streams*, we propose a sliding window-based high utility sequential pattern mining over data streams. Our contributions are summarized as follows.

- We define the problem of sliding window-based high utility sequential pattern mining to discover recent HUSPs over data streams.

- We propose a new utility model, called *suffix utility (SFU)*, to over-estimate the sequence utility. We prove that $SFU$ of a sequence is an upper bound of the

utilities of its super-sequences, which can be used to effectively prune the search space in finding HUSPs.

- We propose 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 in a sliding window over a data stream.

- We propose a new one-pass algorithm called *HUSP-Stream (High Utility Sequential Pattern Mining over Data Streams)* for efficiently constructing and updating *ItemUtilLists* and *HUSP-Tree* by reading a transaction in the data stream *only once*.

### 1.4.3 Top-k High Utility Pattern Mining over Data Streams

To address the problem of *top-k high utility pattern mining over data streams*, we propose two methods to discover top-k high utility itemsets and top-k high utility sequential patterns over data streams. Our contributions are summarized as follows.

- We first propose a method, called T-HUDS, for mining top-$k$ high utility itemsets from data streams. Then, we propose another method, called T-HUSP, for mining top-k high utility sequential patterns over data streams.

- We propose several strategies for initializing and dynamically adjusting the minimum utility threshold during the top-$k$ HUI/HUSP mining process.

- We conduct extensive experiments on both real and synthetic datasets to evaluate the performance of the proposed algorithms. Experimental results show that T-HUDS and T-HUSP serve as efficient solutions for the problems of top-k high utility itemset mining over data streams and top-k high utility sequential pattern mining over data streams, respectively.

### 1.4.4 Mining Meaningful Patterns in Real Life Applications

In order to demonstrate the applicability of the proposed methods in practical cases, we discover meaningful patterns in two real-life applications. Our contributions are summarized as follows.

- We conduct an analysis on a real web clickstream dataset obtained from a Canadian news web portal to extract web users' reading behavior patterns.
- We analyze a publicly available time course microarray dataset to identify gene sequences correlated with a specific disease.
- Using several quality measures, the mined utility-based sequential patterns are compared with the patterns in the frequency/support framework. The results show that our methods provide more meaningful patterns in real-life applications.

## 1.5 Thesis Structure

This dissertation is organized as follows.

In Chapter 2, we first present an introduction to the traditional frequent pattern mining framework, including existing work in frequent itemset/sequence mining over static data and data streams and top-k frequent itemset/sequence mining over data streams. Then, an overview of high utility pattern mining framework will be discussed. This overview describes existing work in high utility itemset mining, high utility itemset mining over data streams, high utility sequential pattern mining and top-k high utility itemset/sequence mining.

In Chapter 3, we explore a fundamental problem that is how the limited memory space can be well utilized to produce high quality HUSPs over the entire 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. An efficient tree structure, called *MAS-Tree*, is proposed to store potential HUSPs over a data stream. *MAHUSP* guarantees that all HUSPs are discovered under certain circumstances. Our experimental study on real and synthetic datasets shows that our algorithm cannot only discover HUSPs over data streams efficiently, but also adapt to allocated memory without sacrificing much the quality of discovered HUSPs.

In Chapter 4, we propose a new method for *sliding window-based high utility se-*

*quential pattern mining over a data stream*. A novel data structure named *HUSP-Tree* is proposed to maintain the essential information for mining HUSPs. HUSP-Tree can be easily updated when new data arrive and old data expire in a data stream. An efficient and single-pass algorithm named *HUSP-Stream* is proposed to generate HUSPs from *HUSP-Tree*. When data arrive at or leave from the window, *HUSP-Stream* incrementally updates *HUSP-Tree* to find HUSPs based on previous mining results. *HUSP-Stream* uses a new utility estimation model to more effectively prune the search space. Experimental results on real and synthetic datasets show that HUSP-Stream outperforms the state-of-the-art algorithms and serves as an efficient solution to the problem of mining recent high utility sequential patterns over data streams.

In Chapter 5, we propose two novel methods, named *T-HUDS* and *T-HUSP*, for finding top-k high utility patterns (i.e., itemsets/sequences) over a data stream. *T-HUDS* discovers top-k high utility itemsets and *T-HUSP* discovers top-k high utility sequential patterns over a data stream. *T-HUDS* is based on a compressed tree structure, called *HUDS-Tree*, that can be used to efficiently find potential top-$k$ high utility itemsets over sliding windows. *T-HUDS* uses a new utility estimation model to more effectively prune the search space. We also propose several strategies for initializing and dynamically adjusting the minimum utility threshold. We prove that no top-$k$ high utility itemset is missed by the proposed method. Inspired by T-HUDS, we propose a single pass algorithm, called *T-HUSP*, to incrementally maintain the content of top-k HUSPs in the

sliding window in a summary data structure, named *TKList*, and discover top-k HUSPs efficiently. In addition, two efficient strategies are proposed for raising the threshold. Our experiments are conducted on both synthetic and real datasets. The results show that both methods incorporating the proposed strategies demonstrate impressive performance without missing any top-k high utility itemset/sequential patterns.

In Chapter 6, in order to show the effectiveness and efficiency of our proposed methods in real-life applications, we apply MAHUSP algorithm to a web clickstream dataset obtained from a Canadian news web portal to showcase users' reading behavior and also to a real biosequence database to identify disease-related gene expression sequences. The results show that MAHUSP effectively discovers useful and meaningful patterns in both cases.

In Chapter 7, we review the contributions of the dissertation and summarize the directions for future.

Figure 1.2 shows the research profile of this thesis.

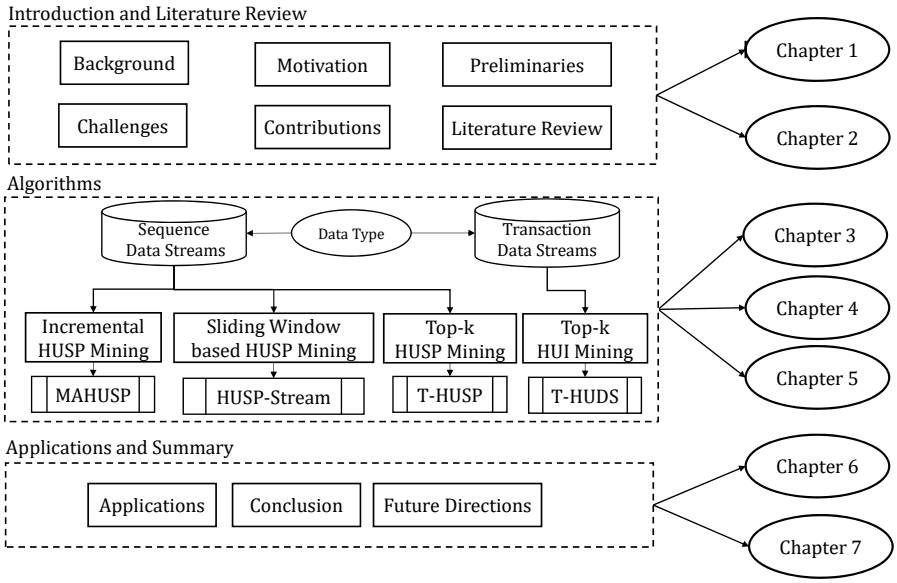



Figure 1.2: Thesis profile

# 2 Literature Review

In this Chapter, we first introduce the traditional frequent pattern mining framework, which contains frequent itemset mining, frequent sequential pattern mining algorithms over static data and data streams and top-k frequent itemset/sequence mining over data streams. Then we present an overview of high utility pattern mining framework, which contains an introduction to the research so far, high utility itemset mining, high utility itemset mining over data streams, high utility sequential pattern mining and top-k high utility itemset/sequence mining.

## 2.1 Frequent Pattern Mining

### 2.1.1 Frequent Itemset Mining

Frequent itemsets play an essential role in many data mining tasks which association rules mining is one of the most popular ones. The motivation for finding association rules came from the need to analyze customer transactions data to examine customer behaviour in terms of purchased products. Association rules describe how often items

are purchased together. An association rule $beer \rightarrow diaper(80\%)$ states that four out of five customers that bought beer also bought diaper. Such rules are helpful for decisions concerning product pricing, promotions, store layout and many others.

In 1994, Agrawal et al [1] proposed a property called *Downward Closure Property*, which also known as the *Apriori Property* [1] and defined as follows. Given two itemsets $X$ and $Y$, $support(X)$ (i.e., frequency of $X$) is equal or less than $support(Y)$ (i.e., frequency of $Y$) if $X \subseteq Y$. Based on the *apriori property*, they proposed the *Apriori algorithm* to find frequent itemsets. An itemset is called *frequent itemset* if its frequency is no less than a given minimum support threshold. The Apriori algorithm discovers frequent itemsets using a level-wise procedure. First, it scans the database to find 1-itemset candidates (itemsets with only one item) and prunes the candidates whose support is less than the minimum support threshold. Then, it joins the frequent 1-itemsets to generate 2-itemset candidates, and keeps those whose frequency satisfies the minimum support. The process repeats recursively until there is no more candidates to be generated, by which time the frequent itemsets have been discovered. With the foundation frequent pattern mining (namely, downward closure property), many algorithms were subsequently published [39].

However, Apriori-based algorithms generate a huge number of candidates and scans the original database multiple times to check the frequency of the candidates. In 2000, Han et al [30] proposed an algorithm called FP-Growth[30] to address this problem. The

FP-Growth algorithm is based on a divide-and-conquer strategy to find frequent itemsets without candidate generation. The foundation of the algorithm is a tree data structure named *Frequent-Pattern tree (FPtree)*, which stores the transaction database information. FP-Growth scans the database $D$ once to find the frequent and infrequent items. The infrequent items are pruned from the original database, and the updated database $D'$ is retained. Then, the FP-tree is constructed based on $D'$. The FP-tree is then divided into a group of conditional databases, each one associated with one frequent pattern. Lastly, each conditional database is mined separately. The process is recursively invoked until no conditional databases can be generated.

*Eclat* [70], is another algorithm which is different from *Apriori* and *FP-Growth*. *Eclat* benefits from the structural properties of frequent itemsets to discover patterns efficiently. The items are organized into a subset lattice search space, which is decomposed into small independent chunks or sub-lattices, which can be stored in memory. The authors proposed lattice traversal techniques to discover all the long frequent itemsets and their subsets efficiently.

### 2.1.2 Frequent Sequential Pattern Mining

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items and $D$ be a set of $N$ sequences: $D = \{S_1, S_2, ..., S_N\}$ where $S_j$ is a sequence of itemsets. The support of a sequence $\alpha = \langle X_1, X_2, ..., X_i \rangle$ in sequence dataset $D$ is the number of sequences in $D$ which contain $\alpha$. If the support

$\alpha$ satisfies a minimum support threshold, $\alpha$ is a *frequent sequential pattern*. Given sequence $\beta = \langle X_1', X_2', ..., X_j' \rangle$ $i \leq j$), $\alpha$ is a sub-sequence of $\beta$ or equivalently $\beta$ is a super-sequence of $\alpha$ if and only if there exist integers $1 \leq e_1 < e_2 < ...e_i \leq j$ such that $X_1 \leq X_{e_1}', X_2 \leq X_{e_2}', ..., X_i \leq X_{e_i}'$.

Mining sequential patterns in sequence databases was first introduced by Agrawal et al [2]. Since then, quite a few algorithms have been proposed such as AprioriAll [2], GSP [53], FreeSpan [29] , PrefixSpan [50], SPADE [71] and SPAM [9] to find sequential patterns efficiently. These algorithms can be generally categorized as using a *horizontal database* (e.g., AprioriAll, GSP, FreeSpan and PrefixSpan) or a *vertical database* (e.g., SPADE, SPAM and ClaSP). A vertical representation provides the advantage of calculating frequencies of patterns without performing costly database scans. This allows vertical mining algorithms to perform better on datasets having dense or long sequences than algorithms using the horizontal format. All of the above algorithms rely on the *downward closure property*. Below, we briefly introduce the above algorithms.

**AprioriAll**: AprioriAll [2] is the first proposed algorithm to mine sequential patterns. It first finds all frequent 1-patterns. Then, it constructs and retains two types of lists, called the *candidate lists* and the *frequent pattern lists*. For every (k + 1)-sequence candidate produced by joining two frequent k-sequences, it requires to scan the original database to calculate the support value. The process repeats until no more patterns can be formed.

**SPADE:** SPADE (Sequential PAttern Discovery using Equivalent classes)[71] uses a vertical data format and is a level-wise sequential pattern mining algorithm. SPADE avoids multiple scans of the original database. Instead, SPADE builds a list of sequence IDs and elements, called *ID-list*, for each candidate. The support count of the candidate can be easily calculated from the list without scanning the original dataset.

**SPAM:** SPAM (Sequential PAttern Mining) [9] is a depth-first algorithm. The authors introduced two novel joining operations called the *sequence-extension step (S-Step)*, *itemset-extension step (I-Step)* and also a *lexicographical tree* for the first time. *SPAM* uses a depth-first strategy to traverse the lexicographical tree to extract the complete set of frequent sequential patterns. SPAM also utilizes a vertical bitmap data structure and puts it in the memory such that the *joining* operation between two sequences is done extremely fast.

**PrefixSpan:** PrefixSpan (Prefix-projected Sequential pattern mining) [50] is an extended pattern-growth approach for mining frequent sequential patterns and is the first algorithm that does not generate candidates. *PrefixSpan* uses the *"prefix"* of the sequence to project the database. Then, it scans the projected database for the items to be concatenated to the prefix, and calculates the support for each item. It removes the infrequent concatenation items and only keeps the frequent items. Finally, for each frequent concatenation item, a new prefix and its corresponding smaller projected database is constructed. The process continues until no more frequent concatenation items can be

scanned.

### 2.1.3 Frequent Sequential Pattern Mining over Data Streams

A desired feature of stream mining algorithms is that when records are inserted into or deleted from the database, the algorithms can incrementally update the patterns based on previous mining results, which is much more efficient than re-running the whole mining process on the updated database. Several studies such as [15, 18, 19, 32, 46] have been performed for mining sequential patterns over data streams. Below, we briefly introduce these algorithms.

**MILE**: MILE [19] is an efficient algorithm to mine sequential patterns over data streams. The proposed algorithm recursively uses the knowledge of existing patterns to mine the new patterns efficiently. One unique feature of *MILE* is when some prior knowledge of the data distribution in the data stream is available, it can be incorporated into the mining process to further improve the performance of MILE.

**SMDS**: *SMDS* has two main features: first, it summarizes batches of transactions using a sequence alignment method. This process works based on a greedy clustering algorithm. The algorithm provides distinct clusters of sequences by considering the main features of Web usage sequences. Second, frequent sequences found by *SMDS* are stored in a prefix tree structure. This data structure enables *SMDS* to calculate the real support of each proposed sequence efficiently.

**IncSpam**: *IncSpam* [32] is single-pass algorithm to mine set of sequential patterns over a stream of itemset-sequences with a sliding window. The authors proposed a bit-sequence representation to reduce the memory requirement of the online maintenance of sequential patterns generated so far. The model receives sequences of a data stream and uses a bit-vector data structure, called *Customer Bit-Vector Array with Sliding Window* (CBASW), to store the information of items for each sequence. Lastly, a weight function is adopted in this sliding window model. The weight function can judge the importance of a sequence and ensure the correctness of the sliding window model.

**SPEED**: SPEED [15] identifies frequent maximal sequential patterns in a data stream. SPEED is based on a novel data structure to maintain frequent sequential patterns coupled with a fast pruning strategy. At any time, users can issue requests for frequent maximal sequences over an arbitrary time interval. Furthermore, SPEED produces an approximate support answer with an assurance that it will not bypass a user-defined frequency error threshold.

**SeqStream**: SeqStream [18] mines closed sequential patterns in stream windows incrementally. It works based on various novel pruning strategies and a synopsis structure called *IST* (Inverse Closed Sequence Tree) to keep inverse closed sequential patterns in current window. There are two main tasks in SeqStream. The first one is to efficiently update supports of tree nodes when an element is inserted to or removed from the window. The second one is to remove nodes that do not need to update or extend to eliminate

the related support counting and node extension costs as much as possible. *SeqStream* uses an insertion database and removal database to update the support information of current IST. These two databases are usually much smaller than the whole database in the sliding window. Therefore the support update task can be completed quickly. For the second task, SeqStream adopts various pruning strategies to safely skip nodes that do not need to be handled.

### 2.1.4 Top-k Frequent Itemset/Sequence Mining

A challenging problem with many frequent pattern mining methods is that the user needs to supply a minimum support threshold. In frequent itemsets mining, several methods were proposed to find top-k frequent itemsets in static datasets [22, 23, 31, 49]. There are also several methods for finding top-k frequent itemsets over data streams. Golab et al. [28] proposed an algorithm, called *FREQUENT*, for the top-k frequent item discovery in sliding windows. It performs well with bursty TPC/IP streams containing a small set of popular item types. Wong and Fu [60] proposed two algorithms to address the problem of top-k frequent $l$-itemsets ($1 \leq l \leq L$) mining over data streams. *TOPSIL-Miner* [62] is another recent algorithm for mining top-k significant itemsets over data streams, which works based on a prefix tree structure. This method is an approximation method and does not guarantee that the exact set of top-k frequent itemsets is found. To address the difficulty of setting minimum support, the problem of top-k sequential pattern mining

was defined by [58]. The authors proposed an algorithm called *TSP*. They proposed two versions of TSP for respectively mining (1) top-k sequential patterns and (2) top-k closed sequential patterns. Although these algorithms are efficient, it is difficult (if not impossible) to simply adapt them to high utility itemset/sequence mining.

## 2.2 High Utility Pattern Mining

In the previous section, we reviewed several frequent pattern mining algorithms which aim to discover various types of frequent patterns such as items, itemsets and sequences. However, in frequent pattern mining, the number of occurrences of an item inside a transaction is ignored in the problem setting, so is the importance (such as price or weight) of an item in the dataset. In order to assign different weights to items, weighted frequent pattern mining methods were proposed [54, 68, 69]. These methods work with different weights for different items. One of the first weighted frequent pattern mining methods is WARM (Weighted Association Rule Mining) [54]. Another method is *WFIM* [69] which was proposed to extend pattern growth algorithm to consider weighted items. This paper defined a weight range and a minimum weight constraint. Items were given different weights within the weight range. The support and weight of items were used to prune the search space. However these methods are just applicable when the frequency of an item in each transaction is either 1 or 0.

In view of this, high utility pattern mining has been studied recently [8, 17, 41, 57]. A

pattern is a *high utility pattern (HUP)* if its utility (such as the total profit that the pattern brings) in a dataset is no less than a minimum utility threshold. Below, we present two main types of high utility pattern mining approaches which are relevant to this dissertation.

### 2.2.1 High Utility Itemset Mining

**MEU**: The $MEU$ (Mining with Expected Utility) model [64] is the first high utility itemset mining method. In this paper, the authors defined the problem of high utility itemset mining. They described two types of utilities for items, internal utility and external utility. The internal utility of an item in a transaction is defined according to the information stored in the transaction. The external utility of an item is based on information which are not available in the transaction. The external utility is proposed as a new measure for describing user preferences. By analyzing the utility relationships among itemsets, they identified the utility bound property and the support bound property. Furthermore, they defined the mathematical model of high utility itemset mining based on these properties. $MEU$ does not use the downward closure property, hence a heuristic approach was proposed to predict if a pattern should be added to the candidate set. $MEU$ checks the candidate patterns using a prediction method with a high computational cost. Later, the $UMining$ algorithm [63] improved its performance. They defined an upper bound utility for each pattern. Using this upper bound, low utility patterns are pruned during

the mining process.

**Two Phase Method**: As mentioned before, the utility of an itemset may be equal to, higher or lower than that of its supersets and subsets. Therefore, we cannot directly use the anti-monotone property (also known as downward closure property) to prune the search space. To facilitate the mining task, Liu et al. [42] introduced the concept of *transaction-weighted downward closure*, which is based on the following definitions.

**Definition 1 Transaction-Weighted Utility (TWU)** of an itemset $X$ over a dataset $D$ is defined as: $TWU_D(X) = \sum_{X \subseteq T_j \bigwedge T_j \in D} TU(T_j)$, *where* $TU(T_j)$ *is the utility of transaction* $T_j$ *and is defined as* $TU(T) = \sum_{item \in T} u(item, T)$.

Clearly, $TWU_D(X) \geq u_D(X)$. In addition, $TWU$ satisfies the downward closure property, that is, for all $Y \subseteq X$, $TWU_D(Y) \geq TWU_D(X)$.

**Definition 2** An itemset $X$ is a *high transaction-weighted utilization* itemset (abbreviated as $HTWU$) if $TWU_D(X) \geq min\_util$, where *min_util* is called a *minimum utility threshold*.

The *Two-Phase* method in [42, 43] proposed based on the overestimate utility (i.e., $TWU$) model. The general process for mining high utility itemsets using $TWU$ from a database is as follows. In the first phase of *Two-Phase*, all of the $HTWU$ itemsets are discovered. In the second phase, the database is scanned one more time and the true high utility itemsets are extracted from the $HTWU$ itemsets. Since $TWU_D(X)$ is

an overestimate of the true utility of $X$, the procedure does not miss any high utility itemset. But the true utility of a generated itemset may be lower than the minimum utility threshold. Thus, *Two-Phase* uses the second phase to compute the exact utility of the $HTWU$ itemsets and remove those whose utility is lower than the threshold.

**CTU-Mine**: Based on the $TWU$ model, *CTU-Mine* [24] was proposed that is more efficient than *Two-phase* in dense databases when the minimum utility threshold is very low. In order to mine HUIs, a prefix tree, called *CUP-tree*, is constructed. It consists of an $ItemTable$ called $GlobalItemTable$ made up of all high $TWU$ items and a tree called *GlobalCUP-Tree* with nodes linked by pointers from $GlobalItemTable$, containing itemset information for utility mining. $GlobalItemTable$ contains all individual $HTWU$ items sorted in descending order by their $TWU$ values. Then, a new algorithm that traverses the tree using a bottom-up approach is presented. During mining process, the algorithm constructs another tree called a High Utility Pattern Tree (*HUP-Tree*) to maintain high utility itemsets and their utility values computed by traversing the *LocalCUP-Tree*. This data structure and algorithm extend the pattern growth approach. Later, to reduce the number of candidates in each database scan, the isolated item discarding strategy ($IIDS$) was proposed in [37]. Using $IIDS$, two efficient algorithms $FUM$ and *DCG+* were proposed by the authors.

**IHUP:** In [8], efficient tree structures were proposed to discover high utility itemsets in incremental databases. The authors proposed three new tree structures with the

*build once mine many* property for high utility pattern mining in an incremental database. They designed the first incremental tree structure, *Incremental High Utility Pattern Lexicographic Tree* ($IHUP_L - Tree$), according to the items lexicographic order. This tree can capture the incremental data without any restructuring operation, but the tree size cannot be guaranteed to be compact. The second incremental tree structure, *Incremental High Utility Pattern Transaction Frequency Tree* ($IHUP_{TF} - Tree$), is designed according to the transaction frequency (descending order) of items to obtain a compact tree. However, it is not guaranteed that items having higher transaction frequency will also have a high utility value. If several low utility items appear in the upper portion of the tree, then the mining task will take longer. To ensure the candidates of high utility items will appear before low-utility items in any branch of the tree, the third incremental tree structure, *Incremental High Utility Pattern-Transaction-Weighted Utilization Tree* ($IHUP_{TWU} - Tree$) is designed according to transaction-weighted utilization of items in descending order, which ensures that only nodes containing candidate items are participating in the mining operation. The proposed algorithm is called $IHUP$ and works as follows. It first constructs one of the proposed tree structures based on transactions and then applies a pattern growth approach to generate candidates. In the second scan, all HUIs are discovered from the generated candidates.

**UP-Growth and UP-Growth$^+$**: In [57] a pattern growth approach (i.e., UP-Growth) was proposed to discover high utility itemsets in two scans. UP-Growth defines a tree

structure and four effective strategies *DGU*, *DGN*, *DLU* and *DLN* for mining HUIs. The information of high utility itemsets is maintained in a special data structure named *UP-Tree* (Utility Pattern Tree) such that the candidate itemsets can be generated with only two scans of the database. The improved version of [57] was presented in [55]. The proposed framework consists of three main steps. First, it scans the database twice to construct *UP-Tree* applying two proposed strategies: $DGU$ and $DGN$. In the second step, either *Up-Growth* or the other proposed method, called *UP-Growth*$^+$, is applied to generate candidates recursively. At the end, by computing the exact utility of each candidate, all HUIs are discovered.

Most of the above algorithms adopt a similar framework: firstly, generate candidate high utility itemsets from a database; secondly, compute the exact utilities of the candidates by scanning the database to identify high utility itemsets. However, the algorithms often generate a very large number of candidate itemsets and thus are confronted with two problems: (1) excessive memory requirement for storing candidate itemsets; (2) a large amount of running time for generating candidates and computing their exact utilities. When the number of candidates is so large that they cannot be stored in memory, the algorithms will fail or their performance will be degraded due to memory thrashing.

Recently, some works have focused on mining HUIs in one scan of data base: *HUI-Miner* and *d*$^2$*HUP*. Both algorithms work in a new framework, which discovers high utility itemsets without generating candidates. In the new framework, high utility itemsets

are directly identified from a set-enumeration tree, which is the search space constructed by enumerating itemsets with prefix or suffix extensions. To facilitate the computation of itemsets' utility values and prune unpromising itemsets efficiently, both HUI-Miner and d$^2$HUP maintain utility information and utility upper bound information by two kinds of data structure respectively. By avoiding the generation of a large number of candidates, HUI-Miner and d$^2$HUP show excellent performance and outperform the state-of-the-art algorithms based on FP-growth over one order of magnitude.

**HUI-Miner**: *HUI-Miner* (High Utility Itemset Miner) proposed by [40] is able to discover HUIs in one scan of database. *HUI-Miner* uses a new data storage, called *Utility-list*, to maintain utility information about the patterns. Once *Utility-list* is constructed, *HUI-Miner* does not need the database and can discover the HUIs and their exact utilities directly. The mining process is similar to Apriori-based algorithms but instead of scanning database several times, *HUI-Miner* scans the data once and after that it only scans the *Utility-list* several times.

**d$^2$HUP:** In [33] a high utility itemset discovery approach (d$^2$HUP) is proposed that works in a single phase without generating candidates. They propose a novel data structure to maintain original utility information, named $CAUL$, during mining process and then enumerate patterns by prefix extensions. A high utility itemset growth approach is proposed, which enumerates an itemset as a prefix extension of another itemset. It computes the utility of each enumerated itemset and an upper bound on the utilities of prefix

extensions of the itemset in order to directly identify high utility itemsets and to prune the search space. The proposed utility upper bound is tighter than $TWU$, and is further tightened by recursively filtering out items irrelevant in growing high utility itemsets with sparse data.

### 2.2.2 High Utility Itemset Mining over Data Streams

Recently, high utility itemset mining from data streams has become an active research topic in data mining [7, 42, 56].

**THUI-Mine**: *THUI-Mine* [56] was the first algorithm for mining temporal high utility itemsets from data streams. It explores the issue of efficiently mining high utility itemsets in temporal databases such as data streams. *THUI-Mine* discovers temporal high utility itemsets from data streams efficiently and effectively. The underlying idea of *THUI-Mine* algorithm is to integrate the advantages of Two-Phase algorithm [42] with the incremental mining techniques for mining temporal high utility itemsets. The main contribution of *THUI-Mine* is that it can efficiently identify the high utility itemsets in data streams so that the execution time for mining high utility itemsets can be reduced. That is, *THUI-Mine* discovers the temporal high utility itemsets in current time window and also discovers the temporal high utility itemsets in the next time window with limited memory space and less computation time by sliding window method. *THUI-Mine* first finds the length-1 and length- 2 candidate patterns, and then all the candidate patterns

from the length-2 candidate patterns are generated in order to reduce the overall database scans.

**MHUI-BIT and MHUI-TID**: Two algorithms, called *MHUI-BIT* and *MHUI-TID*, were proposed in [36] for mining high utility itemsets from data streams. The proposed algorithms consist of two major components, i.e., *item information* and a *lexicographical tree-based summary* data structure based on the item information. Two effective representations of item information, i.e., *Bitvector* and *TIDlist*, were developed and used in the proposed methods to restrict the number of candidates and to reduce the processing time and memory usage. The author proposed two algorithms to mine the set of high utility itemsets based on *Bitvector* and *TIDlist*, respectively. The first proposed algorithm based on the *Bitvector* representation is called *MHUI-BIT* (Mining High Utility Itemsets based on BITvector). Moreover, the second proposed method, called *MHUI-TID* (Mining High Utility Itemsets based on TIDlist), is based on *TIDlist* representation. Both algorithms are composed of three phases, i.e., *window initialization phase*, *window sliding phase*, and *high utility itemset generation phase*. In the first phase, the item information, i.e., *Bitvector* and *TIDlist*, are constructed and the transaction utility of each transaction within the current window is calculated. Then the proposed lexicographical tree-based summary data structure, called *LexTree-2HTU* (lexicographical tree with 2-HTUitemsets), based on the item information is constructed. The second phase, i.e., window sliding phase, is activated when the window is full and a new transaction ar-

rives. In this phase, two operations are performed. The first operation is to update the item-information. The second operation is to update the summary data structure *LexTree-2HTU*. In the third phase, the proposed algorithms use level-wise method to generate a set of candidate *k-HTU-itemsets* from the previous pre-known *(k-1)-HTU-itemsets*.

The proposed representations become very inefficient when the number of distinct items become large in a window. During the mining process, these methods only store length-1 and length-2 candidates. Then, other candidates whose length is more than two are generated using an Apriori-like level-wise candidate generation algorithm.

**GUIDE**: *GUIDE* is a framework proposed in [10] for mining a compact form of high utility itemsets from data streams with different models (i.e. the landmark, sliding and time fading window models). It works based on a tree structure, called *MUI-Tree*, which is constructed in one scan of the data stream. Depending on the type of the window model, the node structure in *MUI-Tree* is different. Once transactions are loaded into the memory, a process named *transaction-projection* is applied to produce the subsets of the transactions, called *projections*. This process may result in some pattern loss. After that, the projections are maintained into the tree. Finally, the high utility itemsets are discovered.

**HUPMS:** *HUPMS* [7] is a recent method for mining HUIs from data streams, which is based on the $TWU$ model. The authors proposed a novel tree structure, called *HUS-tree* (Incremental and Interactive Utility Tree for Stream data) to keep information about

patterns and their $TWU$ values. The proposed tree structure, *HUS-tree* arranges the items in lexicographic order. A header table is constructed to keep an item order in the proposed tree structure. Each entry in a header table explicitly maintains *item-id* and $TWU$ value of an item. Each node in the tree maintains *item-id* and batch-by-batch $TWU$ information to efficiently maintain the window sliding environment. To facilitate the tree traversals, adjacent links are also maintained in the tree structure. The mining process is as follows. To apply a pattern growth mining approach, *HUPMS* first creates a prefix tree for the bottom-most item by taking all the branches prefixing that item. Subsequently, conditional tree for that item is created from the prefix tree by eliminating the nodes containing items having low $TWU$ value with that particular item. Then during the second scan of the database, the exact utility of each candidate is calculated and HUIs are discovered. *HUS-tree* has the *build once mine many* property for interactive mining. That is, it can again mine the resultant patterns for different minimum utility thresholds without rebuilding the tree.

### 2.2.3 High Utility Sequential Pattern Mining

As mentioned several sequential pattern mining algorithms have been proposed such as AprioriAll [2], GSP [53], FreeSpan [29], PrefixSpan [46], SPADE [71] and SPAM [9]. Although the above algorithms are pioneers in sequential pattern mining, they treat all items as having the same importance/utility and assumes that an item appears at most

once at a time point, which do not reflect the characteristics in the scenario of several real-life applications and thus the useful information of sequences with high utilities such as high profits is lost. High utility sequential pattern mining [5, 6, 52, 66] considers the external utility (e.g., unit profits) and internal utility (e.g., quantity) of items such that it can provide users with patterns having a high utility (e.g., profit).

**UMSP**: *UMSP* [52] was designed for mining high utility mobile sequential patterns. Each itemset in a sequence is associated with a location identifier. With this feature, the utility of a mobile sequential pattern is also a single value. The authors integrate mobile sequential pattern mining with utility mining for finding high-utility mobile sequential patterns. Two different types of methods, namely level-wise and tree-based ones, are proposed for this problem. As the level-wise method, an algorithm called $UMSP_L$ (mining high Utility Mobile Sequential Patterns by a Level-wise method) is proposed. Not only supports but also utilities of patterns are considered in the level-wise mining processes. For tree-based method, two algorithms $UMSP_{T(DFG)}$ (mining high Utility Mobile Sequential Patterns by a Tree-based method with a Depth First Generation strategy) and $UMSP_{T(BFG)}$ (mining high Utility Mobile Sequential Patterns by a Tree-based method with a Breadth First Generation strategy) are proposed. Both of the two tree-based algorithms use a tree structure named $MTS - Tree$ (Mobile Transaction Sequence Tree) to summarize the corresponding information, such as locations, items, paths and utilities, in mobile transaction databases. *UMSP* searches for patterns within *MTS-Tree*, which is ef-

ficient. However, due to the specific constraint on the sequences, this algorithm can only handle specific sequences with simple structures (single item in each sequence element, and a single utility per item).

**UWAS**: In [6], an algorithm is specifically designed for utility web log sequences. The utility of a pattern can have multiple values, and the authors choose the utility with maximal values to represent a pattern's utility with two tree structures, i.e. *UWAS-tree* and *IUWAS-tree*. The proposed approach can handle both forward and backward references, avoids the level-wise candidate generation-and-test methodology, does not scan databases several times and considers both internal and external utilities of a web page. However, sequence elements with multiple items such as $\langle(c, 2)(b, 1)\rangle$ cannot be supported, and the scenarios considered in this paper are rather simple, which limit the algorithm's applicability for complex sequences.

**UI and US**: *UI* and *US* [5] extend traditional sequential pattern mining. For mining high-utility sequential patterns, they propose two new algorithms: Utility Level is a high-utility sequential pattern mining with a level-wise candidate generation approach, and Utility Span is a high-utility sequential pattern mining with a pattern growth approach. The utility of a pattern is calculated in two ways. First, the utilities of sequences having only distinct occurrences are added together. Second, the highest occurrences are selected from sequences with multiple occurrences and used to calculate the utility.

**USpan**: Recently, Yin et al.[66] proposed a new definition for high utility sequen-

tial pattern mining, which aims at finding sequences having a maximum utility. They proposed different definition of the utility of a sequence.

**Definition 3** (**Occurrence of a sequence** $\alpha$ **in a sequence** $S_r$) Given a sequence $S_r =< D_1, D_2, ..., D_n >$ and a sequence $\alpha =< X_1, X_2, ..., X_Z >$ where $D_i$ and $X_i$ are itemsets, $\alpha$ occurs in $S_r$ iff there exist integers $1 \leq e_1 < e_2 < ... < e_Z \leq n$ such that $X_1 \subseteq D_{e_1}$, $X_2 \subseteq D_{e_2}, ..., X_Z \subseteq D_{e_Z}$. The ordered list of transactions $D_{e_1}, D_{e_2}, ..., D_{e_Z}$ is called an occurrence of $\alpha$ in $S_r$. Since $\alpha$ may have multiple occurrences in $S_r$, the set of all occurrences of in $S_r$ is denoted as $OccSet(\alpha, S_r)$.

**Definition 4** (**Utility of a sequence** $\alpha$ **in a sequence** $S_r$) Let $\bar{o} =< T_{e_1}, T_{e_2}, ..., T_{e_Z} > \in OccSet(\alpha, S_r)$ be an occurrence of $\alpha =< X_1, X_2, ..., X_Z >$ in the sequence $S_r \in DS$. The utility of $\alpha$ w.r.t. $\bar{o}$ is defined as $su(\alpha, \bar{o}) = \sum_{i=1}^{Z} su(X_i, S_r^{e_i})$. The utility of $\alpha$ in $S_r$ is defined as $su(\alpha, S_r) = max\{su(\alpha, \bar{o}) | \forall \bar{o} \in OccSet(\alpha, S_r)\}$.

Note that the maximum utility of sequence $\alpha$ among all the occurrences of $\alpha$ in sequence $S_r$ is used as the utility of $\alpha$ in $S_r$.

Then, they proposed the *USpan* algorithm for mining high utility sequential patterns. To satisfy *Downward Closure Property*, the authors have defined and used *Sequence Weighted Utilization (SWU)* and *Sequence Weighted Downward Closure (SDCP)*. In $SWU$, the idea behind is very similar to that of $TWU$, such that $SWU$ is also defined as the sum of the transaction utilities of all the transactions containing the pattern, so $SWU$

is sequence-weighted counterpart of $TWU$. Based on the $SWU$ of a pattern, an item is called promising if adding it to a candidate pattern results in a new pattern whose $SWU$ is greater than or equal to the minimum utility threshold. Therefore, the pruning they perform before candidate generation is based on this $SDCP$. They also proposed a depth pruning strategy which is a *Pruning After Candidate Generation (PACG)* mechanism.

### 2.2.4 Top-k High Utility Itemset/Sequence Mining

Top-$k$ high utility itemset mining was first introduced in [17]. However, its high utility itemset definition differs from the ones used in the recently proposed methods and in ours. Recently, the $TKU$ method was proposed in [61] to find top-k high utility itemsets over a static dataset. The proposed approach mines top-k high utility itemsets without setting the minimum utility threshold. It works based on *Up-Growth* [57]. Although it can find top-k *HUIs* effectively, it is not designed for data streams. Not only is it not able to adapt itself dynamically over streaming data, but also the proposed strategies for raising the threshold have much room to be improved so that it could generate few candidates and run faster in a data stream environment.

In [67], the authors proposed an algorithm, called *Top-k high Utility Sequence (TUS for short) mining*, to identify top-k high utility sequential patterns. It works based on *USpan* [66]. They also proposed three features to handle the efficiency problem, including two strategies for raising the threshold and one pruning for filtering unpromising items.

## 2.3 Summary

In this chapter, we introduced high utility sequential pattern mining and the related work. In Section 2.1, the problem of frequent sequential pattern mining is defined and discussed. We presented several algorithms such as *AprioriAll* [2], *GSP* [53], *FreeSpan* [29], *PrefixSpan* [50], *SPADE* [71] and *SPAM* [9]. Then we described several methods for solving the problem of frequent sequential pattern mining over data streams. Lastly, we discussed some existing work on top-k frequent itemset/sequence mining.

In Section 2.2, we introduced the high utility pattern mining framework. First, we discussed the problem of high utility itemset mining over data streams and some existing work has been explained. In sub-section 2.2.3, we briefly defined high utility sequential pattern mining and presented the state-of-the-arts methods. Lastly, we described the recent work on top-k high utility itemset/sequence mining.

# 3 Memory Adaptive High Utility Sequential Patterns over Data Streams

## 3.1 Introduction

Although some preliminary works have been conducted on high utility sequential pattern (HUSP) mining, existing studies [3, 4, 52, 66] mainly focus on mining high utility sequential patterns in static databases and do not consider real-world applications that involve *data streams*. In some applications users want to discover patterns over a long period of time in the data stream. 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. 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. Mining from a dynamically-increasing landmark window allows incremental maintenance of patterns over a long period of time in a data stream. Considering all these applications, in this chapter we aim at finding high utility sequential patterns (HUSPs) over a landmark window. In particular, the landmark in our method is the beginning of the data stream. Thus, we are dealing with the problem of incrementally learning HUSPs over the entire data stream. In this learning scenario, a complete re-scan of a long portion of the data stream is usually impossible or prohibitively costly.

Compared with other data stream mining tasks, there are unique challenges in discovering HUSPs over landmark windows.

1. HUSP mining needs to search a large search space due to a combinatorial number of possible sequences.

2. 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 super-sequences and sub-sequences [3, 52, 66]. Consequently, keeping up the pace with high speed data streams can be very hard for a HUSP mining task.

3. 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 chapter, we tackle these challenges and propose a *memory-adaptive approach* to finding 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.

1. We propose a novel method for incrementally mining HUSPs over a data stream. Our method can identify high utility sequential patterns over a long period of time.

2. 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.

3. Two efficient memory adaptive mechanisms are proposed to deal with the situation when the available memory is not enough to add a new potential HUSPs to *MAS-Tree*. The proposed mechanisms choose the least promising patterns to remove from the tree to guarantee that the memory constraint is satisfied.

4. Using *MAS-Tree* and the memory adaptive mechanisms, our algorithm, called

| | 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 3.1: An example of a data stream of itemset-squences

*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.

5. We conduct extensive experiments and show that *MAHUSP* finds an approximate set of HUSPs over a data stream efficiently and adapts to allocated memory without sacrificing much the quality of discovered HUSPs.

The chapter is organized as follows. Section 3.2 provides relevant definitions and a problem statement. Section 3.3 proposes the data structures and algorithms. Experimental results are shown in Section 3.4. We conclude the chapter in Section 3.5.

## 3.2 Definitions and Problem Statement

Let $I^* = \{I_1, I_2, \cdots, I_N\}$ 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 $\langle X_1, X_2, \cdots, X_Z \rangle$, where $Z$ is the size of $S$. The length of $S$ is defined as $\sum_{i=1}^{Z} |X_i|$. An *L-sequence* is a se-

quence of length $L$. A **sequence database** consists of a set of sequences $\{S_1, S_2, ..., S_K\}$, in which each sequence $S_r$ has a unique sequence identifier $r$ called $SID$ and consists of an ordered list of itemsets $\langle IS_{d_1}, IS_{d_2}, ..., IS_{d_n}\rangle$, where each itemset has a unique global identifier $d_i$ . An itemset $IS_d$ in the sequence $S_r$ is also denoted as $S_r^d$.

In a data stream environment, sequences come continuously over time and they are usually processed in batches. A **sequence batch** (or batch in short) $B_k = \{S_i, S_{i+1}, ..., S_{i+L-1}\}$ 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 $i \neq j$. For example, Figure 3.1 shows a sequence data stream with 2 batches $B_1 = \{S_1, S_2\}$ and $B_2 = \{S_3, S_4, S_5\}$.

**Definition 5** (**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 6** (**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 $\beta$ is a *super-sequence* of $\alpha$ 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 $\beta = \langle\{abc\}\{bce\}\{cd\}\rangle$, $\alpha$ is a sub-sequence of $\beta$

and $\beta$ is the super-sequence of $\alpha$.

**Definition 7** (**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 8** (**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 3.1 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 9** (**Occurrence of a sequence** $\alpha$ **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, $\alpha$ 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* $\alpha$ *in* $S_r$. Since $\alpha$ may have multiple occurrences in $S_r$, the set of all occurrences of $\alpha$ in $S_r$ is denoted as $OccSet(\alpha, S_r)$.

**Definition 10** (**Utility of a sequence** $\alpha$ **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 $\alpha$ 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 $\alpha$ 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 3.1, 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 11 (Utility of a sequence $\alpha$ in a dataset $D$)** The utility of a sequence $\alpha$ in a dataset $D$ of sequences is defined as $su(\alpha, D) = \sum_{S_r \in D} su(\alpha, S_r)$, where $D$ can be a batch or a data stream processed so far.

The **total utility of a batch** $B_k$ is defined as $U_{B_k} = \sum_{S_r \in B_k} su(S_r)$. 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 12 (High utility sequential pattern)** Given a utility threshold $\delta$ in percentage, a sequence $\alpha$ 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 $\delta$ (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.

Given the memory and *one-pass-of-data* constraints, it may not be possible to find the exact set of true HUSPs, especially when the data stream is large. Thus, our goal is to find an accurate approximate set of HUSPs over the data stream.

## 3.3 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 the entire data stream. Below, we first present an overview of *MAHUSP* and then propose a novel tree-based data structure, called *MAS-Tree (Memory Adaptive high utility Sequential pattern Tree)*, to store the essential information of HUSPs over the data stream. Finally, we propose two memory adaptive mechanisms and use them in the tree construction and updating process.

### 3.3.1 Overview of MAHUSP

Algorithm 1 represents an overview of *MAHUSP*. This algorithm processes the data stream batch by batch. Given a utility threshold $\delta$ and a significance threshold $\epsilon$, as a new batch $B_k$ forms, *MAHUSP* first applies an existing HUSP mining algorithm on static

data (e.g., USpan [66])[1] to find a set of HUSPs over $B_k$ using $\epsilon$ as the utility threshold. $\epsilon$ is lower than the utility threshold $\delta$ and specifies a trade-off between accuracy and run time. A smaller $\epsilon$ value leads to more accurate results but longer run time. 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. The *MAS-Tree* contains the potential HUSPs and their approximated utilities over the data stream. Finally, if users request to find HUSPs from the stream so far, *MAHUSP* returns the set of all the patterns (i.e., $appHUSPs$) in *MAS-Tree* with approximate utility more than $(\delta - \epsilon) \cdot U_{DS_k}$, where $DS_k = \langle B_1, B_2, \cdots, B_k \rangle$. In Section 3.3.6, we will explain why we use $(\delta - \epsilon) \cdot U_{DS}$ as the utility threshold.

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

**Algorithm 1** *MAHUSP*

**Input**: $B_k$, $\delta$, $\epsilon$, $availMem$, $mechanismType$

**Output**: *MAS-Tree*, *appHUSPs*

1: $HUSP_{B_k} \leftarrow$ HUSPs returned by $USpan$ on $B_k$ using $\epsilon \cdot U_{B_k}$ as minimum utility threshold

2: **if** MAS-Tree is empty (i.e. $B_k$ is the first batch) **then**

3: Initialize MAS-Tree by creating $root$ node

4: **end if**

5: Call Algorithm 2 to insert the patterns in $HUSP_{B_k}$ into MAS-Tree using $availMem$ and $mechanismType$

6: **if** user requests for HUSPs over current data stream **then**

7: $appHUSPs \leftarrow$ potential HUSPs in MAS-Tree whose approximate utility is no less than $(\delta - \epsilon) \cdot U_{DS}$

8: **end if**

9: **return** *MAS-Tree* and *appHUSPs* if requested

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

### 3.3.2 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 to present

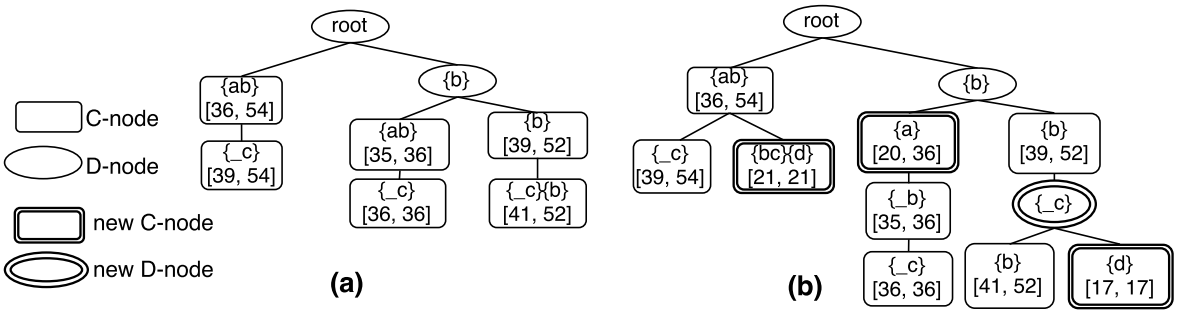



Figure 3.2: (a) An example of *MAS-Tree* for $B_1$ in Figure 3.1. Note that an underscore in a node name $\{\_c\}$ means that the last itemset in the pattern of its parent, such as $\{ab\}$, belongs to the first itemset of the pattern of this node. That is $\{ab\}$ and $\{\_c\}$ forms $\{abc\}$. (b) *MAS-Tree* after inserting three patterns:$\langle\{ab\}\{bc\}\{d\}\rangle$,$\langle\{b\}\{a\}\rangle$ and $\langle\{b\}\{bc\}\{d\}\rangle$.

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

**Definition 13** (**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 $X_1 \lesssim X_2$). Note that items in an itemset are arranged in the lexicographic order.

**Definition 14** (**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 15** (**Prefix sub-sequence and Prefix super-sequence**) Given sequences $\alpha = \langle X_1, X_2, ..., X_i \rangle$ and $\beta = \langle X_1^{'}, X_2^{'}, ..., X_j^{'} \rangle$ $(i \leq j)$, $\alpha$ is a prefix sub-sequence (or *prefixSUB* in short) of $\beta$ or equivalently $\beta$ is a prefix super-sequence (or *prefixSUP* in short) of $\alpha$ 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 16** (**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 $\beta$ w.r.t. $\alpha$.

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

Figure 3.2(a) shows a part of *MAS-Tree* for the potential HUSPs returned by *USpan* in the first batch $B_1$ in Figure 3.1. 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 sub-sequences 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 3.2(a) repre-

senting 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 3.2(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. 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 sub-sequences of potential HUSPs since a subsequence of a HUSP may not be a HUSP.

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 3.2(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.

### 3.3.3 Rest Utility: A Utility Upper Bound

Before we present how a MAS-Tree is built and updated, we first define the *rest utility* of a sequence and prove that it is an upper bound on the true utilities of the sequence and all of its prefix super-sequences ($prefixSUPs$). The rest utilities are stored in the tree and used to prune the tree in the memory adaptation procedure.

**Definition 17 (First occurrences of a sequence $\alpha$ 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 $\alpha$ 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 18 (Rest sequence of $S_r$ w.r.t. sequence $\alpha$)** 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. $\alpha$, 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 $\alpha$ in $S_r$.

**Definition 19 (Rest utility of a sequence $\alpha$ in a sequence $S_r$)** The rest utility of $\alpha$ in $S_r$ is defined as $rsu(\alpha, S_r) = su(\alpha, S_r) + su(restSeq(S_r, \alpha))$.

For example, given sequence $\alpha = \langle\{ac\}\{c\}\rangle$ and $S_1$ in Figure 3.1, $restSeq(S_1, \alpha) = \langle\{(b,1)(c,1)(d,1)\}\{(c,3)(d,1)\}\rangle$. Hence, $su(restSeq(S_1, \alpha)) = 8 + 7 = 15$, then $rsu(\alpha, S_1) = su(\alpha, S_1) + 15 = \max\{7, 9\} + 15 = 24$.

**Definition 20 (Rest utility of a sequence $\alpha$ in dataset $D$)** The rest utility of a sequence $\alpha$ in a dataset $D$ of sequences is defined as $rsu(\alpha, D) = \sum_{S_r \in D} rsu(\alpha, S_r)$.

**Theorem 1** *The rest utility of a sequence $\alpha$ in a data stream $DS$ is an upper-bound of the true utilities of all the $prefixSUPs$ of $\alpha$ 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 $\alpha$ 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 10:

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

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

Sequence $\beta$ can be partitioned into two sub-sequences:

$\alpha = \langle X_1, X_2, ..., X_M\rangle$ and $\beta' = \langle X'_M - X_M, X_{M+1}, ..., X_N\rangle$. The Equation 1 can be

rewritten as follows:

$\exists \tilde{o}_\alpha \in OccSet(\alpha, S_r)$ and $\exists \tilde{o}_{\beta'} \in OccSet(\beta', S_r - \tilde{o}_\alpha)$,

$$su(\beta, S_r) = su(\alpha, \tilde{o}_\alpha) + su(\beta', \tilde{o}_{\beta'}) \tag{2}$$

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

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

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

From (3) and (5):

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

□

### 3.3.4 MAS-Tree Construction and Updating

The tree starts empty. Once a potential HUSP is found in a batch, it is added to the tree. Below, we present how to insert a pattern into the tree in a memory adaptive manner.

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 3.2(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 3.2(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 3.2(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 $D$ correspond-

ing 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 3.2(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 3.2(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 3.2(a).

**Algorithm 2** *Insert potential HUSPs into MAS-Tree* - Part 1

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

**Output**: *MAS-Tree*, *currMem*

1: $newPatSet_{B_k} \leftarrow \emptyset$
2: **for** $\forall S \in HUSP_{B_k}$ **do**
3: $N \leftarrow$ The node with the longest $prefixSUB$ of $S$ in MAS-Tree
4: **if** $S_N$ is the same as $S$ **then**
5: **if** $N$ is C-node **then**
6: $nodeRsu(N) \leftarrow nodeRsu(N) + rsu(S, B_k)$
7: $nodeUtil(N) \leftarrow nodeUtil(N) + su(S, B_k)$
8: **else**
9: Convert $N$ to C-node
10: $nodeRsu(N) \leftarrow rsu(S, B_k) + maxUtil$
11: $nodeUtil(N) \leftarrow su(S, B_k) + maxUtil$
12: **end if**
13: **else**
14: Add pair $\langle S, N \rangle$ to $newPatSet_{B_k}$
15: **end if**
16: **end for**

*Insert potential HUSPs into MAS-Tree - Part 2*

17: **for** $\forall\langle S, N\rangle \in newPatSet_{B_k}$ **do**

18: $CN \leftarrow$ A child of node $N$ with longer common prefixSUB (i.e.,$S_{prefix}$) of $S$ than $S_N$

19: **if** $CN$ does not exist **then**

20: $S_C \leftarrow$ suffix of $S$ w.r.t. $S_N$

21: Call **Algorithm 3** to create C-node $C$ as a child of $N$ using $S_C$,$rsu(S, B_k)$, $su(S, B_k)$

22: **else**

23: **if** $S_{prefix}$ is $S$ **then**

24: $S_C \leftarrow$ suffix of $S$ w.r.t. $S_N$

25: Call **Algorithm 3** to create C-node $C$ as a child of $N$ using $S_C$, $rsu(S, B_k)$, $su(S, B_k)$

26: Assign $C$ as parent of $CN$ and update $nodeName(CN)$ w.r.t. $nodeName(C)$

27: **else**

28: $S_D \leftarrow$ suffix of $S_{prefix}$ w.r.t. $S_N$

29: Call **Algorithm 3** to create D-node $D$ as a child of $N$ using $S_D$

30: $S_C \leftarrow$ suffix of $S$ w.r.t. $S_{prefix}$

31: Call **Algorithm 3** to create C-node $C$ as a child of $D$ using $S_C$, $rsu(S, B_k)$, $su(S, B_k)$

32: Assign $D$ as parent of $CN$ and update $nodeName(CN)$ w.r.t. $nodeName(D)$

33: **end if**

34: **end if**

35: **end for**

36: **return** *MAS-Tree*

Algorithm 2 shows the complete procedure for inserting the potential HUSPs found in batch $B_k$ to the tree. It first updates the tree using the patterns in $HUSP_{B_k}$ that

already exist in the tree. This is to avoid the memory adaption procedure from pruning nodes that will be inserted again soon in the same batch. For each pattern $S$ in $HUSP_{B_k}$, Algorithm 2 finds node $N$ where $S_N$ is either $S$ or the longest ***prefixSUB*** of $S$ in the tree. If $S_N$ is $S$, lines 5 to 12 update 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.

### 3.3.5 Memory Adaptive Mechanisms

*MAS-Tree* adapts its memory usage to the available memory. When inserting a new node in the 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[2] 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

[2]A potential HUSP with higher likelihood to become a HUSP has higher quality.

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

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

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

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

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

**Rationale:** Since in *MAS-Tree* a descendant of a node $N$ represents a prefix super-sequence ($prefixSUP$) of the pattern represented by $N$, according to Theorem 1, the rest utility of $N$ ($nodeRsu(N)$) is an upper bound of the true utilities of all its descen-

dants. 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 (which would be needed if only a single non-leaf node is removed).

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

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

**Algorithm 3** *Memory Adaptive Node Creation* - Part 1

**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: Find the leaf node $node$ with minimum $nodeUtil(node)$ and remove it from the tree

6: $maxUtil \leftarrow nodeUtil(node)$;

7: $relMem \leftarrow$ memory released by pruning the $node$

8: $currMem \leftarrow currMem - relMem$

9: **end if**

10: **if** mechanismType is **SBMA then**

11: Find $node$ with minimum $nodeRsu(node)$ and remove the subtree rooted by $node$

12: $maxUtil \leftarrow nodeRsu(node)$

13: $relMem \leftarrow$ memory released by pruning the subtree

14: $currMem \leftarrow currMem - relMem$

15: **end if**

16: **if** parent $P$ of $node$ has a single child $C$ **then**

17: Merge $P$ and $C$ into a D-node if $C$ is a D-node or a C-node if $C$ is a C-node

18: Adjust the amount of current available memory $currMem$

19: **end if**

20: **end while**

*Memory Adaptive Node Creation* - Part 2

21: **if** parent $P$ of $node$ has been removed **then**

22: Call Algorithm 2 to get a new parent node and exit this procedure

23: **end if**

24: Create node $N$ with pattern $S$

25: **if** $nodeType$ is a C-node **then**

26: $nodeRsu(N) \leftarrow rsu(S, B_k) + maxUtil$

27: $nodeUtil(N) \leftarrow su(S, B_k) + maxUtil$

28: **end if**

29: currMem $\leftarrow currMem + reqMem$

30: **return** $N, maxUtil$

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 leaf nodes can be stored in a sorted list. Thus, a direct access can be done to find a leaf node with minimum utility. The time complexity for initializing and updating this list is $O(NumPot)$. The time complexity to apply $SBMA$ is $O(NumPot \times L_{avg})$. The time complexity to create the new node is $O(L_{avg})$.

### 3.3.6 Mining HUSPs from MAS-Tree

As the data stream evolves, when the user requests to find HUSPs on the stream so far, MAHUSP traverses the MAS-Tree once and returns all the patterns represented by a node whose $nodeUtil$ is no less than $(\delta - \epsilon) \cdot U_{DS_k}$, where $DS_k = \langle B_1, B_2, \cdots, B_k \rangle$ 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 $nodeUtil(N) \geq (\delta - \epsilon) \cdot U_{DS_k}$.

### 3.3.7 Correctness

Given a data stream $DS$, a sequence $\alpha$ and a node $N \in$ ***MAS-Tree*** where $S_N$ is $\alpha$, 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 1** Given a potential HUSP $\alpha$, the difference between the exact utility of $\alpha$ 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 11, $su(\alpha, DS) = \sum_{B_j \in DS} su(\alpha, B_j)$. Given batch $B_k \in DS$, if $su(\alpha, B_k) < \epsilon \cdot U_{B_k}$, then $\alpha$ is not returned by ***USpan***. In this case $\epsilon \cdot U_{B_k}$ is an upper bound on utility of $\alpha$ 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 $\alpha$ is not returned by *USpan*.

□

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

**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 $\alpha$ 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 $\alpha$ was re-inserted, the value of $maxUtil$ was denoted as $maxUtil_2$. According to Algorithm 2, $maxUtil_1 \leq maxUtil_2$. Once $C$ was re-inserted into the tree, $nodeUtil(C)$ was incremented by $maxUtil_2$ which is bigger than or equal to $maxUtil_1$. Since $su_{apprx}(\alpha, DS) = nodeUtil(C)$, $su_{apprx}(\alpha, DS) \geq su_{tree}(\alpha, DS)$.

□

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

**Proof**

We prove it by contradiction. Assume that there is a HUSP $\beta$, 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, $\beta$ 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 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 $maxUtil < su_{tree}(\alpha, DS)$, $\alpha$ is in the tree.

□

**Theorem 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 $\alpha$. According to Definition 11, $su(\alpha, DS) \geq \delta \cdot U_{DS}$. On the other hand, based on Lemma 1, $\epsilon \cdot U_{DS} > su(\alpha, DS) - su_{tree}(\alpha, DS)$. Thus, $\epsilon \cdot U_{DS} + su_{tree}(\alpha, DS) > su(\alpha, DS)$.

According to Definition 11, $\epsilon \cdot U_{DS} + su_{tree}(\alpha, DS) > \delta \cdot U_{DS}$. Hence, $su_{tree}(\alpha, DS) > (\delta - \epsilon) \cdot U_{DS} \geq maxUtil$.

According to Lemma 3, $\alpha$ must exist in the tree. On the other hand, based on Lemma

2, $su_{apprx}(\alpha, DS) \geq su_{tree}(\alpha, DS) > (\delta - \epsilon) \cdot U_{DS}$. Hence, $\alpha$ will be returned by the algorithm.

□

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

## 3.4 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*)[2], and the real-world *Kosarak* dataset [26]. The synthetic datasets are generated by IBM data generator [2]. The parameters in DS1(DS2) mean that the number of sequences in the dataset is 10K(100K), the average number of transactions in a sequence is 10(8), the average number of items in a transaction is 3(3), the average length of a maximal pattern consists of 4(4) itemsets and each itemset is composed of 2(2) items average. The number of items in the dataset is 1k(10k). The *Kosarak* dataset contains web click-stream data of a Hungarian on-line news portal. Table 3.1 shows dataset characteristics and parameter settings in the experiments. The last column shows the available memory (i.e., $availiMem$) assigned to the mining task. We set $availMem$ heuristically based on the average memory to store the data structures

Table 3.1: Dataset characteristics

| **Name** | **#Seq** | **#Item** | **Type** | **batchSize** | **availMem** |
|---|---|---|---|---|---|
| DS1 | 10K | 1K | Dense | 1K | 100MB |
| Kosarak | 25K | 15K | Sparse, Large | 5K | 200MB |
| DS2 | 100K | 1K | Dense, Large | 10K | 400MB |

used by *USpan* and the average memory used by *MAS-Tree* over the datasets. We follow a previous study [4] to generate internal and external utilities of items in the datasets. The external utility of each item is generated between 1 and 100 by using a log-normal distribution and the internal utilities of items in a transaction are randomly generated between 1 and 100. The significance threshold (i.e., $\epsilon$) is set as $0.5 \times \delta$. For example, in DS2, when $\delta = 0.0009$, $\epsilon = 0.00045$. We will later change these parameters (i.e., $availMem$, $batchSize$ and significance threshold) to show the performance of the algorithm under different parameter values.

The $batchSize$ column shows the number of sequences in each batch during the evaluation. We will also investigate the effect of different sizes of batch on performance of the method. The experiments are conducted on an Intel(R) Core(TM) i7 2.80 GHz computer with 4 GB of RAM.

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-Measure***: $2 \times \frac{precision \times recall}{precision + recall}$. (3) ***Run Time***: the total execution time of a method over data streams, (4) ***Memory Usage***: the memory consumption of a method.

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 $\alpha$, the utility of $\alpha$ 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 sub-tree 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).

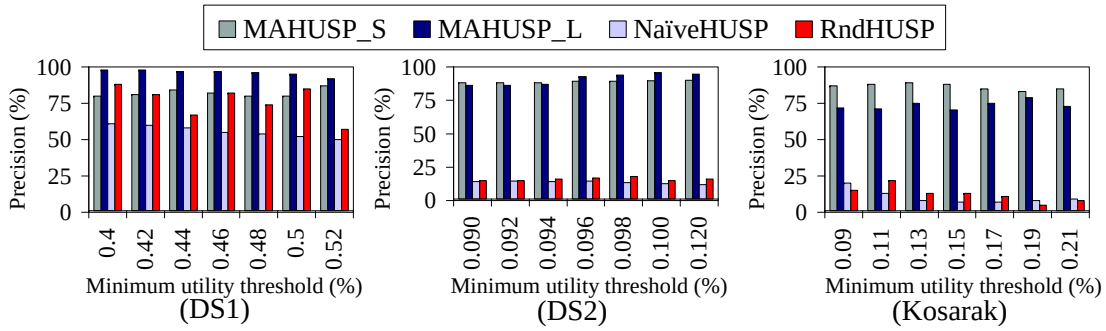



Figure 3.3: Precision performance on the different datasets

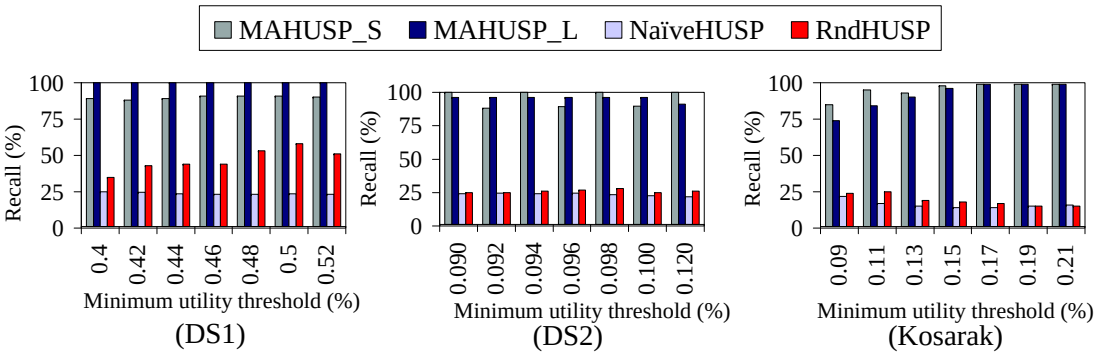



Figure 3.4: Recall performance on the different datasets

### 3.4.1 Effectiveness of MAHUSP

In this section, the effectiveness of the methods is evaluated. Figure 3.3, Figure 3.4 and Figure 3.5 show the results in terms of *Precision*, *Recall* and *F-Measure* on the three datasets respectively. For consistency across datasets, the minimum threshold is shown as a percentage (i.e., $\delta$) of the total utility of the current data stream in the dataset.

Figure 3.3 shows the *precisions* of the methods on the three datasets. The proposed methods outperform *NaiveHUSP* and *RndHUSP* significantly. *MAHUSP_L* outperforms

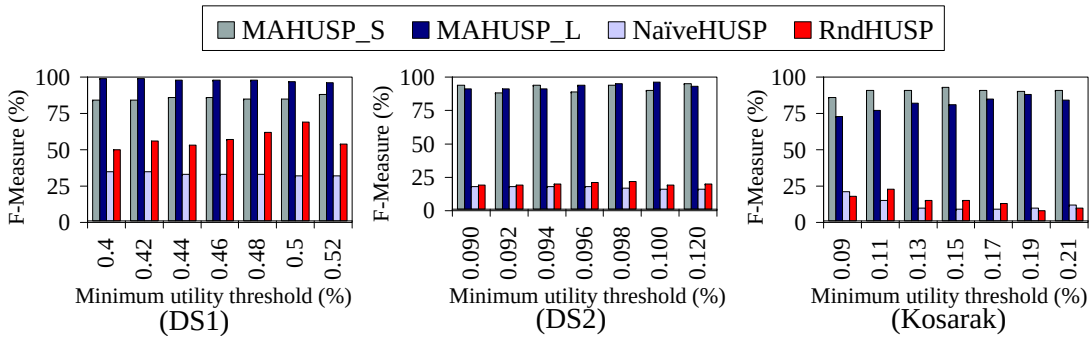



Figure 3.5: F-Measure performance on the different datasets

*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. However, *MAHUSP_S* performs better on the sparse dataset $Kosarak$.

Figure 3.4 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 2 happens often and the proposed memory adaptive mechanisms prune the nodes effectively.

Figure 3.5 shows the *F-Measure* values for the four methods with different $\delta$ values on the 3 datasets. In most of the cases, *NaiveHUSP* is the worst among the four methods. This is because it estimates the utility of a sequence based on the utility of each item

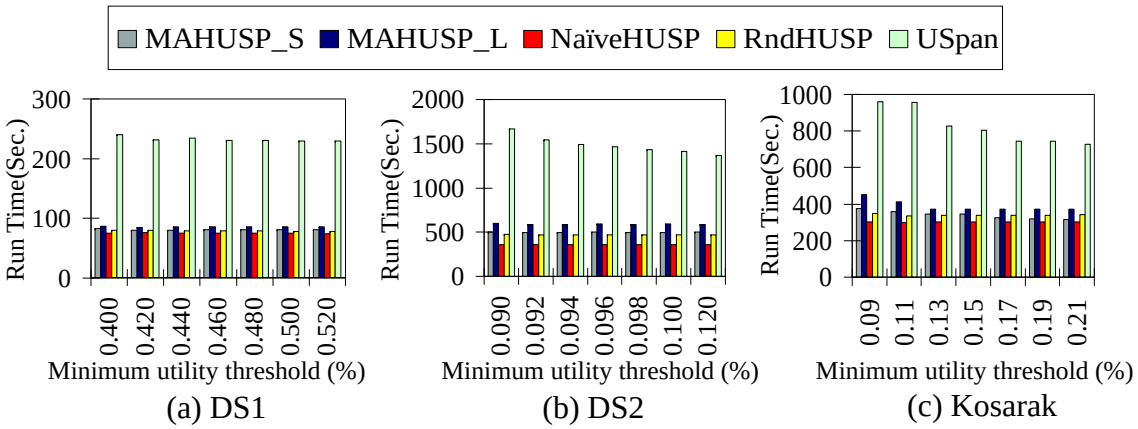



Figure 3.6: Execution time on different datasets.

over the data stream which is not an accurate approximation. Both proposed methods outperform the other methods significantly with an average F-Measure value of 90% over the $DS1$, $DS2$ and $Kosarak$ data sets.

### 3.4.2 Time and Memory Efficiency of MAHUSP

Figure 3.6 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 random pruning method

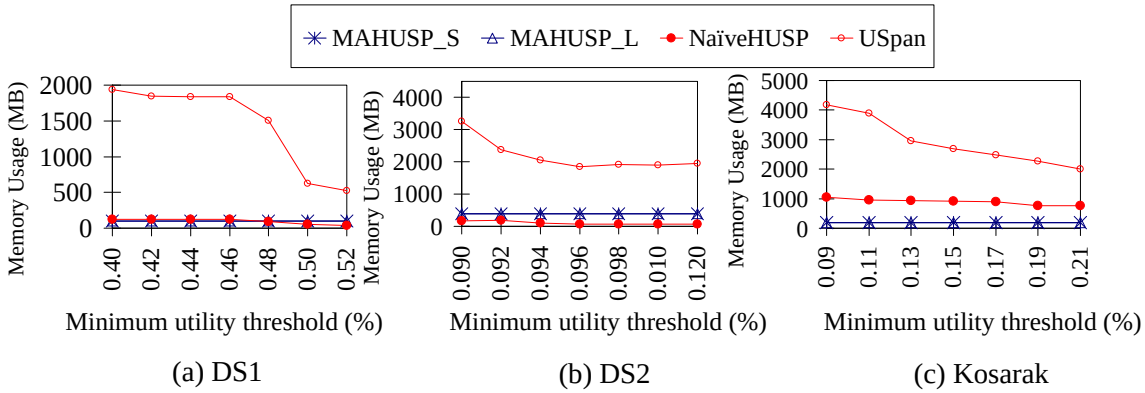



Figure 3.7: Memory Usage on different datasets.

($RndHUSP$). Considering the big difference between them in precision and recall, it is very worthwhile to use the pruning strategies proposed in this work.

Figure 3.7 shows the memory consumption of the methods on different datasets for different values of $\delta$. Since *RndHUSP*, *MAHUSP_L* and *MAHUSP_S* consume the same amount of memory, we only present the results of *MAHUSP_S*, *NaiveHUSP* and *USpan*. *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, when the threshold increases on $DS2$, since the number of promising items decreases, the memory usage decreases. *NaiveHUSP* uses less memory than *MAHUSP_S* on $DS2$, since $DS2$ is a dense dataset. *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* guarantees that memory usage is

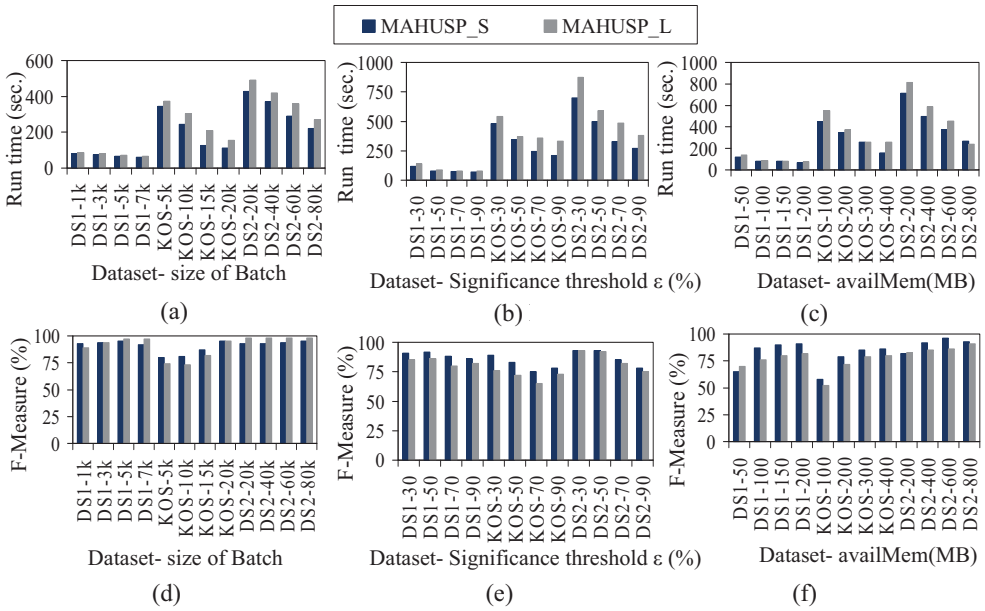



Figure 3.8: Parameter sensitivity on different datasets.

bounded by the given input parameter *availMem*.

### 3.4.3 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 ($\epsilon$) and the amount of available memory ($availMem$). In all the experiments, $\delta$ is set to 0.46%, 0.096%, 0.15% for $DS1$, $DS2$ and $Kosarak$ respectively. Figures 3.8(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 3.8 (a) shows the trend in the execution time with

different batch sizes. In all the datasets, the run time decreases as $batchSize$ increases since increasing the batch size leads to generating less number of intermediate potential HUSPs. Figure 3.8 (d) shows F-Measures on different datasets. From Figure 3.8(d), we can observe that the F-Measure of the methods increases slowly with increasing batch sizes.

Figures 3.8 (b)(e) show the results on *Run time* and *F-Measure* for different values of $\epsilon$. Each bar in the graphs is assigned to each dataset and value of $\epsilon$ is a percentage of $\delta$. As it is observed, a higher value of $\epsilon$ leads to a lower number of HUSPs returned by *USpan* in each batch and thus the *F-Measure* value decreases. On the other hand, when the value of $\epsilon$ increases the processing time decreases since the number of HUSPs returned by USpan decreases.

Figures 3.8(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 3.8(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 3.8 (f) shows the results on F-Measure. When the available memory is small (e.g., 50 MB in DS1), there are fewer HUSPs in the memory and usually F-Measure is lower. However, after a certain value of $availMem$, the performance of the proposed methods is much

higher.

## 3.5 Summary

In this chapter, we tackled the problem of mining HUSPs over the entire of a data stream and proposed a *memory-adaptive approach* to finding 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.

1. **Compact data structure**: we proposed a novel and compact data structure, called *MAS-Tree*, to store potential HUSPs over 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. 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 sub-sequences of potential HUSPs since a subsequence of a HUSP may not be a HUSP.

2. **Memory adaptive mechanisms**: in data stream mining, data can be huge so that the amount of information we need to keep may exceed the size of available mem-

ory. 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. Hence, two efficient memory adaptive mechanisms are proposed to deal with the situation when the available memory is not enough to add a new potential HUSPs to *MAS-Tree*. The proposed mechanisms choose the least promising patterns to remove from the tree to guarantee that the memory constraint is satisfied.

3. **Single pass algorithm**: in data stream mining, data should be usually processed by an online algorithm whose workspace is insufficient to store all the data, so the algorithm must process and then discard each data element. Using *MAS-Tree* and the memory adaptive mechanisms, we proposed a novel single pass algorithm for incrementally mining HUSPs over a data stream. 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.

4. **Extensive experiments**: we conducted extensive experiments and show that *MAHUSP* finds an approximate set of HUSPs over a data stream efficiently and adapts to allocated memory without sacrificing much the quality of discovered HUSPs.

# 4 Sliding Window-based High Utility Sequential Pattern Mining over Data Streams

## 4.1 Introduction

In some applications such as *network traffic monitoring* and *intrusion detection*, users are more interested in the information that reflect recent data rather than old ones. The most common data stream processing model to discover recent 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 were already $w$ data records in the window, the oldest one is removed from the window. In this model, the effect of the expired old data is eliminated, and the patterns are mined from the data in the current window.

Some studies have been conducted on efficiently mining *frequent* sequences over data streams using the sliding window model (See Chapter 2, subsection 2.1.3). However, ex-

isting methods have the following deficiencies. (1) They are frequency-based, and did not consider the utility (e.g., value) of an item and thus cannot be used to find HUSPs over sliding windows. (2) Most of the studies such as [19, 46] 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 [32]. 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.

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 sliding window-based HUSP mining is very desirable in many real-life applications such as *online user behavior analysis* and *web mining*, addressing this topic is not an easy task due to the following challenges.

- Effectively pruning search space for sliding window-based mining high utility se-

quential patterns over a data stream is difficult, because the *downward closure property* does not hold for the utility of sequences.

- Mining HUSPs over a data stream of itemset-sequence need to overcome the large search space problem due to combinatorial explosion of 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, unboundedly 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 sliding windows by reading data records only once 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 sliding window is a challenging problem.

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

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

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 in a transaction-sensitive sliding window over a data stream. To the best of our knowledge, 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 updates when the window slides, and can be used to compute sequence utilities without re-scanning the transactions in the sliding window.

3. We also 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 [4] for HUSP mining.

4. We propose a new one-pass algorithm called *HUSP-Stream (High Utility Sequential Pattern Mining over Data Streams)* for efficiently constructing and updating *ItemUtilLists* and *HUSP-Tree* by reading a transaction in the data stream *only once*, and by making use of both $SFU$ and $SWU$ to prune the size of *HUSP-Tree*. 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 updated databases. 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.

5. We conduct extensive experiments on both real and synthetic datasets to evaluate the performance of the proposed algorithm. Experimental results show that HUSP-Stream outperforms the state-of-the-art HUSP mining algorithm 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 [66], a

Table 4.1: Summary of Notations

| **Notation** | **Description** |
|---|---|
| $u(X, S_r^d)$ | Utility of item/itemset $X$ in transaction $T_d$ of $S_r$ |
| $TU(S_r^d)$ | Utility of transaction $T_d$ of sequence $S_r$ |
| $\alpha \preceq \beta$ | $\alpha$ is a subsequence of $\beta$, or $\alpha$ occurs in $\beta$ |
| $OccSet(\alpha, S_r)$ | Set of all the occurrences of $\alpha$ in sequence $S_r$ |
| $su(\alpha, S_r)$ | Utility of a sequence $\alpha$ in sequence $S_r$ |
| $\alpha \oplus I$ | Itemset-extended of sequence $\alpha$ and item $I$ |
| $\alpha \otimes I$ | Sequence-extended of sequence $\alpha$ and itemset $\{I\}$ |
| $TSWU(\alpha, SW_i)$ | Sequence weighted utility of sequence $\alpha$ in $SW_i$ |
| $suffix(S_r, \alpha)$ | Suffix of sequence $S_r$ w.r.t. sequence $\alpha$ |
| $SFU(\alpha, SW_i)$ | Sequence-suffix utility of sequence $\alpha$ in $SW_i$ |

state-of-the-art HUSP mining algorithm, fails to complete the mining task.

The remaining of the chapter is organized as follows. Section 4.2 provides definitions and a problem statement. Section 4.3 presents the proposed algorithms and data structures. Experimental results are shown in Section 4.4. We conclude the chapter in Section 4.5.

**(a) A Data Stream of Itemset-Sequences**

| SID \ TID | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | ... |
|---|---|---|---|---|---|---|---|---|
| $S_1$ | {(a,2)(b,3)(c,2)} | {(b,1)(c,1)(d,1)} | | {(c,3)(d,1)} | | | {(b,1)(c,3)(d,2)} | ... |
| $S_2$ | | | {(b,4)} | | {(a,4)(b,5)(c,1)} | | | ... |
| $S_3$ | | | | | | {(a,2)(b,5)(e,2)} | | ... |

$SW_1$

$SW_2$

**(b) Profit Table**

| **Item** | **a** | **b** | **c** | **d** | **e** |
|---|---|---|---|---|---|
| **Profit** | 2 | 3 | 1 | 4 | 3 |

Figure 4.1: (a) An example of a data stream of itemset-squences and transaction-sensitive sliding windows over the data stream, (b) an example of external utility table

## 4.2 Definitions and Problem Statement

This section presents definitions and defines the problem of sliding window-based high utility sequential pattern mining over data streams. For more details about preliminaries, readers can refer to definitions in Chapter 3.

Figure 4.1 shows a data stream $DS = \langle S_1^1, S_1^2, S_2^3, S_1^4 S_2^5, S_3^6, S_1^7 \rangle$ with 7 transactions, each belonging to one of three sequences: $S_1, S_2$ and $S_3$.

In many applications, such as customer purchase sequence mining, a new transaction may belong to an existing sequence. In a data stream environment, transactions come continually over time, and they are usually processed in *batches*. A **trans-**

**action batch** $B_i$ consists of transactions arriving continuously in a time period, i.e., $B_i = \{T_j, T_{j+1}, ..., T_k\}$. A **sliding window** consists of $w$ recent batches where $w$ is the size of the window. If the first batch in a sliding window is $B_i$, the window can be presented as $SW_i = \{B_i, B_{i+1}, ..., B_{i+w-1}\}$. The sliding window is called *transaction-sensitive sliding window* if each batch consists of one transaction (i.e., $B_i = \{T_i\}$). Therefore, when a new transaction arrives, the oldest one is removed from $SW$. Consequently, the *i-th* transaction-sensitive sliding window over $DS$ is equivalently defined as $SW_i = \langle T_i, T_{i+1}, ..., T_{i+w-1} \rangle$. Note that, in this chapter, a *transaction-sensitive sliding window* is referred as *sliding window*.

Transactions in a sliding window can belong to different sequences[3]. For example, in Figure 4.1, if the window size $w$ is set to 5, the first and the second windows over $DS$ are $SW_1 = \langle S_1^1, S_1^2, S_2^3, S_1^4, S_2^5 >$ (which has 2 sequences) and $SW_2 = \langle S_1^2, S_2^3, S_1^4, S_2^5, S_3^6 \rangle$ (which has 3 sequences), respectively.

**Definition 21** (**Utility of an itemset in a transaction**) Given itemset $X \subseteq T_d$, the utility of $X$ in the transaction $T_d$ of the sequence $S_r$ is defined as $u(X, S_r^d) = \sum_{I \in X} u(I, S_r^d)$

**Definition 22 (Transaction utility)** The transaction utility of transaction $S_r^d \in DS$ is denoted as $TU(S_r^d)$ and computed as $u(S_r^d, S_r^d)$.

For example, $u(b, S_1^1) = p(b) \times q(b, S_1^1) = 3 \times 3 = 9$, and $u(\{bc\}, S_1^1) = u(b, S_1^1) +$

[3]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 the all transactions arrived in a time interval.

$u(c, S_1^1) = 9 + 2 = 11$. Therefore, *transaction utility* of $S_1^1$ is $TU(S_1^1) = 2 \times 2 + 3 \times 3 + 1 \times 2 = 15$.

**Definition 23 (Utility of a sequence $\alpha$ in a sequence $S_r$)** Let $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z} \rangle$ be an occurrence of $\alpha = \langle X_1, X_2, ..., X_Z \rangle$ 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 $su(\alpha, S_r) = \max\{su(\alpha, \tilde{o}) | \forall \tilde{o} \in OccSet(\alpha, S_r)\}$.

**Definition 24 (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 $su(\alpha, SW_i) = \sum_{S_r \in SW_i} su(\alpha, S_r)$.

For example, let $\alpha = \langle \{ab\}\{c\} \rangle$. In $SW_1$ of Figure 4.1, $OccSet(\alpha, S_1) = \{\langle S_1^1, S_1^2 \rangle, \langle S_1^1, S_1^4 \rangle\}$. The utility of $\alpha$ in $S_1$ is $su(\alpha, S_1) = \max\{su(\alpha, \langle S_1^1, S_1^2 \rangle), su(\alpha, \langle S_1^1, S_1^4 \rangle)\} = \max\{14, 16\} = 16$. 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 25 (High utility sequential pattern (HUSP))** 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 $\delta$.

**Problem statement.** Given a minimum utility threshold $\delta$, the problem of *sliding window-based high utility sequential pattern mining over a transaction data stream* $DS$ is to discover all sequences of itemsets whose utility is no less than $\delta$ from the current sliding window over $DS$.

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 and in-memory data structures are proposed based on this concept, which are presented in the next section.

For convenience, Table 4.1 summarizes the concepts and notations we define in this chapter.

## 4.3 Sliding Window-based High Utility Sequential Pattern Mining over Data Streams

In this section, we propose a single-pass algorithm named *HUSP-Stream (High Utility Sequential Pattern mining over data Stream)* for incrementally mining the complete set of HUSPs in the current window $SW_i$ of a data stream based on the previous mining results for $SW_{i-1}$. We propose a vertical representation of the dataset called *ItemUtilLists (Item Utility Lists)* and a tree-based data structure, called *HUSP-Tree (High Utility Sequential Pattern Tree)*, to model the essential information of HUSPs in the current window.

The overview of *HUSP-Stream* is presented in Algorithm 4. The algorithm includes three main phases: (1) *Initialization phase*, (2) *Update phase* and (3) *HUSP mining phase*. The initialization phase applies 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 $S_r^i$. 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 $S_r^i$ arrives, $S_r^i$ 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 6, which is the *update phase* of the algorithm. 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.

**Algorithm 4** *HUSP-Stream*

**Input**: a new transaction $S_r^i$, window size $w$, minimum utility threshold $\delta$, ItemUtilLists, HUSP-Tree

**Output**: ItemUtilLists, HUSP-Tree, HUSPs

1: **if** $i \leq w$ (*when $S_r^i$ is a transaction in the first window)* **then**
2: $\forall$ item $\in S_r^i$, $put(r, i, u(item, S_r^i))$ to $ItemUtilLists(item)$
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 $S_r^i$, $w$ 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 $ItemUtilLists$, $HUSP-Tree$, $HUSPs$ if requested

### 4.3.1 Initialization phase

In this phase, HUSP-Stream reads the transactions in the first sliding window one by one to construct *ItemUtilLists* and *HUSP-Tree*. Below we first introduce these two structures and then explain how to construct them in the initialization phase.

#### 4.3.1.1 ItemUtilLists (Item Utility Lists)

The first component of the proposed algorithm is an effective representation of items to restrict the number of candidates and to reduce the processing time and memory usage. *ItemUtilLists* is a vertical representation of the transactions in the sliding window. The *ItemUtilLists* of an item $I$ consists of several tuples. Each tuple stores the utility of item $I$ in the transaction $S_v^u$ (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 $S_v^u$ (Definition 7 in Chapter 3). Figure 4.2 shows *ItemUtilLists* for the first sliding window $SW_1$ in Figure 4.1, which is computed easily by using the external utility of $I$ and the internal utility of $I$ in $S_v^u$. $ItemUtilLists$ 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 $ItemUtilLists$ 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 initializing $ItemUtilLists$ is $O(w \times L_{avg})$, where $w$ is the window size and $L_{avg}$ is the average length of these transactions.

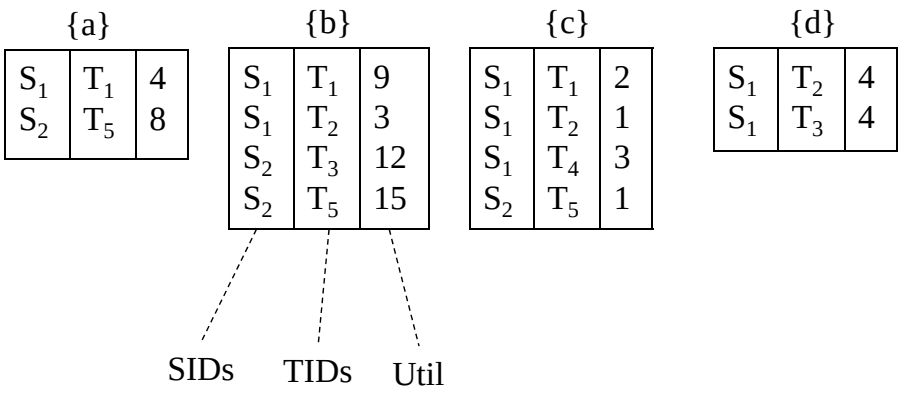



Figure 4.2: *ItemUtilLists* for items in $SW_1$ in Figure 4.1

#### 4.3.1.2 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 in an online fashion. The structure of *HUSP-Tree* is similar to the lexicographic tree proposed in [32] and *LQS-Tree* [66]. 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 in each 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 [32] is used to find frequent sequential patterns, hence a node in this tree mainly stores the frequency of a sequence represented by the node. The *LQS-Tree* is for finding high utility sequential patterns in a static database, and thus its node contains information about the utility of a sequence. However, these trees were not designed for HUSP mining over data streams,

and thus their node structure and content do not allow online updates of sequence utilities and do not support incremental mining of HUSPs over sliding windows.

A HUSP-Tree is a lexicographic sequence tree where each non-root node represents a sequence of itemsets. Figure 4.3 shows part of the HUSP-Tree for the first window $SW_1$ in Figure 4.1, 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 nodes of a parent are listed in alphabetic order of their represented sequences. There are two types of child nodes for a parent: *I-node* and *S-node*, which are defined as follows.

**Definition 26 (Itemset-extended node (I-node))** Given a parent node $p$ representing a sequence $\alpha$, an *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 $\alpha \oplus I$).

**Definition 27 (Sequence-extended node (S-node))** Given a parent node $p$ representing a sequence $\alpha$, an *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 $\alpha \otimes I$).

In Figure 4.3, the node for sequence $\langle\{abc\}\rangle$ is an *I-node*, while the node for $\langle\{ab\}\{c\}\rangle$ is an *S-node*. Their parents are $\{ab\}$,

In sliding window-based data stream mining, a data structure is needed to address two main issues. First, it is not possible to re-construct a tree when a new transaction

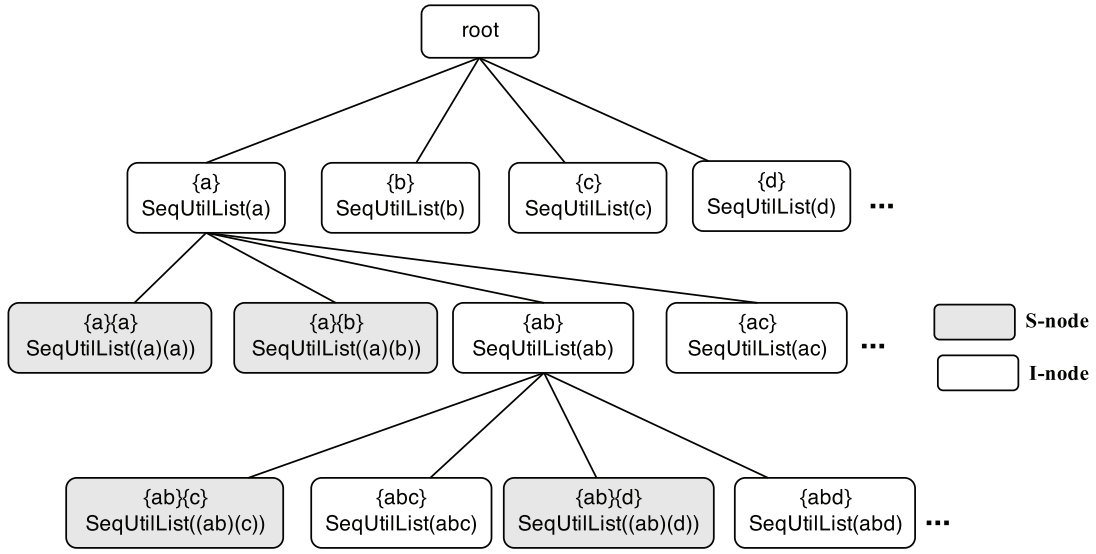



Figure 4.3: An Example of HUSP-Tree for $SW_1$ in Figure 4.1

arrives at or leaves from the window and the data structure should be able to update itself efficiently. Hence, *LQS-Tree* is not applicable here since it is not able to update the contents in the nodes when the window slides. 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 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 HUSPs to prune the tree during tree construction and updating, and also to compute the exact utility of potential HUSPs during the HUSP mining phase. Hence, we design each non-root node of a HUSP-Tree to have a field, called *SeqUtilList*, for storing the needed information about the sequence

represented by the node.

**Definition 28** (**Sequence Utility List**) The sequence utility list (*SeqUtilList*) of a sequence $\alpha$ in sliding window $SW_i$ is a list of 3-value tuples, where each tuple $\langle SID, TID, Util \rangle$ represents an occurrence of $\alpha$ in the sequences of $SW_i$ 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 4.1, since $\alpha$ has two occurrences in $SW_1$, which are $\langle T_1, T_2 \rangle$ and $\langle T_1, T_4 \rangle$, the $SeqUtilList$ of $\alpha$ in $SW_1$ is $\{\langle S_1, T_2, (4+1) \rangle, \langle S_1, T_4, (4+3) \rangle\} = \{\langle S_1, T_2, 5 \rangle, \langle S_1, T_4, 7 \rangle\}$.

#### 4.3.1.3 Major Steps in HUSP-Tree Construction

When the first sliding window becomes full, 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 *ItemUtilLists* 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 26). We define *I-Set* of $\alpha$ as the set of items occurring in the sliding window (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 $Tp = \langle s, t, u \rangle$ in $SeqUtilList(\alpha)$, if there is a tuple $Tp' = \langle s', t', u' \rangle$ in $ItemUtilLists(I)$ 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 $ItemUtilLists(b) = \{\langle S_1, T_1, 9 \rangle, \langle S_1, T_2, 3 \rangle, \langle S_2, T_3, 12 \rangle, \langle S_2, T_5, 15 \rangle\}$, 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 $\beta$ be $\alpha \otimes I$ (i.e., a sequence by adding itemset $\{I\}$ to the end of $\alpha$). First, $SeqUtilList(\beta)$ is initialized to empty. For each tuple $Tp = \langle s, t, u \rangle$ in $SeqUtilList(\alpha)$, if there is a tuple $Tp' = \langle s', t', u' \rangle$ in $ItemUtilLists(I)$ 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

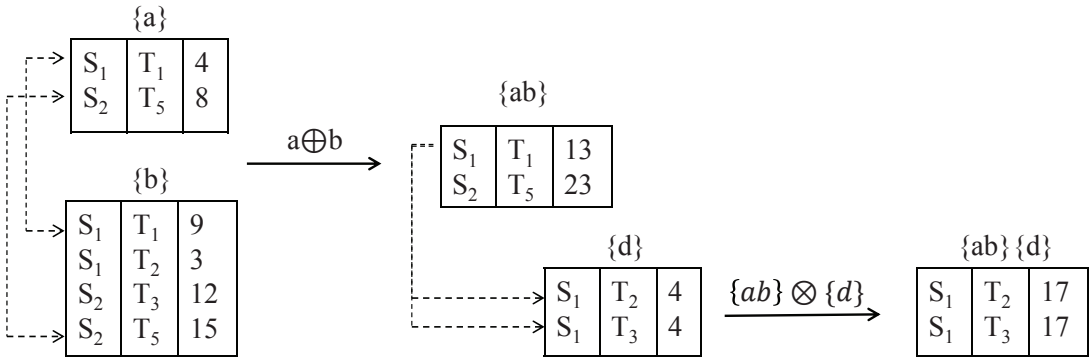



Figure 4.4: I-Step and S-Step to construct *SeqUtilLists* of the sequences $\langle\{ab\}\rangle$ and $\langle\{ab\}\{d\}\rangle$.

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 $ItemUtilLists(d) = \{\langle S_1, T_2, 4\rangle, \langle S_1, T_3, 4\rangle\}$. 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 $\langle S_1, T_3, 4\rangle$ in *ItemUtilLists(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 4.4 shows the details of **I-Step** and **S-Step** to construct *SeqUtilLists* of $\langle\{ab\}\rangle$ and $\langle\{ab\}\{d\}\rangle$.

A complete description of the tree construction process will be given below after our pruning strategies are presented.

#### 4.3.1.4 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 [4] 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 29** 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 $SW_i$:

$$TSWU(\alpha, SW_i) = \sum_{S \in SW_i \wedge \alpha \preceq S} \sum_{T \in S} TU(T)$$

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

For example, in $SW_1$ in Figure 4.1, there are two sequences $S_1$ and $S_2$ contain the sequence $\langle\{b\}\{c\}\rangle$. The $TSWU$ of $\langle\{b\}\{c\}\rangle$ in $SW_1$ is $TSWU(\langle\{b\}\{c\}\rangle, SW_i)$= (15+8+7) + (12+24) = 66.

Since it uses the utilities of all the transactions of all the sequences containing $\alpha$ in $SW_i$, SWU of sequence $\alpha$ is an over-estimate of the utility of $\alpha$ (i.e., Definition 24).

That is, $SWU(\alpha, SW_i) \geq su(\alpha, SW_i)$. The theorem below states that TSWU has the downward closure property over a sliding window.

**Theorem 3** *Given a sliding window $SW_i$ and two sequences $\alpha$ and $\beta$ such that $\alpha \preceq \beta$,* $TSWU(\alpha, SW_i) \geq TSWU(\beta, SW_i)$.

**Proof**

Let $DS_\alpha$ be the set of sequences containing $\alpha$ in $SW_i$ and $DS_\beta$ be the set of sequences containing $\beta$ in $SW_i$. 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 29 $TSWU(\alpha, SW_i) \geq TSWU(\beta, SW_i)$.

□

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 $TSWU(\alpha, SW_i) < \delta$, 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 $su(\beta, SW_i) \leq TSWU(\beta, SW_i) \leq TSWU(\alpha, SW_i) < \delta$, 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 30 (First occurrence of sequence $\alpha$ in sequence $S_r$)** Let $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z} \rangle$ be an occurrence of a sequence $\alpha$ in the sequence $S_r$. $\tilde{o}$ is called the first occurrence of $\alpha$ in $S_r$ if the last transaction in $\tilde{o}$ (i.e., $T_{e_Z}$) occurs before the last transaction of all occurrences in $OccSet(\alpha, S_r)$.

For example, there are two occurrences of $\{a\}\{c\}$ in $S_1$ in $SW_1$ in Figure 4.1: $\langle T_1, T_2 \rangle$ and $\langle T_1, T_4 \rangle$. $\langle T_1, T_2 \rangle$ is the first occurrence because $T_2$ occurs earlier than $T_4$.

**Definition 31 (Suffix of a sequence $S_r$ w.r.t. a sequence $\alpha$)** Given sequence $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z} \rangle$ as the first occurrence of $\alpha$ in $S_r$. The suffix of $S_r$ w.r.t. $\alpha$ (denoted as $suffix(S_r, \alpha)$) is the list of transactions in $S_r$ after the last transaction in $\tilde{o}$ (i.e., after $T_{e_Z}$).

**Definition 32 (Sequence-Suffix utility of sequence $\alpha$ in sequence $S_r$)** Given sequence $\alpha \preceq S_r$, the sequence-suffix utility of $\alpha$ in $S_r$ is defined as follows:

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

where *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, $SFU(\alpha, S_r)$ can be computed easily using the information in the *SeqUtilList* of N. According to Definition 23, $su(\alpha, S_r) = \max_{\tilde{o} \in OccSet(\alpha, S_r)} \{su(\alpha, \tilde{o})\}$ which can be obtained using the highest *Util* value among all the tuples with $S_r$ as its SID. The *TID* field of the first tuple stores the TID of the last transaction in $\alpha$'s first occurrences in $S_r$. With this TID value, we can easily get the TIDs of all the transactions in $suffix(S_r, \alpha)$, and obtain their $TU$ values (which were pre-computed and stored when a transaction was scanned to build *ItemUtilLists*). For example, the sequence-suffix utility of $\alpha = \langle\{a\}\{c\}\rangle$ in $S_1$ in Figure 4.1 is calculated as follow. According to $SeqUtilList(\alpha) = \{\langle S_1, T_2, 5\rangle, \langle S_1, T_4, 7\rangle\}$, $su(\alpha, S_1) = \max(5, 7) = 7$ and $suffix(S_1, \alpha) = \{T_4\}$. Hence, $SFU(\alpha, S_1) = 7 + TU(T_4) = 7 + 7 = 14$.

**Definition 33** The **SFU** of a sequence $\alpha$ in the i-th window $SW_i$, denoted as $SFU(\alpha, SW_i)$, is defined as follows: $SFU(\alpha, SW_i) = \sum_{S \in SW_i} SFU(\alpha, S)$.

**Property 1** *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, $su(\alpha, SW_i) \leq SFU(\alpha, SW_i)$.*

The following theorem states that $SFU$ is an upper bound on the utility of pattern $\beta$ and any pattern prefixed with $\beta$ where $\beta$ is produced by S-Step from $\alpha$.

**Theorem 4** *Given pattern $\alpha$ and sliding window $SW_i$ and item $I$, $SFU(\alpha, SW_i)$ is an upper bound on:*

*(A) the utility of pattern $\beta = \alpha \otimes I$. That is, $su(\beta, SW_i) \leq SFU(\alpha, SW_i)$.*

*(B) the utility of any $\beta's$ offspring $\theta$ (i.e., any sequence prefixed with $\beta$). That is, $su(\theta, SW_i) \leq SFU(\alpha, SW_i)$.*

**Proof**

Let $\beta = \alpha \otimes I$ and $S \in SW_i$. According to Definition 23, the utility of $\beta$ can be rewritten as:

$$su(\beta, S) = \max_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o}) + u(I, \tilde{o})\}$$

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

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

$$su(\beta, S) \leq \max_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o}) + \sum_{T \in suffix(S, \alpha)} TU(T)\}$$

The second part is independent of $\tilde{o}$. Thus,

$$su(\beta, S) \leq \max_{\tilde{o} \in OccSet(\beta, S)} \{su(\alpha, \tilde{o})\} + \sum_{T \in suffix(S, \alpha)} TU(T) = SFU(\alpha, S).$$

Below we prove that utility of any offspring of $\beta$ is less than $SFU(\alpha, S)$. Assume that $\theta = \alpha \otimes I \odot ... \odot ... \odot IS$ where $IS$ is the last itemset in $\theta$ and $\odot \in \{\otimes, \oplus\}$. Let $\tilde{o}_1$ be the occurrence with maximum utility of $\theta$ in $S$. The utility of $\theta$ can be rewritten as follows:

$$su(\theta, \tilde{o}_1) = su(\alpha, \tilde{o}_1) + \sum_{i \in \theta \wedge i \in suffix(S, \alpha)} u(i, \tilde{o}_1)$$

Note that all items in $\theta$ which are not in $\alpha$ occur in $suffix(S, \alpha)$. We know that $su(\alpha, \tilde{o}_1) \leq su(\alpha, S)$. Hence:

$$su(\theta, \tilde{o}_1) \leq su(\alpha, S) + \sum_{i \in \theta \wedge T \in \tilde{o}_1 \wedge T \in suffix(S,\alpha)} u(i, T)$$

Since the utility of each item in a transaction is no more than the utility of the transaction, $su(\theta, \tilde{o}_1) \leq su(\alpha, S) + \sum_{i \in T \wedge T \in suffix(S,\alpha)} TU(T) = SFU(\alpha, S)$.

The conclusion can be easily extended from $S$ to $SW_i$.

□

**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 $SFU(\alpha, SW_i) < \delta$, 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 $SFU(\alpha, SW_i)$, 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 I-step or 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 4.1 $SFU(\langle\{a\}\{b\}\{c\}\rangle\}, S_1) = 10 + 14 = 24$ 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

#### 4.3.1.5 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 34** A sequence $\alpha$ is called **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 $SFU(\alpha, SW_i) \geq \delta$

The complete tree construction process is as follows. We first generate the child nodes of the root as described in Section 4.3.1.3. Then for each child node, the *Tree-Growth* algorithm (see Algorithm 5) 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 4.3.1.3. *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 2 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 2 with each S-node.

**Algorithm 5** *TreeGrowth - Part 1*

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

**Output**: *HUSP-Tree*

1: **if** $TSWU(\alpha, SW_i) < \delta$ **then**

2: remove node $ND(\alpha)$

3: **else**

4: $I_Set \leftarrow$ items in $ItemUtilLists$ whose $TSWU >= \delta$ 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 5 ($ND(\alpha \oplus \gamma)$)

10: **end if**

11: **end for**

*TreeGrowth - Part 2*

12: **if** $SFU(\alpha, SW_i) \geq \delta$ **then**

13: $S_Set \leftarrow$ items in $ItemUtilLists$ whose $TSWU >= \delta$

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 5 ($ND(\alpha \otimes \gamma)$)

19: **end if**

20: **end for**

21: **end if**

22: **end if**

The average time complexity for building HUSP-Tree with the first sliding window is $O(NumPot \times NumOcc_{avg})$, where $NumPot$ is the number of potential high utility patterns and $NumOcc_{avg}$ is the average number of occurrences of a potential high utility pattern. Note that $NumPot$ depends on threshold $\delta$.

### 4.3.2 Update Phase

When a new transaction $S_v^u$ arrives, if the current window $SW_i$ is full, the oldest transaction $S_c^d$ 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 $SW_i$, $D^+$ represents the window after transaction $S_v^u$ is added to $SW_i$, $D^-$ represents the window after $S_c^d$ 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 4** *Given non-empty sequence $S$, after $S_v^u$ is added to the window, one of the following cases is held:*

*(1) If $S \preceq S_v$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^+) \geq su(S, SW_i)$.*

*(2) If $S \preceq S_v$ and $S \notin H^+$, then $su(S, D^+) \geq su(S, SW_i)$.*

*(3) If $S \npreceq S_v$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^+) = su(S, SW_i)$.*

*(4) If $S \npreceq S_v$ and $S \notin H^+$, then $S \notin H^-$ and $su(S, D^+) = su(S, SW_i)$.*

**Proof** Let $S'_v$ be sequence $S_v$ before transaction $S_v^u$ is appended to and $OSet_{SW_i}$ 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 25, $su(S, SW_i) \geq \delta$. Also, $S \preceq S_v$ 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 $su(S, o') > su(S, S'_v)$ 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 $S \preceq S_v$ hence $OSet_{SW_i} \subseteq OSet_{SW_{i+1}}$. In this case there is $o' \in OSet_{SW_{i+1}}$ where $o' \notin OSet_{SW_i}$. Also, $S \notin H$, according to Definition 25, $su(S, SW_i) < \delta$. If

$su(S, o') > su(S, S'_v)$ then $su(S, SW_{i+1}) > su(S, SW_i)$. Otherwise, $su(S, SW_{i+1}) = su(S, SW_i)$.

(3) Since $S \not\preceq S_v$ hence $OSet_{SW_i} = OSet_{SW_{i+1}}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{SW_{i+1}})$ . Also, $S \in H$, according to Definition 25, $su(S, SW_i) \geq \delta$. Since the utility of $S$ is the same, $S \in H^+$.

(4) Since $S \not\preceq S_v$ hence $OSet_{SW_i} = OSet_{SW_{i+1}}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{SW_{i+1}})$ . Also, $S \notin H$, according to Definition 25, $su(S, SW_i) < \delta$. Consequently, $su(S, SW_{i+1}) < \delta$ so $S \notin H^+$.

**Lemma 5** *Given sequence $S$, sequence $S'_c$ before $S^d_c$ is removed from $S_c$, one of the following cases is held:*

*(1) If $S \preceq S'_c$ and $S \in H^+$, then $su(S, D^-) \leq su(S, SW_i)$.*

*(2) If $S \preceq S'_c$ and $S \notin H^+$, then $S \notin H^-$ $su(S, D^-) \leq su(S, SW_i)$.*

*(3) If $S \not\preceq S'_c$ and $S \in H^+$, then $S \in H^-$ and $su(S, D^-) = su(S, SW_i)$.*

*(4) If $S \not\preceq S'_c$ and $S \notin H^+$, then $S \notin H^-$ and $su(S, D^-) = su(S, SW_i)$.*

**Proof** Let $OSet_{SW_i}$ be the set of occurrences of $S$ in $SW_i$ and $OSet_{SW_{i+1}}$ be the set of occurrences of $S$ in $SW_{i+1}$:

(1) Since $S \in H$, according to Definition 25, $su(S, SW_i) \geq \delta$. Also, since $S \preceq S'_c$ and $S_c \preceq S'_c$, hence $OSet_{SW_{i+1}} \subseteq OSet_{SW_i}$. In this case there is $o' \in OSet_{SW_i}$ where $o' \notin OSet_{SW_{i+1}}$. If $su(S, o') > su(S, S_c)$ then $su(S, SW_{i+1}) < su(S, SW)$. Otherwise, $su(S, SW_{i+1}) = su(S, SW_i)$.

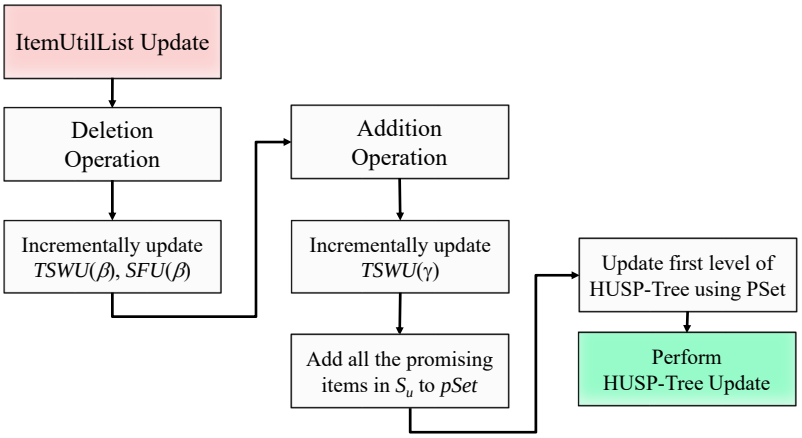



Figure 4.5: An overview of $ItemUtilLists$ update step

(2) Since $S \preceq S'_c$ and $S_c \preceq S'_c$, hence $OSet_{SW_{i+1}} \subseteq OSet_{SW_i}$. In this case there is $o' \in OSet_{SW_i}$ where $o' \notin OSet_{SW_{i+1}}$. Also, $S \notin H$, according to Definition 25, $su(S, SW_i) < \delta$. If $su(S, o') > su(S, S_c)$ then $su(S, SW_{i+1}) < su(S, SW_i)$. Otherwise, $su(S, SW_{i+1}) = su(S, SW_i)$. In both cases, $S \notin H^+$.

(3) Since $S \not\preceq S'_c$ hence $OSet_{SW_{i+1}} = OSet_{SW_i}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{SW_{i+1}})$ . Also, $S \in H$, according to Definition 25, $su(S, SW_i) \geq \delta$. Since the utility of $S$ is the same, $S \in H^+$.

(4) Since $S \not\preceq S'_c$ hence $OSet_{SW_i} = OSet_{SW_{i+1}}$. In this case $su(S, OSet_{SW_i}) = su(S, OSet_{SW_{i+1}})$ . Also, $S \notin H$, according to Definition 25, $su(S, SW_i) < \delta$. Consequently, $su(S, SW_{i+1}) < \delta$ so $S \notin H^+$.

Below we propose an efficient approach to update *ItemUtilLists* and *HUSP-Tree* based on Lemma 4 and Lemma 5.

Figure 4.5 shows the *update ItemUtilLists step*. For each item $\beta$ in the oldest transaction $S_c^d$, 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 4.5). 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 $S_v^u$, the algorithm inserts new tuple $\langle S_v, T_u, u(\gamma, S_v^u)\rangle$ 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 4.6 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 $\alpha \oplus \gamma$ or $\alpha \otimes \gamma$ 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-

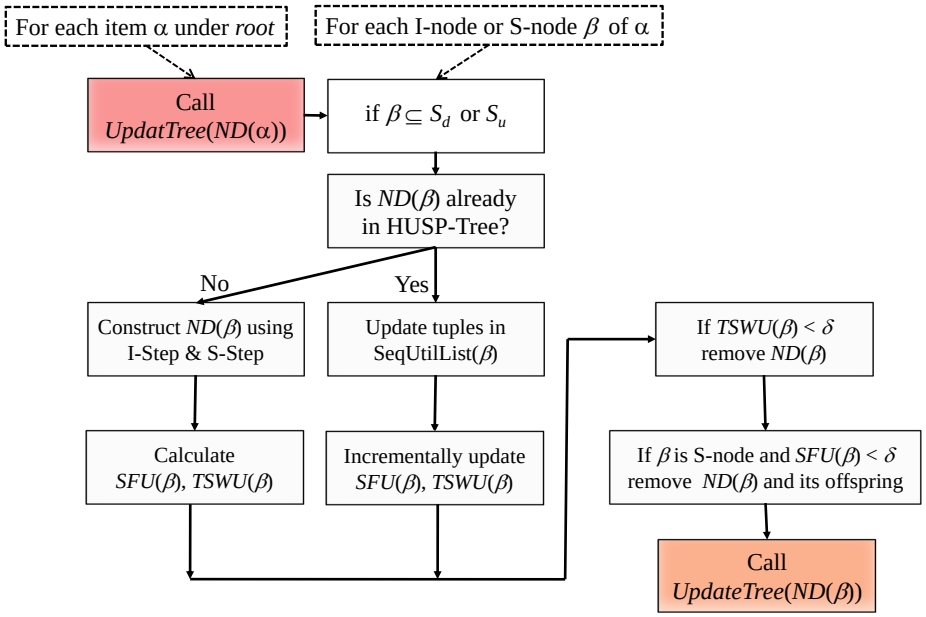



Figure 4.6: An overview of HUSP-Tree update step

Tree, the algorithm incrementally updates the tuples in *SeqUtilList(*$\beta$*)* related to the new and oldest transactions as follows. Given the oldest transaction $S_c^d$ and the newest transaction $S_v^u$, according to Lemma 4 and Lemma 5, 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 reconstructed by applying I-Step (if $\beta$ is $\alpha \oplus \gamma$) or S-Step ( if $\beta$ is $\alpha \otimes \gamma$) on $SeqUtilList(\alpha)$ and $ItemUtilLists(\gamma)$. Then the algorithm updates TSWU of $\beta$ based on the updated $SeqUtilList(\beta)$. If TSWU of $\beta$ is less than the utility threshold, the algorithm removes $ND(\beta)$ and the sub-tree under $ND(\beta)$. Otherwise, if $\beta$ is $\alpha \oplus \gamma$, the algorithm calls the procedure *UpdateTree(*$ND(\beta)$*)* to update the sub-tree of $ND(\beta)$. If $\beta$ is $\alpha \otimes \gamma$, the SFU of $\beta$ is updated using the updated $SeqUtilList(\beta)$. If

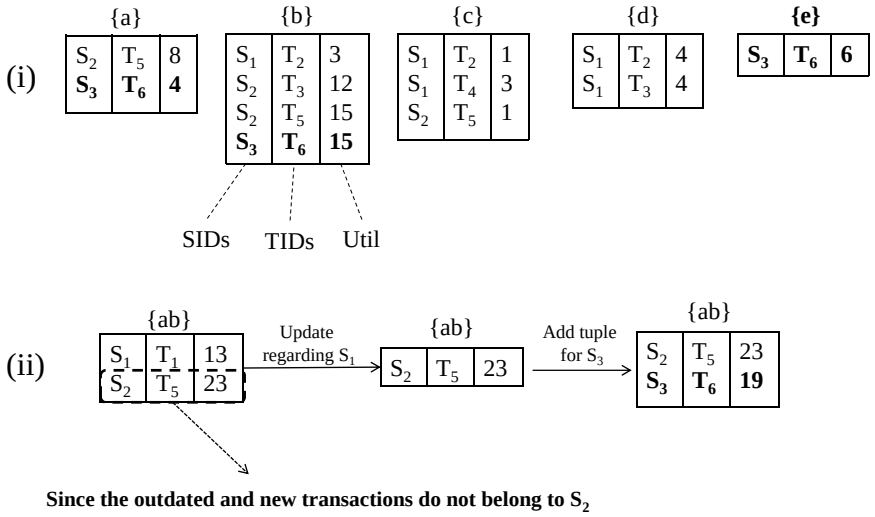



Figure 4.7: The updated (i) $ItemUtilLists$ and (ii) $SeqUtilList(\{ab\})$ after removing $T_1$ from and adding $T_6$ to the window

SFU of $\beta$ is less than the threshold, node $ND(\beta)$ and its subtree are removed from the tree; otherwise, it recursively calls ***UpdateTree(***$ND(\beta)$***)***.

For example, Figure 4.7 shows the updated $ItemUtilLists$ and $SeqUtilList(\{ab\})$ when $T_1$ is removed from and $T_6$ is added to the window. Note that we do not reconstruct the whole $SeqUtilList(\{ab\})$. 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 $\{ab\}$ is not in $S_1$ any more but exists in $S_3$, $SeqUtilList(\{ab\})$ is updated as $SeqUtilList(\{ab\}) = \{\langle S_2, T_5, 23\rangle, \langle S_3, T_6, 19\rangle\}$.

Since a tuple in $ItemUtilLists$ can be accessed directly and the number of tuples needed to be updated in $ItemUtilLists$ is $L_{oldest}+L_{new}$, where $L_{oldest}$ is the length of the

transaction to be removed and $L_{new}$ is the length of the new transaction added to the sliding window, the average time complexity for updating $ItemUtilLists$ is $O(L_{avg})$, where $L_{avg}$ is the average length of transactions in the data stream. The average time complexity for updating HUSP-Tree is $O(NumPot \times NumOccAff_{avg})$ where $NumPot$ is the number of potential high utility patterns in the new sliding window, and $NumOccAff_{avg}$ is the average number of occurrences of a potential high utility pattern in the sequences affected by the removal of the oldest transaction and the addition of the new transaction.

### 4.3.3 HUSP Mining Phase

HUSP mining phase is straight forward. After performing the update phase, *HUSP-Tree* maintains the information of the potential HUSPs in the current window. 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 *SeqUtilList* of $ND(\alpha)$ to calculate the utility of $\alpha$ in the current window. 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 outputted. Note that this HUSP mining phase can be combined with the update phase. During 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 outputted as a HUSP during the update phase.

Table 4.2: Parameters of IBM data generator

| | |
|---|---|
| D | Number of sequences |
| C | Average number of transactions in a sequence |
| T | Average number of items in a transaction |
| S | Average number of itemsets in a potential maximal sequential pattern |
| I | Average number of items in an itemset of a potential maximal sequential pattern |
| N | Number of distinct items |

## 4.4 Experiments

In this section, we evaluate the performance of the proposed method. The experiments were conducted on an Intel(R) Core(TM) i7 2.80 GHz computer with 8 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 [2]. The definition of parameters used by the IBM data generator are shown in the Table 4.2.

*Chainstore* is a real-life dataset acquired from [51], which already contains internal and external utilities. In order to use this dataset as a sequential dataset, we grouped transactions in different sizes such that each group represents a sequence of transactions. Another real dataset *BMS* is obtained from SPMF [25] which contains 3340 distinct items and consists of 77,512 sequences of clickstream data from an e-retailer. We follow the

Table 4.3: Details of parameter setting

| **Dataset** | **#Seq** | **#Trans** | **# Items** | **Window Size (w)** |
|---|---|---|---|---|
| DS1 | 10K | 100K | 1000 | 50K |
| BMS | 77K | 120K | 3340 | 60K |
| DS2 | 100K | 800K | 1000 | 400K |
| ChainStore | 400K | 1000K | 46,086 | 500K |

previous study [4] 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 4.3 shows characteristics of the datasets and parameter settings in the experiments. The *Window Size* column of Table 4.3 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:

- *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.

### 4.4.1 Methods in Comparison

To the best of our knowledge, no study has been proposed for mining high utility sequential patterns over data streams. Hence, we compare our method with USpan [66], which is the current best algorithm for mining high utility sequential patterns in static databases. Since USpan is not applicable to data streams, we design two approaches to apply USpan over data stream: (1) We run 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. (2) The datasets used in the experiments are quite large and the window slides a large number of times, so the first approach runs very slow. 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

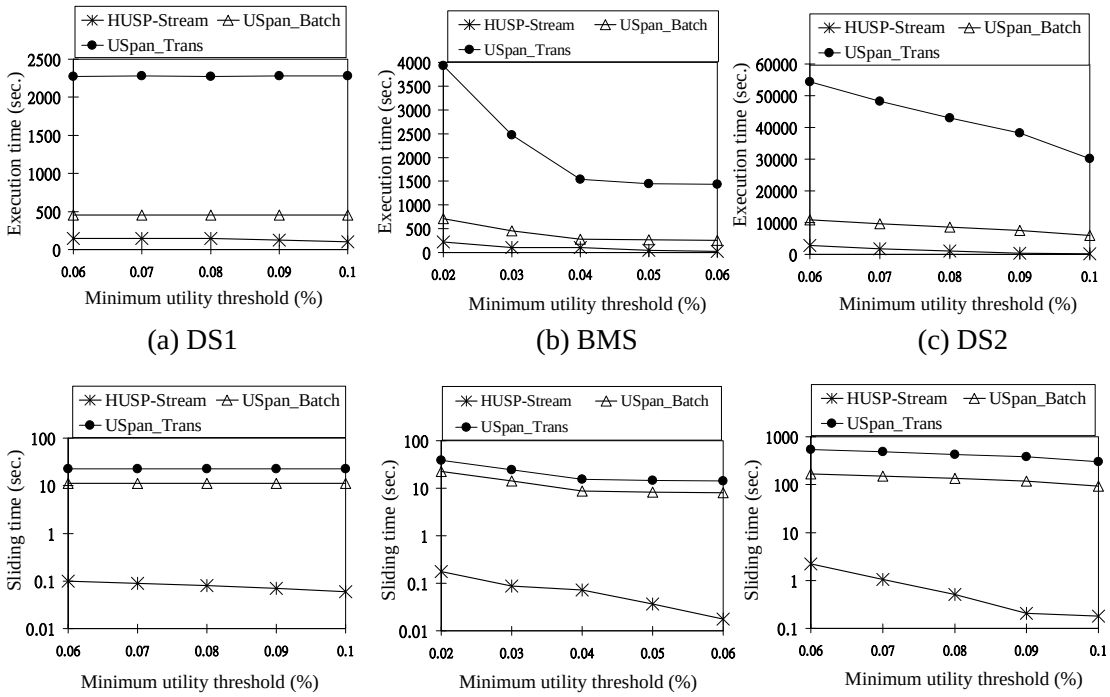



Figure 4.8: Execution time and sliding time (shown in logarithmic scale) over consecutive windows

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.

### 4.4.2 Performance evaluation for sliding two hundred consecutive windows

In this section we investigate the efficiency of update processing of the methods. Since *USpan* is not designed for data stream environment, it is not able to return results over

all the sliding windows in the datasets 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 4.8(a), Figure 4.8(b) and Figure 4.8(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 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 4.8(d), Figure 4.8(e) and Figure 4.8(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 the 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 comparison method since the datasets are really large and it can not return results due to memory problem and long time processing.

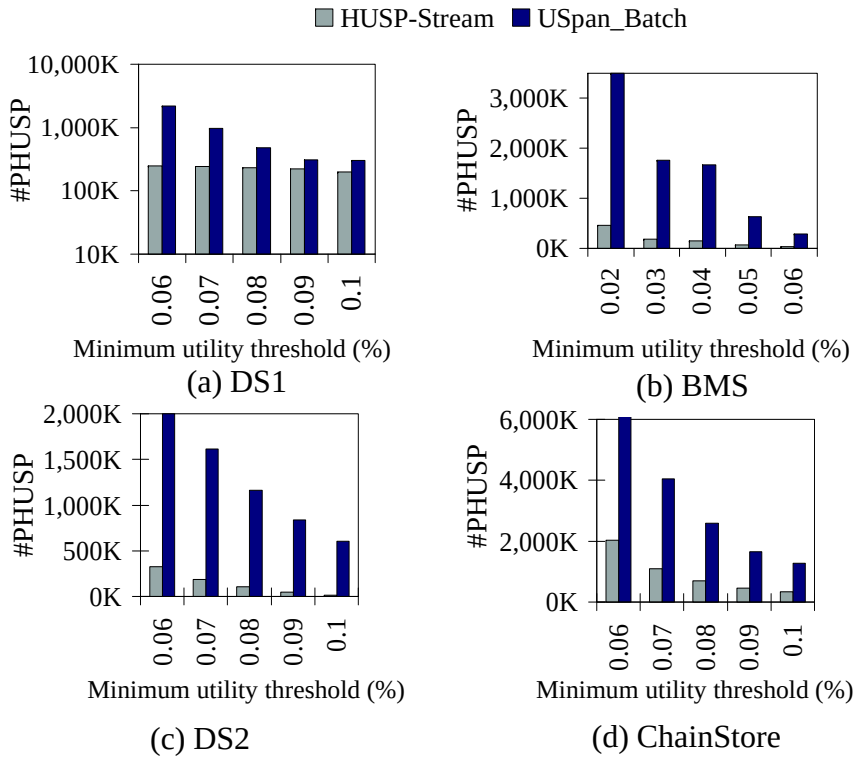



Figure 4.9: Number of PHUSPs on different datasets

### 4.4.3 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 4.9 shows the results under different utility thresholds. For consistency across datasets, the minimum threshold is shown as a percentage of the total utility of all the sequences in a dataset. As shown in Figure 4.9, 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.

### 4.4.4 Time Efficiency of HUSP-Stream

Figure 4.10(a), Figure 4.10(b), Figure 4.10(c) and Figure 4.10(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-run 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.

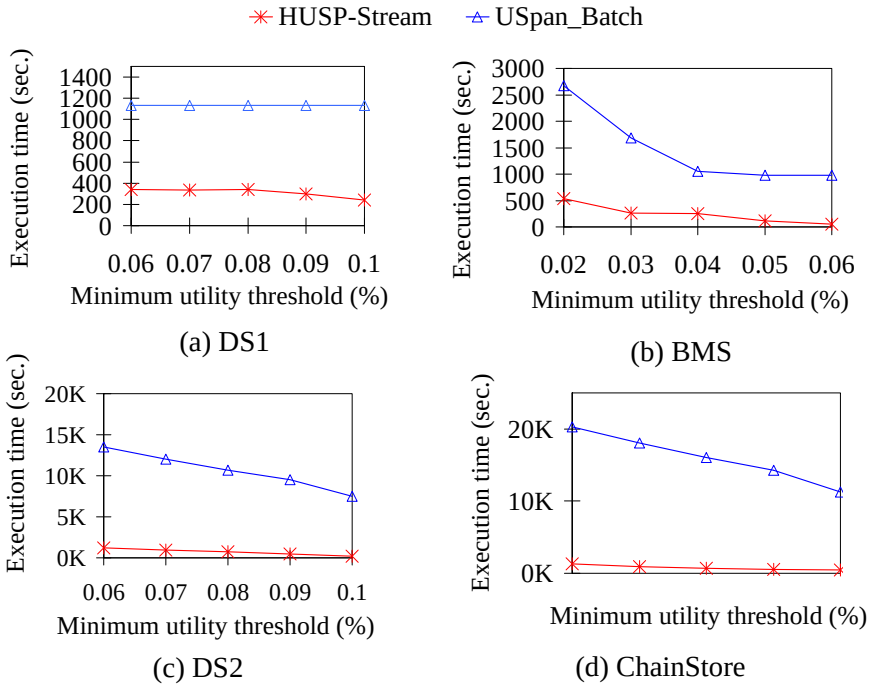



Figure 4.10: Execution time and sliding time (shown in logarithmic scale) on different datasets

Even under the low threshold, it can perform well. From this experiment, the less number of PHUSPs of HUSP-Stream is another reason that the run time of HUSP-Stream is less than that of USpan_Batch. Also maintenance performance of data structures used in the algorithm is the other reason that HUSP-Stream always outperforms USpan_Batch.

Then we evaluate the average window sliding time of the algorithms under different minimum utility thresholds. Figure 4.11 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

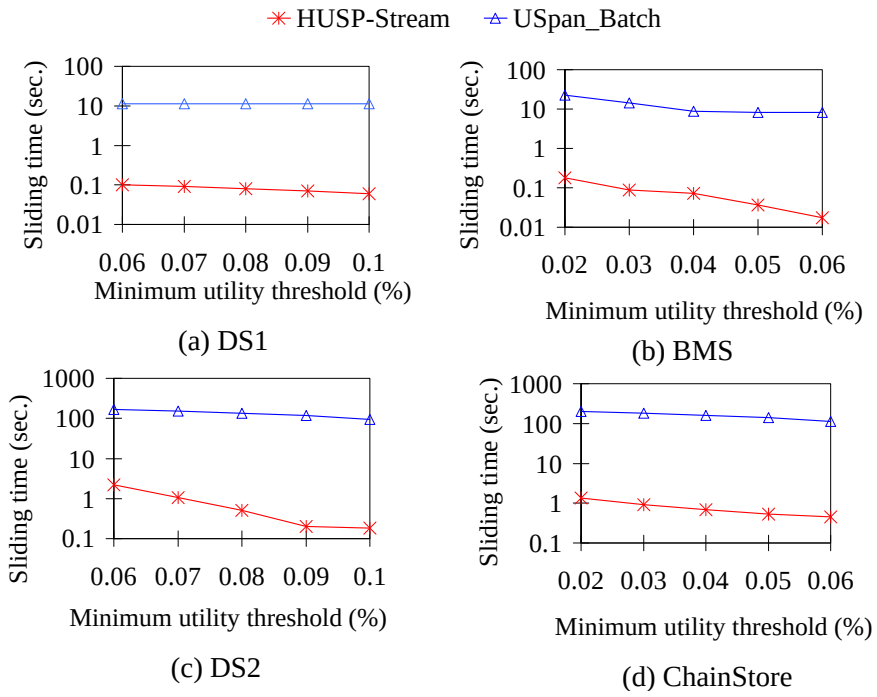



Figure 4.11: Sliding time on different datasets

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 that 260 seconds. In this case, HUSP-Stream is 200 times faster than the USpan_Batch.

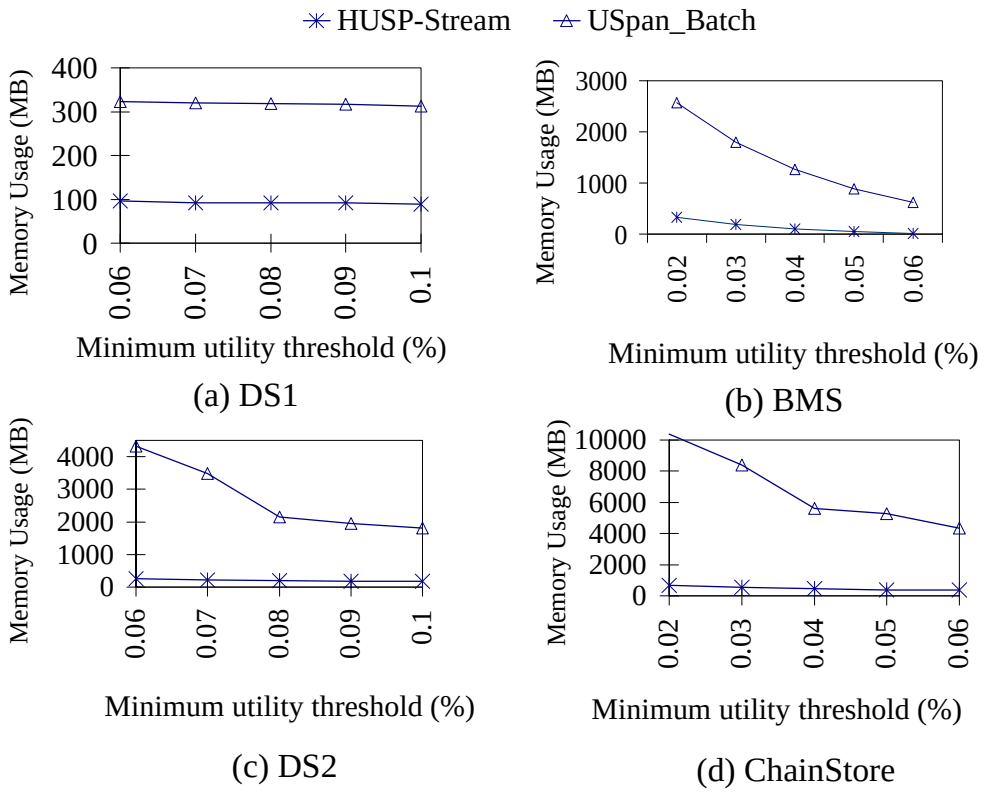



Figure 4.12: Memory Usage of the algorithms

### 4.4.5 Memory Usage

We also evaluate the memory usage of the algorithms under different utility thresholds. The results are shown in Figure 4.12, 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 to have more tree nodes than HUSP-Stream.

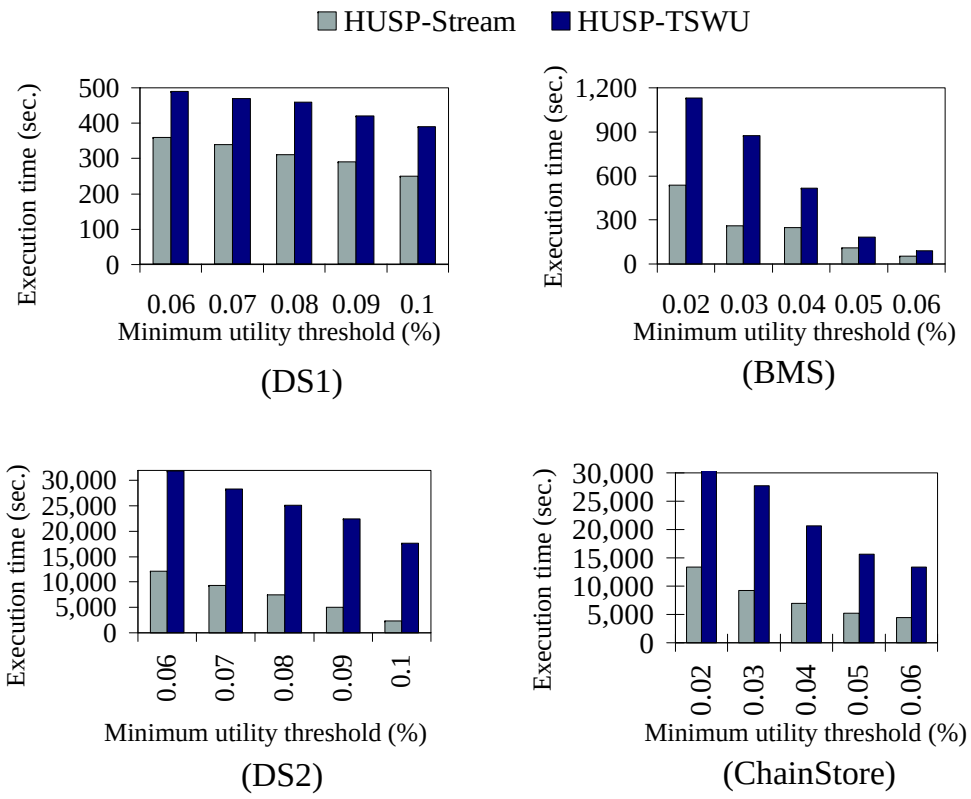



Figure 4.13: Impact of SFU on Run Time

### 4.4.6 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, HUSP-Stream is compared to its basic version, *HUSP_TSWU*, which only applies the $TSWU$ pruning strategy for pruning I-nodes and S-nodes.

Figure 4.13, Figure 4.14 and Figure 4.15 illustrate the run time, the number of potential HUSPs generated by the two methods, and their memory usage under different

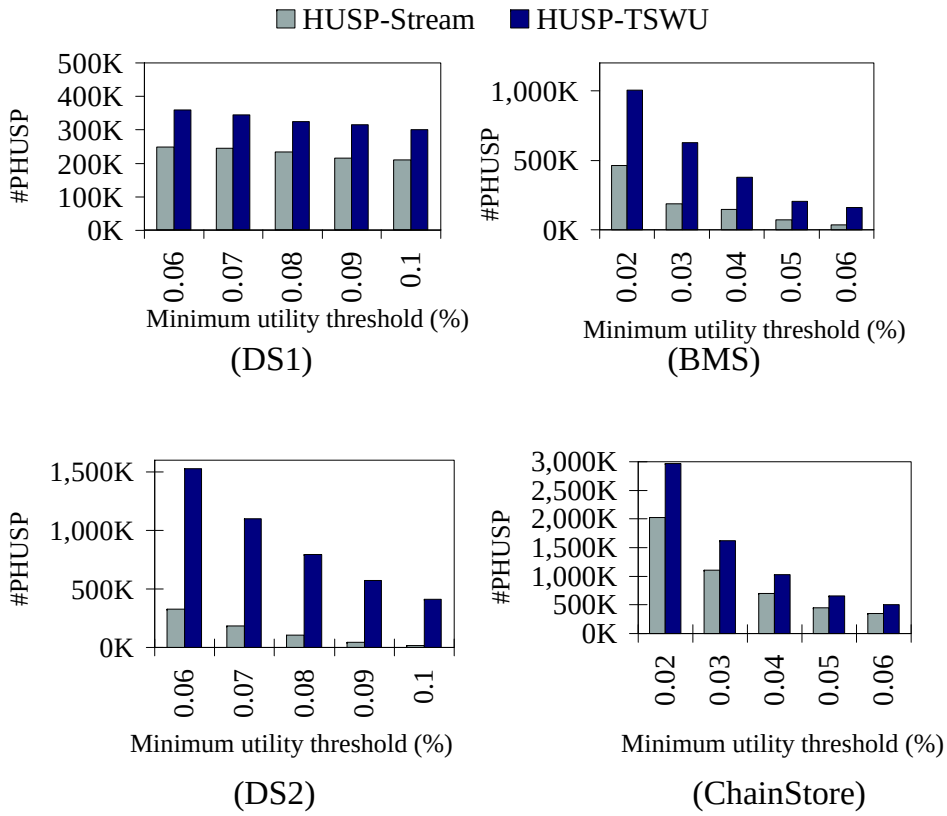



Figure 4.14: Impact of SFU on Number of PHUSPs

utility threshold values. These figures show that our new pruning strategy is more effective than using only *TSWU* in all three performance measures. Moreover, these figures show that the differences between the two pruning methods in the number of PHUSPs, run time and memory usage increase in general when the utility threshold decreases. These results indicate that our proposed $SFU$ is much more effective than $TSWU$ in pruning.

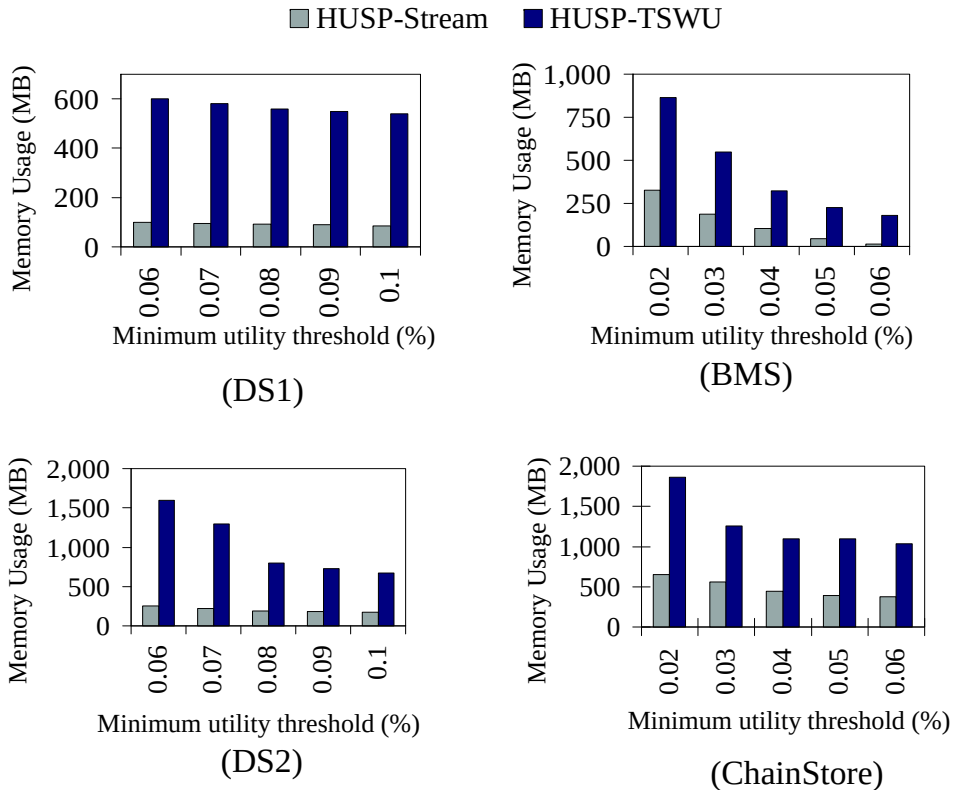



Figure 4.15: Impact of SFU on Memory Usage.

### 4.4.7 Performance Evaluation with Window Size Variation

Below 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 4.16. In Figure 4.16(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.

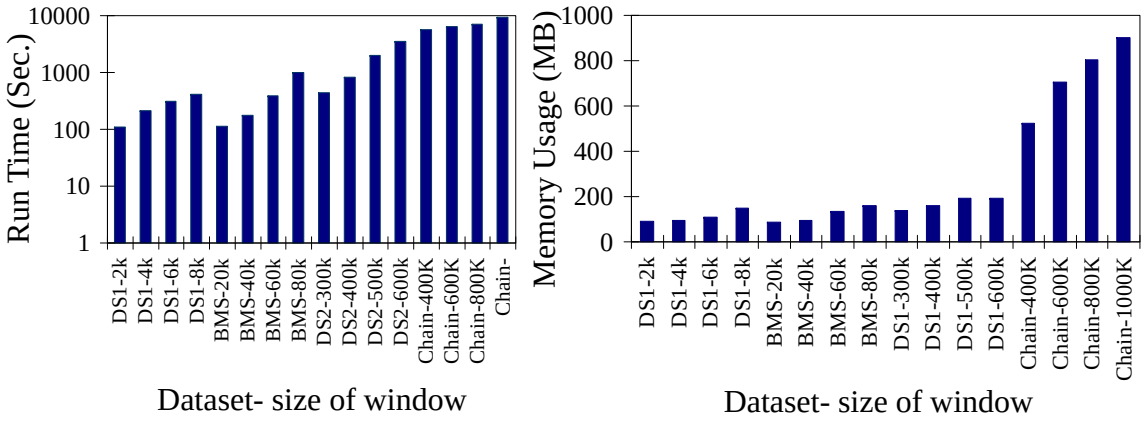



Figure 4.16: Evaluation of HUSP-Stream under different window sizes

From Figure 4.16(a), we can observe that the memory consumption of HUSP-Stream increases very slowly with increasing window sizes. Figure 4.16(b) shows the execution time of HUSP-Stream under different window sizes. We can see that HUSP-Stream is also scalable in time with increasing window sizes.

### 4.4.8 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 4.17 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

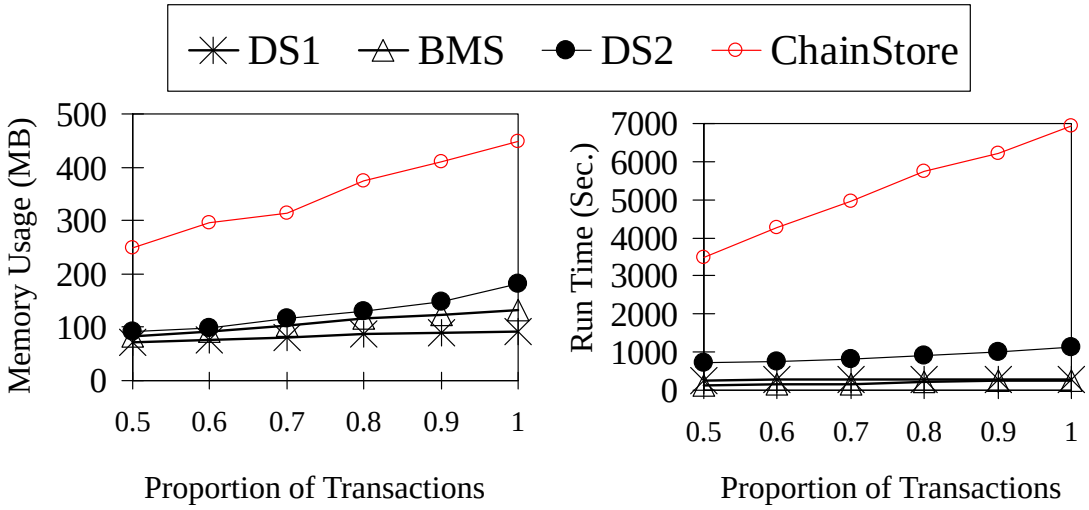



Figure 4.17: Scalability of HUSP-Stream on different datasets: (a) Memory Usage , (b) Run Time

dataset.

## 4.5 Summary

In this chapter, we addressed main deficiencies and challenges of mining recent HUSPs over data streams, by proposing a new framework for *sliding window-based high utility sequential pattern mining over data streams*. To the best of our knowledge, existing methods have the following deficiencies. (1) They are frequency-based, and did not consider the utility (e.g., value) of an item and thus cannot be used to find HUSPs over sliding windows. (2) Most of the studies 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. 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. Our framework incrementally learns HUSPs from a sliding window over data streams of itemset-sequences. The major contributions of this work are summarized as follows.

1. **Efficient data structure to maintain essential information:** we proposed 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 in a transaction-sensitive sliding window over a data stream. To the best of our knowledge, 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 updates when the window slides, and can be used to compute sequence utilities without re-scanning the transactions in the sliding window.

2. **Efficient search space pruning strategy:** we proposed 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 [4] for HUSP mining.

3. **Single pass mining algorithm:** we proposed a new one-pass algorithm called *HUSP-Stream (High Utility Sequential Pattern Mining over Data Streams)* for efficiently constructing and updating *ItemUtilLists* and *HUSP-Tree* by reading a transaction in the data stream *only once*, and by making use of both $SFU$ and $SWU$ to prune the size of *HUSP-Tree*. 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 updated databases. 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.

4. **Extensive experiments:** we conducted extensive experiments on both real and synthetic datasets to evaluate the performance of the proposed algorithm. Ex-

perimental results show that HUSP-Stream outperforms the state-of-the-art HUSP mining algorithm 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 [66], a state-of-the-art HUSP mining algorithm, fails to complete the mining task.

# 5 Top-k High Utility Pattern Mining over Data Streams

Since there could be a large number of high utility patterns, finding only top-$k$ patterns is more attractive than producing all the patterns whose utility is above a threshold. A challenge with finding high utility patterns over data streams is that it is not easy for users to determine a proper minimum utility threshold in order for the method to work efficiently. In this chapter, we propose two methods for finding top-k high utility patterns over sliding windows of a data stream. The first proposed method, called *T-HUDS*, finds top-k *high utility itemsets* over data streams. *T-HUDS* is based on a compressed tree structure, called *HUDS-Tree*, that can be used to efficiently find potential top-$k$ high utility itemsets over sliding windows. *T-HUDS* uses a new utility estimation model to more effectively prune the search space. We also propose several strategies for initializing and dynamically adjusting the minimum utility threshold. Then, inspired by T-HUDS, we propose our second method, called *T-HUSP*, for discovering of top-k *high utility sequential patterns* over data streams. This method is based on our proposed method in Chapter 4 (i.e., *HUSP-Stream*). T-HUSP incrementally maintains the content of top-k HUSPs in

the sliding window in a summary data structure, named *TKList*. In addition, two efficient strategies are proposed for initializing and raising the threshold. Our experimental results on real and synthetic datasets demonstrate impressive performance of the proposed methods without missing any high utility patterns.

## 5.1 Top-k High Utility Itemset Mining over a Data Stream

A key problem with the existing HUI mining methods over data streams is that the user needs to supply a minimum utility threshold. A solution to this threshold setting problem is to mine top-$k$ high utility itemsets, in which the user supplies $k$, the number of HUIs to be returned. A benefit of mining top-$k$ HUIs is that it is easier and more intuitive for the user to indicate how many patterns they would like to see than specifying a utility threshold. In addition, the number of returned patterns will be under control and the results will not overwhelm the user. A major challenge in top-$k$ HUI mining is that the number of itemsets is exponential and it is infeasible to compute the utilities of all the itemsets and identify the top-$k$ ones. A minimum utility threshold is thus needed in the mining process to prune the search space.

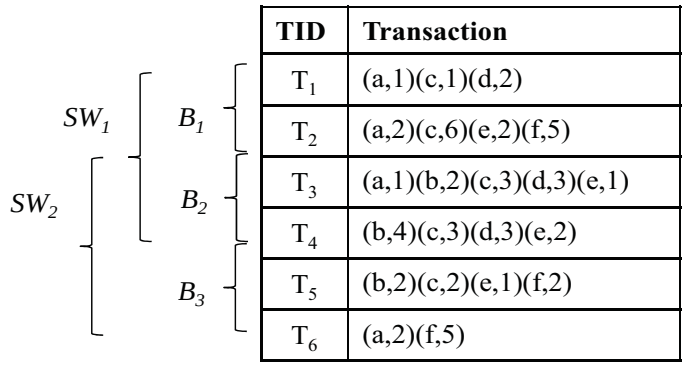
In this section, we propose effective strategies for automatically initializing and dynamically adjusting the minimum utility threshold for mining top-$k$ high utility itemsets over data streams. Three of our strategies can be applied to both static and streaming data, and one of them is specially designed for data streams. We use a sliding window

based data stream mining method, in which a set of recent data (called a *sliding window*) is the target of mining. In addition to the new strategies for setting and adjusting the threshold, we also propose to use another over-estimate utility as the search heuristic for finding HUIs in the first phase of the top-$k$ HUI mining process. This over-estimate (called *prefix utility*) is more effective than the most commonly used $TWU$ (an over-estimate of the itemset utility) in pruning the search space because it is a closer estimate of the true utility than $TWU$. The contributions are as follows:

- We propose a method for mining top-$k$ high utility itemsets from data streams. To the best of our knowledge, existing methods for mining HUIs over data streams do not address the issue of mining top-$k$ HUIs, and previous top-$k$ HUI mining methods do not work on data streams.

- We propose several strategies for initializing and dynamically adjusting the minimum utility threshold during the top-$k$ HUI mining process. We prove that using these strategies will not miss any top-$k$ HUIs.

- We propose an over-estimate of the itemset utility, which is closer to the true utility than TWU. We prove that this estimate (i.e., *prefix utility*) has a special type of downward closure property, which allows it to be used in the pattern growth method to effectively prune the search space. Using a closer over-estimate results in fewer candidates being generated in the first phase of the method.




| TID | Transaction |
|---|---|
| $T_1$ | (a,1)(c,1)(d,2) |
| $T_2$ | (a,2)(c,6)(e,2)(f,5) |
| $T_3$ | (a,1)(b,2)(c,3)(d,3)(e,1) |
| $T_4$ | (b,4)(c,3)(d,3)(e,2) |
| $T_5$ | (b,2)(c,2)(e,1)(f,2) |
| $T_6$ | (a,2)(f,5) |

| Item Name | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| External utility | 3 | 6 | 5 | 8 | 4 | 3 |

Figure 5.1: Example of transaction data base and external utility of items

- We propose a compact data structure (called *HUDS-Tree*) to store the information about the transactions in a sliding window. The tree is used to compute the prefix utility and to initialize and adjust the minimum utility threshold.

- We conduct an extensive experimental evaluation of the proposed method on both real and synthetic datasets, which shows that our proposed method is faster and less memory consuming than the state-of-the-art methods.

### 5.1.1 Preliminaries and Problem Statement

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items and each item $i_j \in I$ is associated with a positive number $p(i_j)$, called its *external utility* (which can be the price or profit) of item $i_j$. Let $D$ be a set of $N$ transactions: $D = \{T_1, T_2, ..., T_N\}$ such that for $\forall T_j \in D, T_j =$

$\{(i, q(i, T_j))|i \in I, q(i, T_j)$ is the quantity of item $i$ in transaction $T_j\}$. Figure 5.1 shows an example of a dataset with six transactions.

**Definition 35** ***Utility of an itemset*** $X$ ***in a dataset of transactions*** $D$ *of transactions is defined as:* $u_D(X) = \sum_{X \subseteq T_j \wedge T_j \in D} \sum_{i \in X} u(i, T_j)$.

We use $u(X)$ to denote $u_D(X)$ when dataset $D$ is clear in the context.

**Definition 36** ***(High Utility Itemset (HUI))*** An itemset $X$ is called a high utility itemset (HUI) *on a dataset* $D$ *if and only if* $u_D(X) \geq min\_util$ *where* min_util *is called a* minimum utility threshold.

We are interested in mining top-$k$ HUIs in data streams. In a data stream environment, transactions come continually over time, and they are usually processed in batches. A **transaction batch** $B_i$ consists of transactions arriving continuously in a time period, i.e., $B_i = \{T_j, T_{j+1}, ..., T_m\}$. For example, assuming that the dataset in Figure 5.1 is a data stream and that each batch contains 2 transactions, there are three batches in the stream: $B_1 = \{T_1, T_2\}, B_2 = \{T_3, T_4\}$, and $B_3 = \{T_5, T_6\}$.

A **sliding window** is a set of $w$ most recent batches, where $w$ is called the size of window, denoted as *winSize*. If the first batch in a sliding window is $B_i$, the window can be represented as $SW_i = \{B_i, B_{i+1}, ..., B_{i+winSize-1}\}$. As a new batch forms up in a data stream, the sliding window removes its oldest batch and adds the new batch to the window. For example, consider the data stream in Figure 5.1. Assume that the $winSize$

is 2. The first two batches form the first sliding window: $SW_1 = \{B_1, B_2\}$. When the third batch $B_3$ is filled up with transactions, the second sliding window is formed: $SW_2 = \{B_2, B_3\}$. The problem tackled is defined as follows.

**Problem 1** For each sliding window $SW_i$ in a data stream, the problem is to find the top-$k$ high utility itemsets in $SW_i$, ranked in descending order of their utility, where $k$ is a positive integer given by the user.

### 5.1.2 Challenges and New Definitions

There are inherent challenges in mining top-$k$ *HUIs* in data streams. First, since streaming data can come continuously in a high speed, they need to be processed as fast as possible. As mentioned earlier, the utility of an itemset does not have the downward closure property, and thus most of the existing HUI mining methods use $TWU$ (an over-estimate of the itemset utility) as the search heuristic to prune itemsets whose $TWU$ is below the minimum utility threshold. To further speed up the HUI mining process, we define another over-estimate utility of an itemset, which provides a closer estimation of the true utility of an itemset than $TWU$. This over-estimate utility, called *Prefix Utility*, is used in our HUI mining to more effectively prune the search space.

**Definition 37 Prefix utility of an itemset $X$ in a transaction $T$.** Assume the items in $T$ are ranked in an order (such as the lexicographic order) and that $X \subseteq T$. The *prefix*

*set* of $X$ in $T$, denoted as $PrefixSet(X,T)$, consists of all the items in $T$ that are not ranked after any item in $X$. The *prefix utility* of $X$ in $T$ is defined as:

$$PrefixUtil(X,T) = \sum_{i \in PrefixSet(X,T)} u(i,T)$$

For example, in Figure 5.1, the prefix set of itemset $\{ac\}$ in transaction $T_3$ is $\{abc\}$. Thus, $PrefixUtil(\{ac\},T_3) = u(a,T_3) + u(b,T_3) + u(c,T_3) = 3 + 12 + 15 = 30$.

**Definition 38 Prefix utility of an itemset $X$ in a dataset $D$** is defined as:

$$PrefixUtil_D(X) = \sum_{X \subseteq T_j \bigwedge T_j \in D} PrefixUtil(X,T_j)$$

Here we assume that items in all the transactions are ranked in the same order.

**Example 1** *Let $D$ be the dataset in Figure 5.1. Since only $T_1$, $T_2$ and $T_3$ in $D$ contain itemset $\{ac\}$, we have*

$$\begin{aligned} PrefixUtil_D(\{ac\}) &= PrefixUtil(\{ac\},T_1) + PrefixUtil(\{ac\},T_2) + \\ & \quad PrefixUtil(\{ac\},T_3) = 8 + 36 + 30 = 74 \end{aligned}$$

**Property 2** *For any itemset $X$ in a dataset $D$, the following relationship holds:*

$$TWU_D(X) \geq PrefixUtil_D(X) \geq u_D(X)$$

**Lemma 6** *Assume that items in all the transactions in a dataset $D$ are ranked in an order. Let $X$ be an itemset and $X = Y \cup \{i\}$ where $i$ is the last item in X in the ranked order. For all $Z \subseteq Y$,*

$$PrefixUtil_D(Z \cup \{i\}) \geq PrefixUtil_D(X).$$

**Proof**

Let $S_X$ be the set of transactions containing $X$ in a dataset $D$. According to Definition 38, we have:

$$PrefixUtil_D(Z \cup \{i\}) = PrefixUtil_{S_X}(Z \cup \{i\}) + PrefixUtil_{D-S_X}(Z \cup \{i\}).$$

Since itemset $Z \cup \{i\}$ contains the last item in $X$ and $Z \cup \{i\} \subseteq X$, we have:

$$PrefixUtil_{S_X}(Z \cup \{i\}) = PrefixUtil_{S_X}(X).$$

Clearly, $PrefixUtil_{S_X}(X) = PrefixUtil_D(X)$. Thus,

$$PrefixUtil_D(Z \cup \{i\}) = PrefixUtil_D(X) + PrefixUtil_{D-S_X}(Z \cup \{i\}).$$

Since $PrefixUtil_{D-S_X}(Z \cup \{i\}) \geq 0$,

$$PrefixUtil_D(Z \cup \{i\}) \geq PrefixUtil_D(X).$$

□

This lemma means that the prefix utility of an itemset $X$ has the *downward closure property* if we only concern the subsets of $X$ that contain the last item in $X$ in the ranked order. Such a special kind of the downward closure property allows us to use $PrefixUtil$ to prune search space in our HUI mining algorithm to be described later.

**Example 2** *Assume that $a$, $b$ and $c$ are items in a dataset and that the items in the dataset are ranked in the lexicographic order. According to Lemma 6, $PrefixUtil(\{ac\}) \geq PrefixUtil(\{abc\})$ and $PrefixUtil(\{bc\}) \geq PrefixUtil(\{abc\})$. Thus, if $PrefixUtil(\{ac\})$ or $PrefixUtil(\{bc\})$ is less than a minimum utility threshold, $PrefixUtil(\{abc\})$ must be less than the threshold. Since $PrefixUtil(\{abc\}) \geq u(\{abc\})$, $u(\{abc\})$ must be less than the threshold.*

The second challenge of our problem is in finding top-$k$ patterns. An efficient method for finding top-$k$ patterns is to first find potential patterns whose (estimated) utility is above a threshold and then identify the top-$k$ patterns from the potential ones [61]. Since the minimum utility threshold is not given in the top-$k$ problem, a challenge in top-$k$ pattern mining is how to set up the threshold so that the process generates fewer number of potential patterns that include all the top-$k$ patterns. To meet this challenge, we propose some strategies for initializing and dynamically raising the minimum utility threshold during the stream mining process. Below we define *minimum transaction utility*, which will be used in our strategy for initializing the threshold.

**Definition 39 Minimum Transaction Utility (mtu)** of a transaction $T$ is defined as: $mtu(T) = \min_{i \in T}(u(i, T))$.

For example, in Figure 5.1, $mtu(T_4) = \min(u(b, T_4), u(c, T_4), u(d, T_4), u(e, T_4)) = \min(24, 15, 24, 8) = 8$.

Based on the $mtu$ values of the transactions, we define an underestimate utility of an itemset in a dataset as follows.

**Definition 40 Minimum Transaction Utility (MTU)** of an itemset X over a dataset $D$ is defined as: $MTU_D(X) = \sum_{X \subseteq T \bigwedge T \in D} mtu(T)$.

We use $MTU(X)$ to denote $MTU_D(X)$ when the dataset $D$ is clear in the context. For example, for the dataset in Figure 5.1:

$MTU(\{bc\}) = mtu(T_3) + mtu(T_4) + mtu(T_5) = 3 + 8 + 4 = 15$.

**Lemma 7** *For any itemset $X$ in a dataset $D$, the following relationship holds: $MTU_D(X) \leq u_D(X)$.*

**Proof**

Given itemset $X$, let $S_X$ be the set of transactions in $D$ that contain $X$. For a transaction $T \in S_X$, according to Definition 39, we have:

$$mtu(T) = \min_{i \in T}(u(i, T)) \text{ and } u(X, T) = \sum_{i \in X} u(i, T)$$

Hence, $mtu(T) \leq u(X,T)$. According to Definitions 35 and 40,

$$MTU_{S_X}(X) = \sum_{X \subseteq T \bigwedge T \in S_X} mtu(T) \leq \sum_{X \subseteq T \bigwedge T \in S_X} u(X,T) = u_{S_X}(X)$$

Since transactions not in $S_X$ do not contain $X$, we have $MTU_D(X) \leq u_D(X)$.

□

**Lemma 8** *The minimum transaction utility of an itemset satisfies the downward closure property. That is, for all* $Y \subseteq X$, $MTU(Y) \geq MTU(X)$.

**Proof**

Since all the transactions containing an itemset $X$ also contains any subset $Y$ of $X$, $MTU(Y) \geq MTU(X)$.

□

The third challenge for mining top-k HUIs in streaming data is that there can be a huge amount of data in a data stream. Thus, use of compact memory data structures is necessary in the mining process. To meet this challenge, a compressed data structure, called *HUDS-Tree*, is used in our method which can be built with one scan of data. Finding potential patterns is done based on the information in *HUDS-Tree*. *HUDS-Tree* and our method for finding top-$k$ HUIs are described in the next section. For convenience, Table 5.1 summarizes the concepts and notations we define in this section.

Table 5.1: Summary of Notations

| **Concept** | **Description** |
|---|---|
| $u(i, T)$ | Utility of item $i$ in transaction $T$ |
| $u(X)$ | Utility of itemset $X$ in a dataset |
| $TWU(X)$ | Transaction-Weighted Utility (an over-estimate utility) |
| $HUI$ | High Utility Itemsets |
| $PrefixUtil(X)$ | Prefix Utility of itemset $X$ |
| $mtu(T)$ | Minimum Transaction Utility of transaction $T$ |
| $MTU(X)$ | Minimum Transaction Utility of Itemset $X$ (an underestimated utility) |
| $LPI(X)$ | Lowest Profit Item Utility of Itemset $X$ (an underestimated utility) |
| $miu(i)$ | Minimum Item Utility of item $i$ in any transaction of a dataset |
| $MIU(X)$ | Minimum Itemset Utility of Itemset $X$ (an underestimated utility) |
| $maxUtilList$ | List of maximum values of MTUs and LPIs for each level of *HUDS-Tree* |
| $MIUList$ | List of top-k MIU values in potential HUIs |
| $minTopKUtil_i$ | Minimum Top-$k$ Utility of the $i$th sliding window |
| $PTKHUI$ | Potential Top-$k$ High Utility Itemset |
| $PTKSet$ | Set of Potential Top-$k$ High Utility Itemsets |

### 5.1.3 T-HUDS: Top-k High Utility Itemset Mining over a Data Stream

In this section, we propose an efficient method (called *T-HUDS*) to find top-k *HUIs* in data streams without specifying a minimum utility threshold. *T-HUDS* works based on a prefix tree, called *HUDS-Tree* (High Utility Data Stream Tree), and two auxiliary lists of utility values. *HUDS-Tree* dynamically maintains a compressed version of the transactions in a sliding window. The two auxiliary lists each maintain a utility list of length $log_2(k+1)$ or $k$, where $k$ is the number of top-$k$ itemsets to be returned, and are used to dynamically adjust the minimum utility threshold during the mining process.

#### 5.1.3.1 An Overview of T-HUDS

The *T-HUDS* method includes three main steps: (1) *HUDS-Tree* construction: construct a *HUDS-Tree* and two auxiliary lists; (2) *HUDS-Tree* mining: discover top-k *HUIs* from the current sliding window; and (3) *HUDS-Tree* update: once a new batch arrives, inserts the transactions in the new batch into the tree, removes transactions in the oldest batch from the tree if the sliding window had been filled up, and updates the two auxiliary lists.

Algorithm 6 presents an overview of the proposed method. We assume that the data stream comes in batches. Given a batch $B_i$ of transactions, $k$ and the sliding window size ($winSize$), if a *HUDS-Tree* does not exist yet (i.e., the batch is the very first one), a *HUDS-Tree* is constructed based on the transactions in $B_i$, and two auxiliary lists,

$maxUtilList$ and $MIUList$, are also computed or initialized. If a *HUDS-Tree* already exists, the tree and the two auxiliary lists are updated to reflect the additions or changes of transactions in the sliding window. Once a new window is formed, *T-HUDS* calls Algorithm 8 to find top-$k$ HUIs for the new sliding window.

**Algorithm 6** T-HUDS

**Input**: $B_i$, $k$, $winSize$, *HUDS-Tree*

**Output**: *Top-$k$ HUIs*

1: **if** HUDS-Tree is empty (i.e., $B_i$ is the very first batch $B_1$) **then**
2: $minTopKUtil_0 \leftarrow 0$
3: Construct a *HUDS-Tree* based on $B_i$ (i.e., $B_1$)
4: Construct the auxiliary list $maxUtilList$ based on the information in the *HUDS-Tree*
5: Initialize the auxiliary list $MIUList$ using the top-$k$ $miu$ values of the items (to be defined in later)
6: **else**
7: Call Algorithm 10 to update *HUDS-Tree*, $maxUtilList$ and $MIUList$ using $B_i$ and $winSize$
8: **end if**
9: **if** batch ID $i \geq winSize$ **then**
10: Call Algorithm 8 to compute top-$k$ HUIs on the current sliding window with the *HUDS-Tree*, $maxUtilList$, $MIUList$ and $minTopKUtil_{i-1}$
11: **end if**
12: Return Top-$k$ HUIs

Below we first describe how the *HUDS-Tree* is structured and constructed. Then we present our methods for estimating the minimum utility threshold, our top-$k$ HUI mining

algorithm and finally our procedure for updating the *HUDS-Tree*.

#### 5.1.3.2 HUDS-Tree Structure and Construction

The structure of *HUDS-Tree* is similar to that of *FP-tree* [30], *UP-tree* [57] or *HUS-tree* [7]. These trees are used to compress a transaction database into a tree. A non-root node in the trees represents an item in the transaction database, and a path from the root to a node compresses the transactions that contains the items on the path. Since the *FP-tree* is used to find frequent itemsets, a node in an *FP-tree* mainly stores the frequency of an itemset represented by the path from the root to the node. The *UP-tree* is for finding high utility itemsets, and thus its node contains not only frequency but also an estimated utility of the itemset. The *HUS-tree* is used for mining high utility patterns over data streams. Thus, its node stores the $TWU$ value of the itemset for each batch in a sliding window to facilitate the update process. Since we are dealing with data streams as well, our *HUDS-Tree* is similar to a *HUS-tree*. But instead of storing $TWU$ values, a node in a *HUDS-Tree* stores the $PrefixUtil$ of the represented itemset for each batch, which is, as discussed earlier, a closer estimate of the true utility of the itemset than $TWU$. In addition, to effectively estimate the minimum utility threshold, a node in *HUDS-Tree* also stores the $MTU$ value of the itemset for each batch. The node structure of the *HUDS-Tree* is described below.

A non-root node in a *HUDS-Tree* contains the following fields: *nodeName*, *node-*

*Counts*, *nodePUtils*, *nodeMTUs* and *succ*. *nodeName* is the name of the item represented by the node. The *nodeCounts* field is an array with $winSize$ elements, where $winSize$ is the number of batches in the sliding window. Each element in *nodeCounts* corresponds to a batch in the current sliding window and registers the number of the transactions in the batch falling onto the path from the root to the node. Let $X$ be the itemset represented by the path. The *nodePUtils* field is an array of $winSize$ elements, each corresponding to a batch and storing the prefix utility of $X$ in the transactions of the batch falling onto the path. Similarly, *nodeMTUs* is an array of the *minimum transaction utilities* ($MTU$) of $X$ in the transactions falling onto the path for all the batches of the sliding window. Keeping separate information for each batch facilitates the update process, that is, when a new batch $B_i$ arrives, if the oldest batch needs to be removed, it is easy to remove the information of the oldest batch and include the information for the new batch. Finally, *succ* points to the next node of the tree having the same *nodeName*.

**Example 3** *A* HUDS-Tree*, built from the transactions in sliding window* $SW_1 = \{B_1, B_2\}$ *in Figure 5.1, is illustrated in Figure 5.2, where the* $winSize$ *is 2 and thus* $nodeCounts$*,* $nodePUtils$ *and* $nodeMTUs$ *each contains two values. For example, in node* $\langle b : [0,1],[0,15],[0,3]\rangle$*,* $nodeName$ *is* $b$*,* $nodeCounts$ *holds [0,1], meaning the number of transactions matching path* $a \rightarrow b$ *is* $0$ *in* $B_1$ *and* $1$ *in* $B_2$*, respectively, and [0,15] and [0,3] are the contents of* $nodePUtils$ *and* $nodeMTUs$*, respectively. Since* $b$ *appears only in the second batch, its values for* $nodeCounts$*,* $nodePUtils$ *and* $nodeMTUs$ *in*

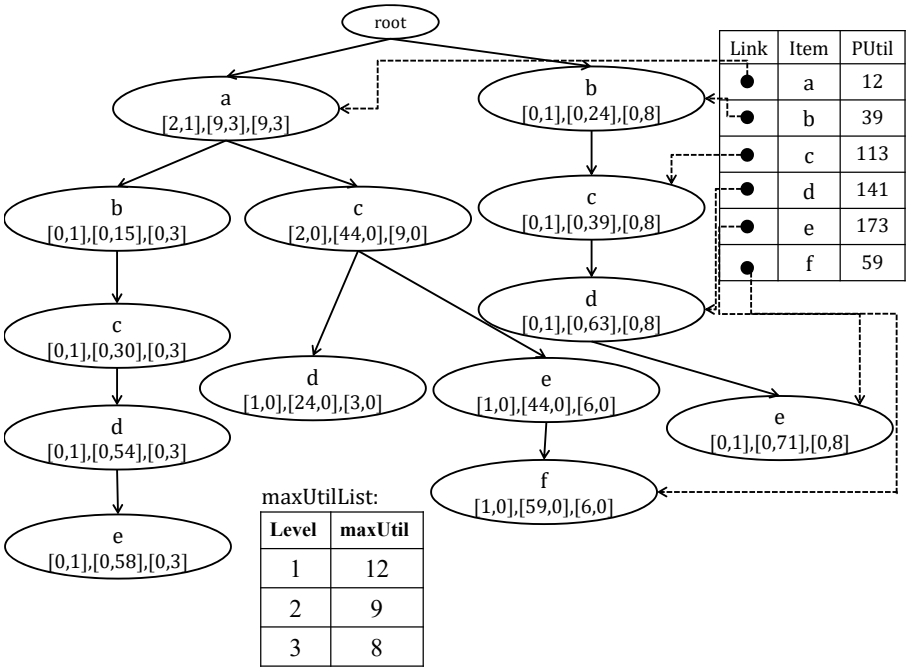



Figure 5.2: HUDS-Tree after inserting transaction in $SW_1$ in Figure 5.1.

*the first batch are* 0*. The field* $succ$ *is not illustrated for the clarity reason.*

Each item has an entry in the header table of the *HUDS-Tree*. An entry in the header table contains the name of the item, the $PrefixUtil$ value of the item in the transactions represented by the tree and a *link* pointing to the first node in the *HUDS-Tree* carrying the item. The $PrefixUtil$ value of an item is computed by adding up all the $nodePUtils$ values of the nodes labeled with the item in the tree.

**Algorithm 7** Insert Transaction into HUDS-Tree

**Input**: Transaction $T$, $rootNode$, $idx$, $batchNumber$

**Output**: Updated *HUDS-Tree*, maxUtilList

1: let $item_{idx}$ be the $idx$th item in $T$

2: **if** $\exists$ $node \in$ the children of the $rootNode$ & $nodeName(node) = item_{idx}$ **then**

3: $node.nodePUtils[batchNumber] += \sum_{j=1}^{idx} u(item_j, T)$

4: $node.nodeCounts[batchNumber] ++$

5: $node.nodeMTUs[batchNumber] += MTU(T)$

6: **else**

7: $node.nodeName \leftarrow item_{idx}$

8: $node.nodePUtils[batchNumber] \leftarrow \sum_{j=1}^{idx} u(item_j, T)$

9: $node.nodeCounts[batchNumber] \leftarrow 1$

10: $node.nodeMTUs[batchNumber] \leftarrow MTU(T)$

11: add $node$ as a child node of $rootNode$

12: **end if**

13: update the $idx$th element, $maxUtil_{idx}$, in the $maxUtilList$

14: **if** $idx \neq$ the length of $T$ **then**

15: $Algorithm7(T, node, idx + 1, batchNumber)$

16: **end if**

17: HUDS$-Tree \leftarrow rootNode$

18: Return HUDS$-Tree$, $maxUtilList$

Given the first batch $B_1$ of transactions, a ***HUDS-Tree*** is constructed as follows. For each transaction in $B_1$, we first arrange the items in the transaction in an order (such as the lexicographic order or the descending external item utility order) , and then insert the items into the HUDS-Tree in the way similar to building an FP-tree [30]. For example, for the first item $item_1$ in a transaction $T$ in $B_1$, if a node with the same item name is not found under the root, a new child is created and its fields are initialized as follows: $nodeName = item_1$, $nodePUtils[1] = u(item_1, T)$, $nodeCounts[1] = 1$, $nodeMTUs[1] = MTU(T)$. If the node with the item name already exists under the root, its fields for the current batch are updated. Details of the procedure for inserting one transaction $T$ in batch $B_i$ into the ***HUDS-Tree*** are presented in Algorithm 7. In the algorithm, the input parameter $batchNumber$ should be given a value of $i\%winSize+1$, where $i$ is the ID of the current batch $B_i$ in the data stream and $\%$ is the modulo operator which returns the remainder of dividing $i$ by $winSize$. For example, if $i = 2$ or $winSize + 2$, $batchNumber$ is 2. The algorithm is a recursive algorithm. Each call to the algorithm "inserts" one item of the input transaction $T$ into the tree. The input parameter $idx$ indicates which item in $T$ is being "inserted". $idx$ is initialized to 1 for each transaction. Clearly, the tree can be built with one scan of the data in $B_i$.

Before we describe how to mine HUIs from a ***HUDS-Tree*** and how to update the tree with new batches, we first present our method for estimating the minimum utility threshold.

#### 5.1.3.3 Estimation of Minimum Utility Threshold

Our objective is to find top-$k$ high utility itemsets. Since the number of itemsets is exponential with respect to the number of items in the data, it is infeasible to enumerate all the itemsets, find their utilities in the sliding window and outputs the top-$k$ highest utility itemsets. An efficient procedure for finding top-k itemsets is to first use an efficient method to find potential itemsets whose utility is above a threshold and then identify the top-$k$ itemsets from the potential ones [61]. To do this, a proper minimum utility threshold is needed in the first phase of the procedure. If the threshold is set too low, many unwanted HUIs are produced, which is time-consuming. If it is set too high, we may not be able to produce $k$ itemsets. A good strategy for setting the threshold should satisfy the following conditions: (1) it should not miss any top-k *HUIs*; (2) the estimated threshold should be as close as possible to the utility of the $kth$ highest utility itemset.

In our method, we use four strategies to initialize and dynamically adjust the threshold during the mining process. These strategies lead to significant pruning of search space. Below we describe three strategies, which will be used in the first phase of our mining method. The fourth strategy (to be used in the second phase) will be described in Section 5.1.3.4.

**Initializing the Threshold Using $maxUtilList$:** In a *HUDS-Tree*, the $nodeMTUs$ field of a node $n$ stores the $MTU$ values of the itemset represented by the path from

the root to $n$ in the set of transactions falling onto the path in each batch separately. The $MTU$ value of the itemset in the transactions on the path in the sliding window can be easily calculated by summing up all the values in $nodeMTUs$ of node $n$. We use $nodeMTU(n)$ to denote this sum. Similarly, $nodeCount(n)$ is used to denote the count of the itemset in the set of the transactions falling on the path in the whole sliding window. Now we are ready to define the $maxUtilList$.

**Definition 41 (Maximum Utility List (maxUtilList))** of a *HUDS-Tree* is a list of length $d$:

$$maxUtilList = \{maxUtil_1, ..., maxUtil_d\}$$

where $d$ is the depth of the *HUDS-Tree* and $maxUtil_i$ is computed based on the nodes on the $i$th level of the tree as follows:

$$\begin{aligned} maxUtil_i &= \max_j\{\max(minProfit(node_{i,j}) \times nodeCount(node_{i,j}), \\ &\quad nodeMTU(node_{i,j}))\} \end{aligned}$$

where $node_{i,j}$ is the $j$th node in level $i$ of the tree, $minProfit(node_{i,j}) = \min\{p(item)|item \in X\}$ where $p(item)$ is the external utility of the $item$ and itemset $X$ is formed by the path from the root to $node_{i,j}$ in the tree, $nodeCount(node_{i,j})$ is the sum of the counts in the $nodeCounts$ field of $node_{i,j}$ (i.e., the total number of transactions in the sliding window that have prefix $X$), and $nodeMTU(node_{i,j})$ is sum of the values in the $nodeMTUs$

field of $node_{i,j}$ (i.e., the total $MTU$ value of itemset $X$ in all the transactions of the sliding window that have prefix $X$).

For example, assume that the root is at level 0 in Figure 5.2. The level 2 has one $b$ node and two $c$ nodes. $maxUtil_2$ is thus computed as:

$$\begin{aligned} maxUtil_2 &= \max\{\max(3 \times nodeCount(b), nodeMTU(b)), \\ &\qquad \max(3 \times nodeCount(c), nodeMTU(c)), \\ &\qquad \max(5 \times nodeCount(c), nodeMTU(c))\} \\ &= \max\{\max(3 \times 1, 3), \max(3 \times 2, 9), \max(5 \times 1, 8)\} = 9. \end{aligned}$$

**Lemma 9** *Let $util_k$ be the utility of the $k$th itemset in the top-k high utility itemset list. $util_k$ is no less than $maxUtil_L$ where $L = \lceil \log_2(k+1) \rceil$.*

**Proof**

Let's call $nodeCount(node_{i,j}) \times minProfit(node_{i,j})$ ***Lowest Profit Item utility (LPI)*** of the itemset $X$ formed by the path from the root to $node_{i,j}$ in the set $S$ of transactions represented by the path. Clearly, $LPI(X)$ is another underestimate of the utility of $X$ in $S$, i.e., $LPI(X) \leq u(X)$ on $S$. Also, for all $Y \subseteq X$, $LPI(Y) \geq LPI(X)$ on $S$.

Let $node_{L,j}$ be a node on level $L$ of the tree, $X_{L,j}$ denote the itemset formed by the path from the root to $node_{L,j}$, and $S_{L,j}$ denote the set of transactions falling onto the path. Assume that $node_{L,j}$ is the node with $maxUtil_L$, that is, $maxUtil_L$ is either $nodeMTU(node_{L,j})$ (i.e., $MTU(X_{L,j})$ on $S_{L,j}$) or $LPI(X_{L,j})$ on $S_{L,j}$.

Assume that $Y$ is a subset of $X_{L,j}$. According to Lemma 8, $MTU(Y) \geq MTU(X_{L,j})$ on set $S_{L,j}$. According to Property 7, $u(Y) \geq MTU(Y)$ on $S_{L,j}$. Similarly, $u(Y) \geq LPI(Y) \geq LPI(X_{L,j})$ on $S_{L,j}$. Thus,

$$u(Y) \geq max(nodeMTU(node_{L,j}), LPI(X_{L,j})) = maxUtil_L.$$

Since $u(Y)$ on the entire dataset represented by the tree is no less than $u(Y)$ on $S_{L,j}$. Thus, $u(Y)$ on the entire dataset is no less than $maxUtil_L$.

Since $node_{L,j}$ is at level $L$ of the tree, $X_{L,j}$ contains $L$ items (assuming the root is at level 0). Thus, $X_{L,j}$ has $2^L - 1$ subsets. Thus, there are at least $2^L - 1$ itemsets whose utility is no less than $maxUtil_L$.

If $L = \lceil \log_2(k+1) \rceil$, we have

$$L \geq \log_2(k+1) \Rightarrow 2^L \geq k+1 \Rightarrow 2^L - 1 \geq k$$

Thus, there are at least $k$ itemsets with utility higher than or equal to $maxUtil_L$. Thus, $util_k$ is no less than $maxUtil_L$.

□

Lemma 9 declares that $maxUtil_L$ can be used to set the minimum utility threshold for finding top-$k$ HUIs, where $L = \lceil \log_2(k+1) \rceil$. No top-$k$ HUIs can be missed with such a threshold. Intuitively, $maxUtil_L$ is the maximum value among the $nodeMTU$ values and $LPI$ values of the nodes on level $L$ of the tree.

The $maxUtilList$ can be computed while constructing and updating the ***HUDS-Tree***.

If $k$ is fixed, only $maxUtil_L$ needs to be computed in the list; otherwise, the values of $maxUtil_i$ for all the levels are maintained.

**Adjusting the Threshold Using $MIUList$:** $MIUList$ is another list that we maintain to dynamically adjust the minimum utility threshold. It keeps the top-$k$ *minimum itemset utility* ($MIU$) values of current potential high utility itemsets. Below we first define the concept of MIU [61]:

**Definition 42 Minimum Item Utility of an item $a$ in any transaction of a dataset $D$** is defined as: $miu_D(a) = u(a, T_q)$ where $T_q \in D$ and $\neg\exists\ T_p \in D$ such that $u(a, T_p) < u(a, T_q)$.

**Definition 43 Minimum Itemset Utility of an itemset** $X$ in a dataset $D$ is defined as: $MIU_D(X) = \sum_{a_i \in X} miu_D(a_i) \times SC_D(X)$ where $SC_D(X)$ is support count of $X$ in $D$.

We use $MIU(X)$ to denote $MIU_D(X)$ when the dataset $D$ is clear in the context.

**Property 3** *For any itemset $X$ in dataset $D$, $MIU_D(X) \leq u_D(X)$.*

The $miu$ value of an item can be computed during the ***HUDS-Tree*** construction and update. It can be stored in the global header table of the ***HUDS-Tree***. The $MIU$ value of an itemset can be computed based on the $miu$ values of its elements and the support count of the itemset (maintained in the $nodeCounts$ fields). In [61], the $MIU$ values of itemsets are used to raise the minimum support threshold during the HUI mining process.

But they may not be used properly. We use them to adjust the minimum utility threshold by maintaining a *minimum itemset utility list* defined as follows.

**Definition 44 Minimum Itemset Utility List(MIUList)** *Given a set of already-generated HUIs,* $MIUList$ *contains the top-*$k$ *list of the* $MIU$ *values of these HUIs, ranked in* $MIU$*-descending order, denoted as* $MIUList = \{MIU_1, MIU_2, ...., MIU_k\}$*, where* $MIU_1 \geq MIU_2 \cdots \geq MIU_k$.

**Lemma 10** *Let* $MIU_k$ *be the* kth *member of* MIUList *and* $util_k$ *be the utility of the* $k$*th highest utility itemset in the top-*$k$ *HUI list.* $util_k$ *is no less than* $MIU_k$.

**Proof**

Assume that the $MIU_i$ values in $MIUList$ are the $MIU$ values of itemsets $X_1$, $X_2$, $\ldots X_k$, respectively. According to Property 3, we have:

$$\forall X_i \in \{X_1, X_2, \ldots X_k\}, MIU(X_i) \leq u(X_i).$$

According to the Definition 44, $MIU_k$ is the smallest value in the $MIUlist$. Thus, there are at least $k$ itemsets whose utility is no less than $MIU_k$.

□

According to this lemma, if the minimum utility threshold is set to $MIU_k$, no top-$k$ HUI will be missed. Thus, we have the following strategy for adjusting the threshold. Once the *HUDS-Tree* is built or updated for a sliding window $SW_i$, $MIUList$ is initialized to the top-$k$ highest $miu$ values of single items. During the process of mining HUIs

for window $SW_i$, once a new potential HUI is generated, its $MIU$ is compared with the current $MIU_k$. If it is greater than the current $MIU_k$, the new MIU value is inserted into the $MIUList$. If the new $MIU_k$ is greater than the current minimum utility threshold, then the threshold can be raised to the new $MIU_k$.

**Adjusting the Threshold with $minTopKUtil$ of Last Window:** Our third strategy for adjusting the minimum utility threshold is to make use of the utility values of the top-$k$ HUIs in the last sliding window. For this, we define the *minimum top-k utility* ($minTopKUtil$) of a sliding window as follows.

**Definition 45** Let $SW_i = \{B_i, B_{i+1}, \ldots, B_{i+winSize-1}\}$ be the $i$th sliding window and let $TopkHUISet_i$ denote the set of top-$k$ HUIs in window $SW_i$. The minimum top-$k$ utility of a sliding window $SW_i$ is defined as:

$$minTopKUtil_i = \min_{itemset \in TopkHUISet_i} \sum_{j=i+1}^{i+winSize-1} u_{B_j}(itemset)$$

In other words, the $minTopKUtil$ of sliding window $SW_i$ is the minimum of the utilities of the itemsets in $TopkHUISet_i$ in the last $winSize - 1$ batches of $SW_i$.

**Lemma 11** *Let $util_k$ be the utility of the $k$th highest utility itemset over sliding window $SW_{i+1}$, and $minTopKUtil_i$ be the minimum top-$k$ utility of window $SW_i$. We have $util_k \geq minTopKUtil_i$.*

**Proof**

Let $B$ be the union of last $winSize-1$ batches in window $SW_i$. Then the next sliding window $SW_{i+1} = B \cup B_{new}$ where $B_{new}$ is the new batch in $SW_{i+1}$. Since $B \subset SW_{i+1}$, for each itemset $X$ in $TopkHUISet_i$, $u_B(X) \leq u_{SW_{i+1}}(X)$. Since $minTopKUtil_i \leq u_B(X)$ for all $X \in TopkHUISet_i$ and there are $k$ itemsets in $TopkHUISet_i$, there are at least $k$ itemsets whose utility in $SW_{i+1}$ is at least $minTopKUtil_i$.

□

According to this lemma, if the minimum utility threshold in window $SW_{i+1}$ is set to $minTopKUtil_i$, no top-$k$ high utility itemsets will be missed.

The $minTopKUtil_i$ value is computed during the second phase of our procedure for mining top-$k$ HUIs from sliding window $SW_i$, which is to be described in Section 5.1.3.4.

#### 5.1.3.4 Mining Top-$k$ High Utility Itemsets

After a ***HUDS-Tree*** is built or updated for a sliding window $SW_i$, we use a 2-phase procedure to find top-$k$ HUIs in $SW_i$. In the first phase, the ***HUDS-Tree*** is mined to generate a set of potential top-$k$ high utility itemsets (i.e., $PTKHUI$s) that satisfy a dynamically-changing minimum utility threshold. In the second phase, the exact utilities of the $PTKHUI$s are computed and the top-$k$ high utility itemsets are returned.

This 2-phase procedure is shown in Algorithm 8. At the beginning of the procedure, we initialize the minimum utility threshold, min_util, according to the strategies proposed in Section 5.1.3.3 as follows:

$min\_util = \max\{maxUtil_L, MIU_k, minTopKUitl_{i-1}\}$,

where $minTopKUitl_{i-1}$ is the minimum top-$k$ utility of the last sliding window (initialized to 0 in Algorithm 6 if the new batch is the first one), $maxUtil_L$ is the $L$th element in $maxUtilList$ (where $L$ is computed in Line 1), and $MIU_k$ is the $k$th element of the $MIUList$ that initially contains the list of the top-k minimum item utilities ($miu$) of single items.

With this initial $min\_util$ threshold, Algorithm 9 is called to find PTKHUIs from the *HUDS-Tree* (Line 3). This is the first phase of the top-$k$ procedure. The second phase (from Line 4 to the end) finds exact top-$k$ HUIs from the set of PTKHUIs. Below we describe each phase in detail.

**Phase I: Discover $PTKHUI$s from *HUDS-Tree*:** In Phase I, a set of potential top-$k$ HUIs ($PTKHUI$s) is found from the *HUDS-Tree*. Our objective in this phase is to find as few $PTKHUI$s as possible (so that the second phase will be faster) while not missing any top-$k$ HUIs. Our procedure for this phase follows a pattern growth approach, similar to *FP-growth* [30] and $HUPMS$ [7].

**Algorithm 8** Top-$k$ HUI Mining - Part - 1

**Input**: $HUDS\text{-}Tree$, $maxUtilList$,$MIUList$, $minTopKUtil_{i-1}$, $k$, $SW_i$

**Output**: $TopkHUISet$ , $minTopKUtil_i$

1: $L \leftarrow \lceil \log(k+1) \rceil$

2: $min\_util \leftarrow \max\{maxUtil_L, MIU_k, minTopKUtil_{i-1}\}$

3: Generate a set of potential top-$k$ HUIs (PTKSet) by calling Algorithm 9 with $min\_util$. The $min\_util$ is also dynamically updated in Algorithm 9

4: Scan the transactions in the current sliding window $SW_i$ to obtain $u_{SW_i}(itemset)$ and $u_{SW_i - B_i}(itemset)$ for each $itemset$ in $PTKSet$, where $B_i$ is the first batch in $SW_i$.

5: $TopkHUISet \leftarrow \emptyset$

6: **for** each $itemSet \in PTKSet$ **do**

7: **if** $u_{SW_i}(itemSet) \geq min\_util$ **then**

8: Insert $\langle itemSet, u_{SW_i}(itemSet) \rangle$ into $TopkHUISet$ so that the elements in $TopkHUISet$ are ranked in the utility-descending order

9: **if** the size of $TopkHUISet > k$ **then**

10: Remove the last element from $TopkHUISet$

11: **if** $u_{SW_i}(lastItemSet) > min\_util$ where $lastItemSet$ is the current last itemset in $TopkHUISet$ **then**

12: $min\_util \leftarrow u_{SW_i}(lastItemSet)$

13: **end if**

*Top-$k$ HUI Mining - Part - 2*

14: **end if**

15: **end if**

16: **end for**

17: $minTopKUtil_i \leftarrow \min\{u_{SW_i-B_i}(itemset)|itemset \in TopkHUISet\}$

18: Return $TopkHUISet$, $minTopKUtil_i$

The major differences between our Phase I procedure and the others are as follows. First, we use both $PrefixUtil$ and local $TWU$s to prune the search space, while others for HUI mining mainly use $TWU$. Second, we use effective strategies for initializing and dynamically adjusting the $min\_util$ threshold during the mining process.

The pseudocode of the *HUDS-Tree* mining procedure is described in Algorithm 9. Like *FP-growth*, the algorithm is a recursive algorithm. In the first call to the procedure, the input *HUDS-Tree* is the global tree, and the itemset $X$ in the input list is empty. In a recursive call, the input tree is the $X$-conditional *HUDS-Tree* where $X$ is a non-empty itemset. The algorithm works as follows. For each item $t$ in the (conditional) header table, the algorithm checks if the $PrefixUtil$ of $t$ satisfies the $min\_util$ threshold (Line 2). If yes, a potential top-$k$ HUI $IS$ is generated by extending $X$ with item $t$. $IS$ is then added into the potential top-$k$ HUI set (i.e., *PTKSet*). Then, the $min\_util$ threshold is adjusted in lines 5 to 7. If *MIU(IS)* is more than the current $min\_util$, the $MIU$ value is inserted into $MIUList$ and $min\_util$ is raised by the minimum value of $MIUList$. $MIU(IS)$ can be computed easily because $SC_{SW_i}(IS)$ can be computed

**Algorithm 9** HUDS-Tree Mining to Generate PTKHUIs (Phase I)

**Input**: $HUDS\text{-}Tree$, itemset $X$, $min\_util$, $MIUList$, $k$

**Output**: $PTKSet$, $min\_util$,$MIUList$

1: **for** each item $t$ in the header table of *HUDS-Tree* **do**

2: **if** $PrefixUtil(t) \geq min\_util$ **then**

3: Generate a potential top-k itemset: $IS \leftarrow \{t\} \cup X$

4: Add $IS$ into the $PTKSet$ set

5: **if** $MIU_{SW_i}(IS) \geq min\_util$ **then**

6: Insert $MIU_{SW_i}(IS)$ into the $MIUList$

7: $min\_util \leftarrow MIU_k$

8: **end if**

9: $Pattern\_base_{IS} \leftarrow$ all prefix paths of the nodes for item $t$ with their utilities

10: Prune all items in the $Pattern\_base_{IS}$ whose $TWU$ in $Pattern\_base_{IS}$ is less than $min\_util$.

11: Construct conditional HUDS$-$Tree$_{IS}$ and its header table

12: **if** $HUDS\text{-}Tree_{IS}$ is not empty **then**

13: call Algorithm 9($HUDS\text{-}Tree_{IS}$,$IS$, $min\_util$, $MIUList$, $k$)

14: **end if**

15: **end if**

16: **end for**

17: Return $PTKSet$, $min\_util$,$MIUList$

using the $nodeCounts$ fields of the $t$ nodes and the $miu$ values of all the items have already been computed when building the global *HUDS-Tree*.

After $IS$ is generated, to find longer PTKHUIs containing $IS$, $IS$-conditional pattern base ($Pattern\_base_{IS}$) is built by enumerating all the prefix paths of the $t$ nodes in the tree. The utility of each prefix path is the sum of the values in the $nodePUtils$ field of the $t$ node in that path. Each item's local $TWU$ value can then be computed by adding up the utilities of the prefix paths it is in. In Line 9, we eliminate items in the conditional pattern base whose local $TWU$ is less than the $min\_util$ threshold. After that, the $IS$-conditional *HUDS-Tree* is constructed based on the conditional pattern base with the remaining items. At the end of tree construction, all the $nodePUtils$ values of nodes with the same $nodeName$ in the conditional tree are added and the result is added to local header table as the $PrefixUtil$ value of the item. Once a conditional tree is built, Algorithm 9 is called recursively to discover longer *PTKHUI*s ending with $IS$.

In the performance evaluation section, we will show that this pattern-growth procedure generates fewer potential top-$k$ HUIs and has less overall run time than the state-of-the-art algorithms for high utility itemset mining. This is due to the use of the prefix utility in pruning the search space and also the dynamical increase of $min\_util$ during the mining process.

**Phase II: Identifying Top-$k$ HUIs from PTKHUIs:** *HUDS-Tree* is a compact repre-

sentation of the transactions in a sliding window. It allows the use of the pattern growth method to efficiently find the potential top-$k$ HUIs. However, since the quantity of an item inside a transaction may vary among transactions, the exact utility of an itemset cannot be obtained from the *HUDS-Tree*. Thus, in this second phase, we scan the transactions in the current sliding window to obtain the exact utility of each potential top-$k$ HUI, and then identify the top-$k$ HUIs based on the true utility of the PTKHUIs.

The second phase procedure is shown in Lines 4-12 of Algorithm 8. In Line 4, it scans the transactions in the current sliding window $SW_i$ to obtain the exact utility of each itemset in $PTKSet$ in $SW_i$ and also the exact utility of each itemset in the last $winSize - 1$ batches of $SW_i$. From Line 6 to Line 12, top-$k$ HUIs are identified using a *selected insertion sort*, in which only the itemsets whose utility is no less than $min\_util$ are inserted to the top-$k$ list (denoted as $TopkHUISet$). $TopkHUISet$ is maintained to have no more than $k$ elements, ranked in utility-descending order. In addition, if $TopkHUISet$ contains $k$ elements, $min\_util$ is adjusted dynamically to be the utility of the $k$th itemset in $TopkHUISet$ (Lines 11 and 12). We call this adjustment our fourth strategy for increasing the $min\_util$ threshold.

Finally, in Line 13 of the algorithm, the minimum top-$k$ utility of the current sliding window ($SW_i$) is set to minimum utility value of the itemset in $TopkHUISet$ in the last $winSize - 1$ batches of $SW_i$. This is for adjusting the $min\_util$ threshold for mining tip-$k$ HUIs in the next sliding window $SW_{i+1}$.

**Theorem 5** *Given a sliding window $SW_i$, if $X$ is among the top-k high utility itemsets, it is returned by Algorithm 6.*

**Proof**

We prove the theorem by showing that the $min\_util$ in our algorithm is never over the exact utility of the $k$th highest utility itemset in the current sliding window, and also that our *HUDS-Tree* mining procedure does not prune out any itemset whose true utility is greater than $min\_util$.

Let $util_k$ be the exact utility of the $k$th highest utility itemset for sliding window $SW_i$. In our algorithms, the $min\_util$ is set or adjusted in the following three places:

- In Line 2 of Algorithm 8:

  $min\_util = \max\{maxUtil_L, MIU_k, minTopKUtil_{i-1}\}$

  where $L = \lceil \log_2(k+1) \rceil$.

  According to Lemmas 9, 10, and 11, $maxUtil_L \leq util_k$, $MIU_k \leq util_k$ and $minTopKUtil_{i-1} \leq util_k$. Thus, $min\_util$ is no larger than $util_k$.

- In Lines 5-7 of Algorithm 9, $min\_util$ is dynamically adjusted to $MIU_k$, which is the $kth$ highest MIU value of the already generated potential top-$k$ HUIs. According to Lemma 10, $MIU_k \leq util_k$. Thus, $min\_util \leq util_k$.

- In Lines 11-12 in Algorithm 8, $min\_util$ is dynamically adjusted to the lowest utility of the current top-$k$ HUI set. Thus, $min\_util$ is no larger than $util_k$.

Below we show that our *HUDS-Tree* mining procedure for generating potential top-$k$ HUIs (i.e., Algorithm 9) does not miss any top-$k$ HUIs. There are two places where we prune the search space in Algorithm 9.

- In Line 2, if the PrefixUtil of an item $t$ is less than $min\_util$, item $t$ will not be added to itemset $X$ to form longer HUI containing $\{t\} \cup X$. The $PrefixUtil$ of $t$ in the (conditional) header table is actually $PrefixUtil(\{t\} \cup X)$ (according to how it is computed). Assume $X = Y \cup \{i\}$ where $i$ is the last item in $X$ in the item order for building the *HUDS-Tree*. Then $\{t\} \cup X = \{t\} \cup Y \cup \{i\}$. According to Lemma 6, $PrefixUtil(\{t\} \cup Y \cup \{i\}) \geq PrefixUtil(S \cup \{t\} \cup Y \cup \{i\})$ where $S$ is a set of items containing the items ranked before $t$ in the item order for building the tree. Thus, if $PrefixUtil(\{t\} \cup Y \cup \{i\}) < min\_util$, $PrefixUtil(S \cup \{t\} \cup Y \cup \{i\}) < min\_util$. This means that if the $PrefixUtil$ of $t$ in the header table is less than $min\_util$, there is no need to check any itemsets whose "suffix" is $\{t\} \cup X$.

- In Line 9 of the algorithm, we prune out all the items whose local $TWU$ is less than $min\_util$. Since $TWU$ has the downward closure property, the pruning does not miss any itemsets whose $TWU$ is no less than $min\_util$.

Both $PrefixUtil$ and $TWU$ are over-estimates of the true utility of an itemset. If an over-estimate is less than $min\_util$, the true utility must be less than $min\_util$. Thus, if an itemset is pruned by $PrefixUtil$ or $TWU$, its true utility must be less than $min\_util$.

Thus, no itemsets whose utility $\geq min\_util$ is pruned by the algorithm. Since $min\_util$ is never over $util_k$, no top-$k$ HUI is missed by our algorithms.

□

The following example illustrates how the proposed strategies are applied during the mining process.

**Example 4** *Given the transactions in Figure 5.1, let* $winSize = 2$ *and* $k = 5$*. Once the first window arrives, a complete* HUDS-Tree *is constructed (Figure 5.2). Since no candidate is generated before learning from the first window,* $MIUList$ *is initialized by the five most largest* $miu$ *values of the items. Therefore,* $MIUList = \{16, 15, 12, 5, 4\}$*. Also,* $maxUtilList$ *is built during the tree construction and updating:* $maxUtilList = \{12, 9, 8\}$*, where the length of* $maxUtilList$ *is* $\lceil \log_2(k+1) \rceil = 3$*. Since this window is the first window,* $minTopKUtil_0 = 0$*. Thus, the initial minimum utility threshold (*$minUtil$*) is computed as follows:*

$$minUtil = \max(maxUtil_3, MIU_5, minTopKUtil_0) = \max(8, 4, 0) = 8.$$

*During the candidate generation in Algorithm 8,* $MIUList$ *is updated based on the* $miu$ *values of each new candidate. At the end of candidate generation,* $MIUList = \{48, 40, 38, 37, 36\}$*.* $minUtil$ *is then updated to 36 (*$minUtil = \max(8, 36, 0)$*). After the second phase the first set of top-*$5$ *high utility itemsets are discovered as follows:*

$$\{(bcd, 114), (cd, 99), (bcde, 126), (cde, 90), (bde, 96)\}$$

*Next, we compute the minimum Top-$k$ utility of the 1st sliding window ($minTopKUtil_1$) as follows. The exact utilities of top-5 high utility itemsets in the first sliding window are:*

$$\{(bcd, 114), (cd, 78), (bcde, 126), (cde, 90), (bde, 95)\}$$

*Hence, $minTopKUtil_1 = 78$. This value is used to help initialize the minimum utility threshold ($minUtil$) for the next sliding window. The process of initializing and updating $minUtil$ for the second and later sliding windows is the same as the one for the first window.*

#### 5.1.3.5 HUDS-Tree Update

When a new batch of transactions arrives, the *HUDS-Tree* needs to be updated to represent the transactions in the new sliding window. This involves removing from the tree the information of the oldest batch in the last window (if the last window was full) and adding to the tree the transactions in the new batch. Algorithm 10 describes this update process.

In Line 1, the index of the batch in the tree node fields is computed as $batchNumber = i\%winSize + 1$, where $i$ is the new batch ID (assuming the very first batch in the data stream is $B_1$), and $winSize$ is the maximum number of batches in a sliding window. The information about the new batch will be put into the $batchNumber$th slots in the $nodeCounts$, $nodePUtils$ and $nodeMTUs$ fields of the tree nodes. In Lines 2 to 8,

if the new batch ID (i.e., $i$ in $B_i$) is greater than the size of the sliding window (which means that the last sliding window was full), then the information about the oldest batch is removed by changing $nodeCounts[batchNumber]$, $nodePUtils[batchNumber]$ and $nodeMTUs[batchNumber]$ in each node to zero. If the sum of the values in $nodePUtils$ for all the remaining batches is zero in a node, the node and the subtree rooted at the node are removed (Line 7). Then, the transactions in the new batch are inserted into the tree one by one by calling Algorithm 7. $batchNumber$ is passed to Algorithm 7 so that the information about the new batch will be stored the $batchNumber$th slots in the node fields. In Algorithm 7, $maxUtilList$ is also updated. After all the transactions are inserted into the tree, the prefix utilities of each item is updated in Line 12. Finally, the $MIUList$ is updated as described in Line 13.

**Algorithm 10** HUDS-Tree-Update - Part 1

**Input**: $HUDS\text{-}Tree$, new batch $B_i$, $k$

**Output**: $HUDS\text{-}Tree$, $maxUtilList$,$MIUList$

1: $batchNumber \leftarrow i\%winSize + 1$

2: **for** each node in *HUDS-Tree* **do**

3: $nodeCounts[batchNumber] \leftarrow 0$

4: $nodePUtils[batchNumber] \leftarrow 0$

5: $nodeMTUs[batchNumber] \leftarrow 0$

6: **if** $\forall i$ $(1 \leq i \leq winSize)$ $nodePUtils[i] = 0$ **then**

7: remove the node and its subtree from the tree

8: **end if**

9: **end for**

10: **for** every $T \in B_i$ **do**

11: $\{HUDS\text{-}Tree, maxUtilList\} \leftarrow$ Algorithm7($T$, *HUDS-Tree*, root of *HUDS-Tree*, 1, $batchNumber$)

12: update the $miu$ value of each item in $T$

13: **end for**

14: Update the $PrefixUtil$ value of each item in the header table by summing up all the values in the $nodePUtils$ fields of all the nodes for the item in the tree.

*HUDS-Tree-Update - Part 2*

15: Update $MIUList$ by (1) computing the $MIU$ value of each item in the header table using the $miu$ value of the item and the $nodeCounts$ values in all the nodes for the item and (2) select the top-$k$ $MIU$ values.

16: **Return** $HUDS\text{-}Tree$, $maxUtilList$, $MIUList$

## 5.2 Top-k High Utility Sequential Pattern Mining over Data Streams

In this section, we propose a sliding window-based method, called *T-HUSP*, to find top-k high utility sequential patterns over data streams. This method is an extended version of our proposed method in Chapter 4 (i.e., *HUSP-Stream*), for discovering high utility sequential patterns over sliding windows.

Although HUSP-Stream can discover high utility sequential patterns based on a given minimum utility threshold efficiently, it is difficult to set an appropriate minimum utility threshold for the following reasons. First, a large number of patterns in a data stream are needed to be analyzed before a proper utility threshold can be determined. Second, the set of HUSPs in a data stream may change over time, hence it is difficult or impossible to choose a proper utility threshold from a dynamic set of patterns. Third, in high utility sequential pattern mining, there are multiple factors such as the distribution of the items and utilities, density of the database and lengths of the sequences which make the mining process challenging. For example, it is possible that using a same minimum util-

ity threshold, some datasets produce a large amount of patterns while others contribute nothing. Fourth, if a large threshold is chosen to mine HUSPs, a large number of patterns might be produced; if a small threshold is set, very few or no high utility sequential patterns might be discovered. Consequently, in practice, it is more interesting for users to mine HUSPs whose utilities are sorted in the top $k$ order, instead of giving a predefined threshold. In this case, the utility threshold varies with time.

Although mining top-k HUSPs over data streams is very desirable, addressing this topic is not an easy task due to the following challenges.

- We need to overcome the large search space problem that inherits from combinatorial explosion of sequences. In comparison to top-k high utility itemset mining method proposed in Section 5.1, top-k high utility sequential pattern mining is more general but more challenging because it finds out not only all high utility itemsets but also their sequential orders.

- In many stream mining applications, users are more interested in finding top-k patterns in the most recent data. To satisfy this requirement, new records are added to the databases and uninteresting/old ones are removed from the databases. However, it is very inefficient to apply existing algorithms designed for static databases [67] to rerun the whole mining process on updated databases whenever a record is added to or deleted from the databases.

- When mining top-k HUSPs in a sliding window, the set of patterns discovered in the sliding window varies as the sliding window moves forward, and the utility of the k-th high utility pattern, denoted by $su_k$, varies with time. Therefore, it is a challenge to accurately set the value of $su_k$ and efficiently mine the top-k HUSPs.

In this section, we propose a new method, called *T-HUSP*, for *mining recent top-k high utility sequential patterns over data streams* to address all of the above challenges. To our best knowledge, this topic has not been addressed so far. Our contributions are summarized as follows.

- We propose a sliding window-based method for mining top-k high utility sequential patterns over data streams. To the best of our knowledge, existing methods for mining HUSPs over data streams do not address the issue of mining top-k HUSPs, and previous top-k HUSP mining methods do not work on data streams.

- We propose several strategies for initializing and dynamically adjusting the minimum utility threshold during the top-k HUSP mining process. We prove that using these strategies will not miss any top-k HUSPs.

- We conduct extensive experiments on both real and synthetic datasets to evaluate the performance of the proposed algorithm. Experimental results show that T-HUSP serves as an efficient solution for the problem of top-k HUSP mining over data streams.

**(a) A Data Stream of Itemset-Sequences**

| SID \ TID | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | ... |
|---|---|---|---|---|---|---|---|---|
| $S_1$ | {(a,2)(b,3)(c,2)} | {(b,1)(c,1)(d,1)} | | {(c,3)(d,1)} | | | {(b,1)(c,3)(d,2)} | ... |
| $S_2$ | | | {(b,4)} | | {(a,4)(b,5)(c,1)} | | | ... |
| $S_3$ | | | | | | {(a,2)(b,5)(e,2)} | | ... |

$SW_1$

$SW_2$

**(b) Profit Table**

| **Item** | **a** | **b** | **c** | **d** | **e** |
|---|---|---|---|---|---|
| **Profit** | 2 | 3 | 1 | 4 | 3 |

Figure 5.3: (a) An example of a data stream of itemset-sequence and sliding windows over the data stream, (b) an example of external utility table

### 5.2.1 Definitions and Problem Statement

Here, we use the same notations presented in Chapter 4. For more details about preliminaries, readers can refer to definitions in Chapter 4. Table 5.2 shows a list of notations used in this chapter.

**Definition 46** (**Sequence Data stream**) A transaction data stream of itemset-sequences (or *data stream* in short) $DS = \langle T_1, T_2, ..., T_M \rangle$ is an ordered list of transactions that arrive continuously in a time order. Each transaction $T_i \in DS$ (1≤i≤M) belongs to a sequence of transactions. A data stream can thus also be considered as a set of dynamically-changing sequences.

Table 5.2: Notations

| **Notation** | **Description** |
|---|---|
| $u(X, S_r^d)$ | Utility of item/itemset $X$ in transaction $T_d$ of $S_r$ |
| $TU(S_r^d)$ | Utility of transaction $T_d$ of sequence $S_r$ |
| $\alpha \preceq \beta$ | $\alpha$ is a subsequence of $\beta$, or $\alpha$ occurs in $\beta$ |
| $OccSet(\alpha, S_r)$ | Set of all the occurrences of $\alpha$ in sequence $S_r$ |
| $su(\alpha, S_r)$ | Utility of a sequence $\alpha$ in sequence $S_r$ |
| $\alpha \oplus I$ | Itemset-extended of sequence $\alpha$ and item $I$ |
| $\alpha \otimes I$ | Sequence-extended of sequence $\alpha$ and itemset $\{I\}$ |

**Definition 47** *(**Transaction-sensitive sliding window**) Given a user-specified window size $w$ and a data stream $DS = \langle T_1, T_2, ..., T_M \rangle$, 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 $SW_i = \langle T_i, T_{i+1}, ..., T_{i+w-1} \rangle$.*

**Definition 48 (Utility of a sequence $\alpha$ in a sequence $S_r$)** Let $\tilde{o} = \langle T_{e_1}, T_{e_2}, ..., T_{e_Z} \rangle$ be an occurrence of $\alpha = \langle X_1, X_2, ..., X_Z \rangle$ in the sequence $S_r \in DS$. The utility of $\alpha$ w.r.t. $\tilde{o}$ is defined as $su(\alpha, \tilde{o}) = \sum_{i=1}^{Z} su(X_i, S_r^{e_i})$. The utility of $\alpha$ in $S_r$ is defined as $su(\alpha, S_r) = \max\{su(\alpha, \tilde{o}) | \forall \tilde{o} \in OccSet(\alpha, S_r)\}$.

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

**Definition 49 (Utility of a sequence $\alpha$ in a dataset $D$)** The utility of a sequence $\alpha$ in a dataset $D$ is denoted and defined as $su(\alpha, D) = \sum_{S_r \in D} su(\alpha, S_r)$, where $D$ is clear in the context.

**Definition 50 (Top-k high utility sequential patterns in sliding window $SW_i$)** A sequence $\alpha$ is called a top-k high utility sequential pattern in $SW_i$, if there are less than $k$ sequences whose utility in the current sliding window $SW_i$ is no less than $su(\alpha, SW_i)$. The optimal minimum utility threshold is denoted and defined as $minUtil_{opt} = \min\{su(\beta, SW_i) | \beta \in THUSP_{SW_i}\}$, where $THUSP_{SW_i}$ is the set of top-k high utility sequential patterns over sliding window $SW_i$.

**Problem Statement.** Given a sequence data stream $DS$ and $k$, the problem of finding the complete set of top-k high utility sequential patterns in sliding window $SW_i$ over $DS$ is to discover all the HUSPs whose utility is no less than $minUtil_{opt}$ in the current sliding window $SW_i$.

### 5.2.2 Top-k High Utility Sequential Pattern Mining over Data streams

In this section, we propose an efficient algorithm called T-HUSP (Top-k High Utility Sequential Pattern mining over data streams) to discover top-k HUSPs without specifying minimum utility threshold. First, a basic approach called ***T-HUSP***$_{basic}$ is presented. Later, we present two novel strategies to initialize and raise the threshold with respect to the given $k$ in ***T-HUSP***$_{basic}$.

#### 5.2.2.1 *T-HUSP$_{basic}$* approach

The proposed baseline approach *T-HUSP$_{basic}$* takes $k$ as an input parameter and returns the $k$ sequences with the highest utilities over the current sliding window $SW_i$. It is an extended version of *HUSP-Stream*, the proposed method in Chapter 4 for mining high utility sequential patterns over data streams, and it applies the idea of *HUSP-Tree* to keep the information of potential top-k high utility sequential patterns over $SW_i$.

*T-HUSP$_{basic}$* engages a structure called *TKList* to keep the information of top-k high utility sequential patterns over the current sliding window and it is defined as follows:

**Definition 51** ***Top-K HUSP List (TKList)*** *is a fixed-size sorted list which maintains the top-k high utility sequential patterns and their* SeqUtilLists *(See Definition 28) dynamically. Each tuple in TKList has three elements:* $\langle \alpha, SeqUtilList(\alpha), util \rangle$*, where* $\alpha$ *is the pattern and* $SeqUtiList(\alpha)$ *is the current SeqUtilList of* $\alpha$ *and* $util$[4] *is the utility of pattern* $\alpha$ *in the current sliding window* $SW_i$*.*

*T-HUSP$_{basic}$* employs a variable called $minUtil$ which is the current minimum utility threshold and is set to zero at the beginning. This variable is used to prune unpromising candidates during the mining process.

The basic idea of T-HUSP$_{basic}$ is to modify the main procedure of the HUSP-Stream algorithm to transform it in a top-k pattern mining algorithm. This is done as follows.

[4]Note that the utility of $\alpha$ can be also calculated using SeqUtilList($\alpha$).

In the construction phase, to find the top-k high utility sequential patterns, once the first window forms, T-HUSP$_{basic}$ first sets $minUtil$ to 0. Then, similar to HUSP-Stream, T-HUSP$_{basic}$ constructs ItemUtilList and HUSP-Tree by applying the S-Step and I-Step procedures. As soon as a new node is added to HUSP-Tree, the pattern represented by the node, its $SeqUtilList$ and its utility are added as a new tuple to $TKList$. Once $k$ valid patterns are found, the $minUtil$ is raised to the utility of the pattern with the lowest *util* value in $TKList$. Raising the $minUtil$ value is used to prune the search space when searching for more patterns. Thereafter, whenever a new node is inserted to the tree, its pattern is added to $TKList$. Then, the patterns in $TKList$ whose utility is not more than or equal to $minUtil$ anymore are removed from $TKList$, and $minUtil$ is raised to the *util* value of the *kth* pattern in $TKList$. T-HUSP$_{basic}$ continues constructing HUSP-Tree and finding more patterns until no node can be generated, which means that it has found the top-k HUSPs in the current sliding window. In the update phase, since the utility of patterns in $TKList$ may change, the list will be emptied for the new window and $minUtil$ is set to zero to mine the correct set of top-k HUSPs in the new window. During HUSP-Tree update, when a node is updated/added, its pattern, its SeqUtilList and its utility are inserted as a new tuple to TKList. Once $k$ valid tuples are inserted to TKList, *minUtil* is raised to the *util* value of $kth$ tuple in TKList. T-HUSP$_{basic}$ continues updating the tree until no node is updated or added to the tree.

Since HUSP-Stream is correct and complete, it is clear that T-HUSP$_{basic}$ is correct

and will not miss any top-k HUSPs.

#### 5.2.2.2 Effective Strategies

Although T-HUSP$_{basic}$ correctly discovers the top-k high utility sequential patterns over a sliding window, it generates too many invalid sequence candidates since $minUtil$ starts from 0. This directly degrades the performance of the mining task. To address this problem, we propose two effective strategies, i.e., one for initializing the threshold in the construction phase and one for initializing the threshold in the update phase, to improve the performance.

**Strategy 1: (PES: Pre-Evaluation using 1-sequences and sequences)** The PES strategy is applied during the construction phase of the algorithm. This strategy inserts the utility of 1-sequences (items) and sequences in the current window to the TKList before the mining phase. After all transactions in the first sliding window are successfully inserted to $ItemUtilList$, PES calculates the utility of each item and each sequence. In this phase, we insert every sequence and every distinct item in $SW_1$ to TKList. Given $k = 4$, in Figure 5.3 in $SW_1$, the utility of item $a$ in $SW_1$ is calculated as follows: $u(a, SW_1) = u(a, S_1) + u(a, S_2) = 4 + 8 = 12$. The other utilities are $u(b, SW_1) = 18$, $u(c, SW_1) = 4$ and $u(d, SW_1) = 4$. Utility of each sequence can be easily calculated using ItemUtilLists. For example, after $SW_1$ is processed, $S_1$ itself, its $seqUtilList$ and its utility (e.g., 30) will be inserted into the $TKList$. Similarly, the other sequence in

$SW_1$ (e.g., $S_2$) is scanned and it is inserted into the $TKList$. With the two sequences and four items and their utilities in $SW_1$, the utilities in the TKList are $\{36, 30, 18, 12\}$, and then $minUtil = 12$ after applying PES strategy.

As seen from the example, the PES strategy effectively raises the minimum threshold to a reasonable level before the mining phase, and prevents from generating unpromising candidates.

**Strategy 2: (RTU: Raising threshold after update strategy)** When a new transaction $S_v^u$ arrives, if the current window $SW_i$ is full, the oldest transaction $S_c^d$ expires. Similar to HUSP-Stream, T-HUSP incrementally updates *ItemUtilLists* and *HUSP-Tree* to find the top-k HUSPs in $SW_{i+1}$. In addition to updating *ItemUtilLists* and *HUSP-Tree*, T-HUSP updates $TKList_i$ (i.e., TKList for sliding window $SW_i$) to initialize $TKList_{i+1}$. Our second strategy is to make use of the utility values of the top-k HUSPs in the previous sliding window to initialize the threshold when the window slides.

Let $TKList_i$ be the set of top-k HUSPs in the current sliding window $SW_i$, $TKList_i^-$ be the updated set of top-k HUSPs after a transaction $S_c^d$ is removed from $SW_i$. In order to build $TKList_i^-$ from $TKList_i$, for each pattern $\alpha \in TKList_i$, T-HUSP removes all the tuples from $SeqUtilList(\alpha)$ whose $SID$ is $S_c$. Accordingly, $util$ field for $\alpha$ is updated. Given $TKList_i^-$, we define the minimum top-k utility ($minTKUtil$) of a sliding window as follows.

**Definition 52** *Given sliding window* $SW_i = \{T_i, T_{i+1}, ..., T_{i+w-1}\}$ *and* $TKList_i^-$*, the*

*minimum top-k utility of a sliding window $SW_i$ is the minimum of utilities of the patterns in $TKList_i^-$ and is denoted as $minTKUtil_i$.*

**Lemma 12** *Let $util_k$ be the utility of the kth highest utility sequential pattern over sliding window $SW_{i+1}$, and $minTKUtil_i$ be the minimum top-k utility of $SW_i$. We have $util_k \geq minTKUtil_i$.*

**Proof**

Let $SW$ be the union of the last (w-1) transactions in $SW_i$. Then the next sliding window $SW_{i+1} = SW \cup S_u^v$ where $S_u^v$ is the new transaction in $SW_{i+1}$. Given sequence $\alpha \in TKList_{i+1}$, since $SW \subset SW_{i+1}$, then $su(\alpha, SW) \leq su(\alpha, SW_{i+1})$. According to Definition 52, $\forall \alpha \in TKList_i, minTKUtil_i \leq su(\alpha, SW)$ and the fact that there are $k$ sequences in $TKList_i$, there are at least $k$ sequences whose utility in $SW_{i+1}$ is at least $minTKUtil_i$.

□

According to this lemma, if the minimum utility threshold in $SW_{i+1}$ is set to $minTKUtil_i$, no top-k HUSPs will be missed.

#### 5.2.2.3 T-HUSP

The overview of *T-HUSP* is presented in Algorithm 11. Similar to HUSP-Stream, it includes three main phases: (1) *Initialization phase*, (2) *Update phase* and (3) *HUSP*

*mining phase*. The initialization phase applies when the input transaction belongs to the first sliding window (i.e., when $i \leq w$). In this phase, we first construct the ***ItemUtilLists*** for storing the utility information for every item in the input transaction $S_r^i$. When there are $w$ transactions in the first window (i.e., when $i = w$), we initialize $TKList$ and $minUtil$ by applying PES strategy based on 1-sequences in the $ItemUtilLists$ and the sequences in the first sliding window. Then, we construct ***HUSP-Tree*** for the first window. During the tree construction, whenever a new node is added to the tree, $TKList$ and $minUtil$ are updated as explained in section 5.2.2.1. If there are already $w$ transactions in the window when the new transaction $S_r^i$ arrives, $S_r^i$ is added to the window and the oldest transaction in the window is removed. This is done by the update phase of the algorithm (lines 9-12). We first update $TKList$ and $minUtil$ using RTU strategy. Then the ***ItemUtilLists*** is updated using the new transaction and the current $minUtil$. Given the updated $ItemUtilLists$, we apply PES strategy to update TKList and $minUtil$ based on the 1-sequences and sequences in the current window. After the update phase, if the user requests to find top-k HUSPs from the new window, T-HUSP returns all the patterns and their $util$ values in the current $TKList$ as top-k HUSPs (i.e., $THUSP_{SW_i}$).

**Algorithm 11** *T-HUSP*

**Input**: a new transaction $S_r^i$, window size $w$, $ItemUtilLists$, HUSP-Tree, $TKList$

**Output**: ItemUtilLists, HUSP-Tree, $TKList$, $THUSP_{SW_i}$

1: **if** $i \leq w$ (*when $S_r^i$ is a transaction in the first window)* **then**

2: $\forall$ item $\in S_r^i$, $put(r, i, u(item, S_r^i))$ to $ItemUtilLists(item)$

3: **if** i = w **then**

4: Initialize $TKList$ and $minUtil$ using **PES strategy** in the first window.

5: Construct HUSP-Tree using $ItemUtilLists$ and $minUtil$, Update $minUtil$ whenever a new node is added to the tree

6: **end if**

7: **else**

8: Update $TKList$ and $minUtil$ using **RTU strategy**

9: Update $ItemUtilLists$ using $S_r^i$, $w$ and $minUtil$

10: Update $TKList$ and $minUtil$ using **PES strategy** in the current window

11: Update HUSP-Tree, $TKList$ and $minUtil$ using $S_r^i$, $w$

12: **end if**

13: **if** the user requests to get top-k HUSPs for the current window **then**

14: $THUSP_{SW_i} \leftarrow$ all the patterns and their $util$ values stored in $TKList$

15: **end if**

16: Return $ItemUtilLists$, $HUSP-Tree$, $THUSP_{SW_i}$ if requested

## 5.3 Experimental Results

In this section, the proposed methods for finding top-k high utility patterns (i.e., itemset/sequence) over a data stream are evaluated. All the algorithms are implemented in Java. The experiments are conducted on an Intel(R) Core(TM) i7 2.80 GHz computer with 8 GB of RAM.

### 5.3.1 T-HUDS Performance Evaluation

#### 5.3.1.1 Datasets and Performance Measures

Four datasets are used in our experiments. The first one is *Connect-4* from the *UC-Irvine* Machine Learning Database Repository [11]. It is compiled from the *Connect-4* game state information. The total number of transactions is 67,557, while each transaction is with 43 items. It is a dense dataset with a lot of long itemsets. The second dataset is the IBM synthetic dataset *T10I4D100K* [27], where the numbers after $T$, $I$, and $D$ represent the average transaction size, average size of maximal potentially frequent patterns, and the number of transactions, respectively. The other two datasets are *BMS-POS* and *ChainStore*. *BMS-POS* contains several years worth of point-of-sale data from a large electronics retailer [27]. *ChainStore* is a dataset with over a million transactions, obtained from [51]. Table 5.3 shows details of the datasets. The *ChainStore* dataset already contains external utilities of the items and the frequency of each item in a transaction.

Table 5.3: Details of the datasets

| **Dataset** | **# Trans.** | **# Items** | **Avg.Length** | **batchSize** | **winSize** |
|---|---|---|---|---|---|
| Connect-4 | 67,557 | 132 | 43 | 10,000 | 3 |
| IBM | 100,000 | 870 | 10.1 | 10,000 | 5 |
| BMS-POS | 515,597 | 1,657 | 6.53 | 50,000 | 4 |
| ChainStore | 1,112,949 | 46,086 | 7.2 | 100,000 | 6 |

But the three other datasets do not provide external utility or the quantity of each item in each transaction. Hence, we randomly generated these numbers using a method described in [7] as follows. The external utility of each item is generated between 1 and 10 by using a *log-normal* distribution and the quantity of each item in a transaction is generated randomly between 1 and 10. $batchSize$ in Table 5.3 shows how many transactions are in a batch. It is set in the same way as in [7] so that each dataset has around 10 batches. The last column, $winSize$, shows the number of batches in a sliding window. We will later change the $winSize$ setting to show the effect of $winSize$ on performance measures.

We use the following performance measures in our experiments: (1) *number of generated candidates*: the total number of generated PTKHUIs at the end of phase $I$ among all the sliding windows, (2) *Threshold*, the threshold value obtained at the end of execution, (3) *Run Time (seconds)*: the total execution time of a method over all the sliding

windows, (4) *First Phase Time (seconds)*: the total run time of a method for phase $I$ (generating *PTKHUIs*) over all the windows, (5)*Second Phase Time (seconds)*: the total run time of a method for phase $II$ (finding *Top-HUI* set) over all the windows, (6) *Memory Usage(Mega Bytes)*: the memory consumption of a method, average over all the sliding windows.

#### 5.3.1.2 Methods in Comparison

To the best of our knowledge, there does not exist a top-k high utility itemset mining method over data streams. Hence, two modified approaches are implemented as comparison methods. The first one is the method proposed in [61] which discovers top-k high utility itemsets from a static dataset based on the UP-Growth method [57]. Since this method is not applicable to data streams, we run this method on each sliding window individually, and collect the aggregated values for the performance measures. This method is named $TKU$. $TKU$ has different versions, each employing a different set of threshold-raising strategies [61]. Here we use $TKU$ by employing all of the proposed strategies for raising the threshold. $TKU$ is able to set up its initial threshold to either 0 or the $k$th highest value of the lower bounds for the utility of certain 2-itemsets. Note that we need to scan the dataset twice to compute them before mining starts, which is not suitable for data stream mining. As will be observed, the obtained threshold using this method is better sometimes.

The second method that we compare our method with is the *HUPMS* algorithm [7], which discovers all the high utility itemsets over data streams given a user-input minimum utility threshold. To compare with the top-$k$ mining methods, we run the HUPMS algorithm with a minimum utility threshold being the threshold raised at the end of the Phase I execution of the basic version of *TKU* [61]. This is a fair choice of the threshold because a too low threshold would certainly make *HUPMS* very time-consuming, and a too high threshold would unfairly favor *HUPMS* in terms of run time. We denote this *HUPMS* method that uses a threshold from *TKU* as *HUPMS*$_T$ in our results.

In order to see the effect of using $PrefixUtil$ to prune the search space in comparison to other over-estimate utility measures, we compare our performance of $PrefixUtil$ with $TWU$ and the model proposed by [57] in terms of HUI mining with different user-specified minimum utility thresholds. In such a comparison, we do not use any threshold-raising strategies in *T-HUDS*, but let it return all the HUIs satisfying the input utility threshold.

To see how effective our threshold-setting/raising strategies is in the first phase of the method, we use two versions of our *T-HUDS* method to compare with $TKU$ and $HUPMS$. The first one, denoted as *T-*$HUDS_I$, uses only the 3 strategies that apply to the first phase of our method. The second one, denoted as *T-HUDS* is the full version of our method that uses all the 4 strategies, including the one in the second phase.

#### 5.3.1.3 Effectiveness of the Obtained Threshold

Figure 5.4 shows the threshold values obtained from different methods on the four datasets for differen $k$ values, where $k$ is the number of output HUIs specifed by the user (i.e., $k$ in top-$k$). Since *HUPMS* does not raise the threshold during the mining process, we just compare the results of *TKU* with the proposed methods. The results of *TKU* are the average threshold values through all of the windows. This figure shows that *T-HUDS*$_I$ and *T-HUDS* have similar performance and their final thresholds are higher than *TKU* especially on the large datasets. Since none of these three methods miss any top-$k$ HUIs, the higher the final threshold, the better the method. Although *TKU* could get better or similar results on some experiments, both T-HUDS$_I$ and *T-HUDS* outperform *TKU* in general. Note that, as it is presented later, not only *TKU* is time-consuming to find top-k HUIs, but also some of its strategies for raising the threshold requires a large amount of memory. Between T-HUDS$_I$ and *T-HUDS*, *T-HUDS* is bit better, but not significantly. This means that the 3 strategies used in Phase I of *T-HUDS* are very effective, raising the threshold close to the exact utility of the $k$th highest utility itemset. Recall that the threshold value at the end of Phase II is the exact utility of the $k$th itemset in the top-$k$ list.

The figure also shows that the threshold value decreases when $k$ increases. It is because the larger the $k$ value is, the lower the threshold value needs to be to return more

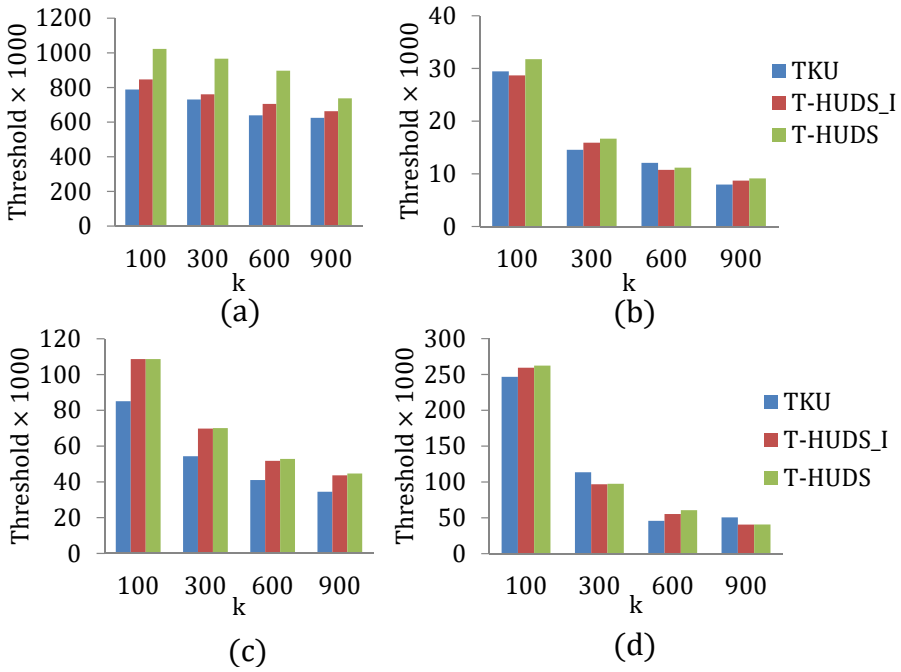



Figure 5.4: Reached threshold on (a) Connect-4, (b) IBM, (c) BMS-POS,(d) ChainStore Datasets

itemsets.

#### 5.3.1.4 Number of generated candidates

In addition to the obtained threshold, the number of generated candidates (i.e., *PTKHUI*s) at the end of the first phase is another metric to assess the effectiveness of HUI mining methods. Table 5.4 presents the numbers of generated candidates on different datasets from different methods for different $k$ values. The numbers show that *T-HUDS* significantly outperforms *TKU*. Although *TKU* could achieve better threshold on some ex-

periments in the previous section, since for each window, it starts from a small threshold value (initial value), it generates much more candidates in comparison to *T-HUDS*. The results for *T-HUDS*$_I$ are not shown here because they are the same as the ones for *T-HUDS*. The table also shows that *HUPMS*$_T$ method generates fewer candidates in smaller datasets than *T-HUDS*, but much more candidates on larger datasets. The number of candidates generated by *HUPMS*$_T$ is determined by the minimum utility threshold given to the method, which is the threshold reached at the end of Phase I of $TKU$. Even though the final Phase I threshold of *T-HUDS* is higher than that of $TKU$, the number of candidates generated by *HUPMS*$_T$ can still be smaller than that from *T-HUDS*. This is because the initial threshold of *T-HUDS* can be lower than the final Phase I threshold of $TKU$. But on very large dataset (such as *ChainStore*), the initial threshold of *T-HUDS* can be higher than or close to the final Phase I threshold of $TKU$ since the number of candidates generated by *HUPMS*$_T$ is much higher than the one by *T-HUDS*.

Table 5.4: Number of candidates generated in phase I

| **Dataset** | **k** | **TKU** | **T-HUDS** | **HUPMS$_T$** |
|---|---|---|---|---|
| Connec-4 | 100 | 2280595 | 1189624 | 657934 |
| | 300 | 2587463 | 1258241 | 717934 |
| | 600 | 2865490 | 1315869 | 857934 |
| | 900 | 3069445 | 1472473 | 1007934 |
| IBM | 100 | 103485 | 69959 | 22038 |
| | 300 | 135998 | 84898 | 26668 |
| | 600 | 198671 | 94850 | 54969 |
| | 900 | 276668 | 100875 | 217874 |
| BMS-POS | 100 | 45054 | 35697 | 31407 |
| | 300 | 59357 | 37251 | 35682 |
| | 600 | 119479 | 47215 | 42112 |
| | 900 | 177725 | 50189 | 51463 |
| ChainStore | 100 | 40419 | 19751 | 101435 |
| | 300 | 140236 | 32213 | 152451 |
| | 600 | 258318 | 102385 | 211627 |
| | 900 | 371408 | 227826 | 282074 |

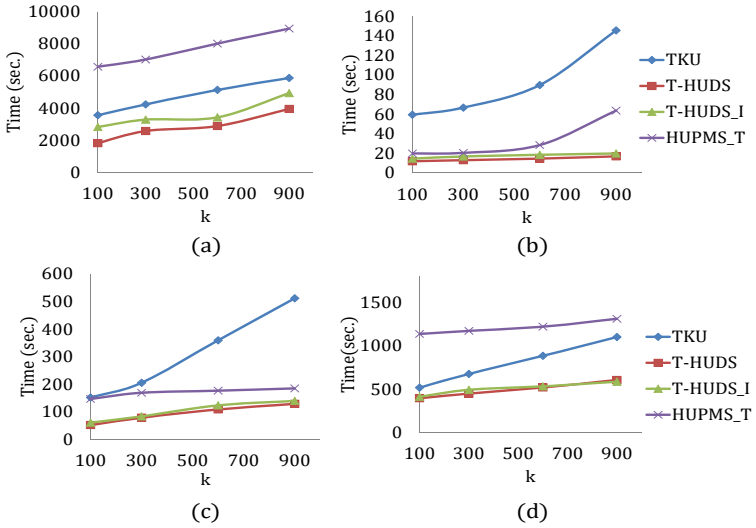



Figure 5.5: Run time on (a) Connect-4, (b) IBM, (c) BMS-POS, (d) ChainStore Datasets

#### 5.3.1.5 Efficiency of T-HUDS: Run Time

Figure 5.5 shows the total run time of each method, including the run time for both Phase I and Phase II. On the IBM and BMS-POS datasets, the execution time of *TKU* is much worse than others, and *HUPMS*$_T$ is a bit worse than *T-HUDS*$_I$ and *T-HUDS*. On *Connect-4* and $ChainStore$, *T-HUDS*$_I$ and *T-HUDS* are significantly faster than both *HUPMS*$_T$ and *TKU*. On the largest dataset ($ChainStore$) and the most densest dataset (*Connect-4*), *HUPMS*$_T$ is the worst, even much worse than *TKU*. The run time for *T-HUDS*$_I$ and *T-HUDS* are very similar, although *T-HUDS* is slightly faster due to its raising $min\_util$ dynamically for pruning out unpromising itemsets in Phase II. Also, it can be observed that the run time of the proposed methods are not affected significantly by the $k$ values, and it increase slightly and slowly when $k$ increases. It is also worth mentioning that

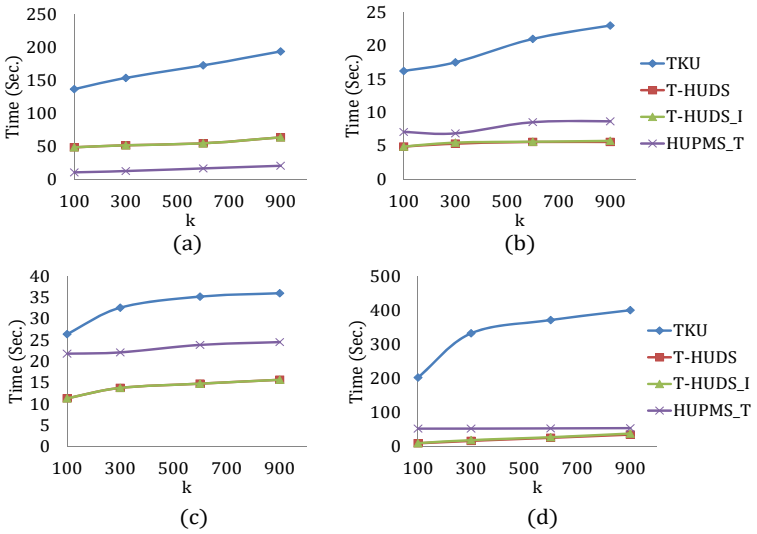



Figure 5.6: Run Time for Phase I: (a) Connect-4, (b) IBM, (c) BMS-POS, (d) ChainStore

*T-HUDS* significantly outperforms other methods in both large and dense datasets.

To see how each method works in different phases, Figures 5.6 and 5.7 present the execution time for Phases I and II, respectively. It can be observed that in both phases the proposed methods outperform $TKU$ and $HUMP_T$. In Phase I, the two proposed methods have the same performance. But in the second phase, *T-HUDS* is more efficient. This is because it dynamically increases the $min\_util$ threshold in Phase II and consequently the number of candidates compared with the running top-$k$ list is fewer than that in *T-HUDS*$_I$.

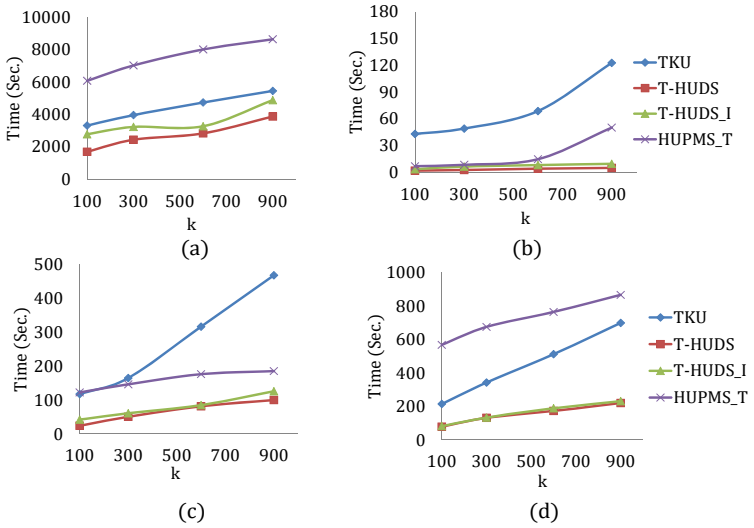



Figure 5.7: Run Time for Phase II: (a) Connect-4, (b) IBM, (c) BMS-POS, (d) ChainStore

#### 5.3.1.6 Efficiency of T-HUDS: Memory Usage

In this section, we report the memory usage taken by the trees, their header tables, auxiliary data structures and one window of transactions. Table 5.5 reports the memory consumption on the four datasets. *TKU* consumes the most memory, even though the structure of its tree node is the smallest among the three methods. This is due to the large amount of memory that it needs to initialize the threshold and also the larger number of conditional *UP-trees* recursively generated during the mining process. It is caused by the fact that *TKU* starts by a low threshold value at the beginning of each window and its strategies for raising the threshold are not very effective in the data stream environment. Also, as *TKU* is not designed for mining over data streams, it cannot utilize the information from the past windows to raise the threshold. In all cases, the proposed method

Table 5.5: Memory comparison (MB), k=600

| **Dataset** | **TKU** | **T-HUDS** | **HUPMS$_T$** |
|---|---|---|---|
| Connect-4 | 368 | 31.27 | 72.7 |
| IBM | 287 | 5.18 | 7.58 |
| BMS-POS | 301 | 15.2 | 33.5 |
| ChainStore | 4287 | 102 | 305 |

Table 5.6: Methods with different strategies

| Method | $maxUtilList$ | $MIUList$ | $minTopKUtil$ |
|---|---|---|---|
| $T$-$HUDS_1$ | $\times$ | ✓ | ✓ |
| $T$-$HUDS_2$ | ✓ | $\times$ | ✓ |
| $T$-$HUDS_3$ | ✓ | ✓ | $\times$ |

*T-HUDS* consumes less memory than both *TKU* and *HUPMS$_T$*. Note that the node structure in *HUPMS$_T$* is also smaller than that in *T-HUDS*. But again the effective pruning strategies used in *T-HUDS* lead to generation of a smaller stack of trees in the recursive execution of the tree mining algorithm.

#### 5.3.1.7 Effectiveness of the Individual Strategies

In this section we investigate the impact of each of the three threshold-setting strategies used in Phase $I$ of our method. Table 5.6 describes three different versions of the proposed method. The first method does not use $maxUtillist$ values to set the threshold but uses $MIUList$ and the minimum top-$k$ utility from the last window (i.e., $minTopKUtil$). ***T-HUDS***$_2$ increases the threshold by means of $maxUtilList$ and $minTopKUtil$, but not by $MIUList$. ***T-HUDS***$_3$ applies the first and second strategies only.

Table 5.7 and Figure 5.8 show the number of generated candidates and run time of these three methods on the *IBM*, *BMS-POS* and *ChianStore* datasets, respectively. In general,***T-HUDS***$_3$ (the method without the third strategy) is the worst among the three methods. It means that third strategy ( i.e., using the last window's $minTopKUtil$) is the most effective strategy. ***T-HUDS***$_2$ has better performance than ***T-HUDS***$_1$, meaning that the first strategy (i.e., the use of $maxUtilList$) works better than the second one (i.e., using $MIUList$). Since in our implementation of $TKU$, $MIUList$ is used as one of the threshold-raising strategies, this results explain in part why ***T-HUDS*** outperforms $TKU$.

Table 5.7: Number of candidates at the end of first phase for different versions of T-HUDS:

| **Dataset** | **k** | **T-HUDS**$_1$ | **T-HUDS**$_2$ | **T-HUDS**$_3$ |
|---|---|---|---|---|
| IBM | 100 | 89804 | 77968 | 224262 |
| | 300 | 110487 | 87108 | 365500 |
| | 600 | 113352 | 95988 | 394716 |
| | 900 | 116889 | 107163 | 408102 |
| BMS-POS | 100 | 44117 | 39075 | 111172 |
| | 300 | 51201 | 49870 | 104019 |
| | 600 | 63544 | 61962 | 120607 |
| | 900 | 80827 | 80469 | 149012 |
| ChainStore | 100 | 24409 | 21620 | 61511 |
| | 300 | 44276 | 43125 | 89950 |
| | 600 | 137794 | 134363 | 261534 |
| | 900 | 366902 | 365277 | 676419 |

#### 5.3.1.8 Effectiveness of PrefixUtil

Below we evaluate the use of $PrefixUtil$ (in comparison to the use of other over-estimate utility models) for pruning the search space during the recursive tree mining

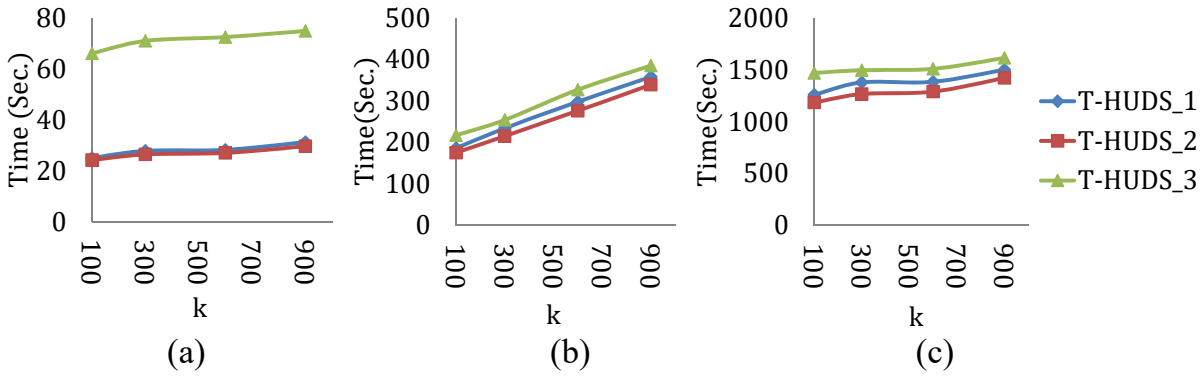



Figure 5.8: Run time for different versions of T-HUDS: (a) IBM, (b) BMS-POS, (c) ChainStore datasets

process. For such a purpose, we run *T-HUDS* in the problem setting of *HUPMS*. That is, we do not use any of the threshold raising strategies in *T-HUDS* and use it as a method for finding all the high utility itemsets that satisfy an input $min\_util$ threshold. *TKU* mines top-$k$ HUIs based on the *UPGrowth* method [57]. Hence *TKU* without threshold raising strategies is *UPGrowth* method that finds all high utility itemsets given an input minimum utility threshold. Since *UPGrowth* is not applicable to data streams directly and we would like to evaluate the performance of its over-estimate utility model not the method, we use its proposed over-estimate utility model as the over-estimated utility in *T-HUDS* to replace $prefixUtil$. This method is called *T-HUDS*$_U$. This is to make *T-HUDS* and *T-HUDS*$_U$ the same as *HUPMS* except that *T-HUDS* uses $PrefixUtil$ while *T-HUDS*$_U$ uses the proposed over-estimate model in [57] and *HUPMS* uses $TWU$ to prune the search space. Hence, a comparison between these methods will illustrate the

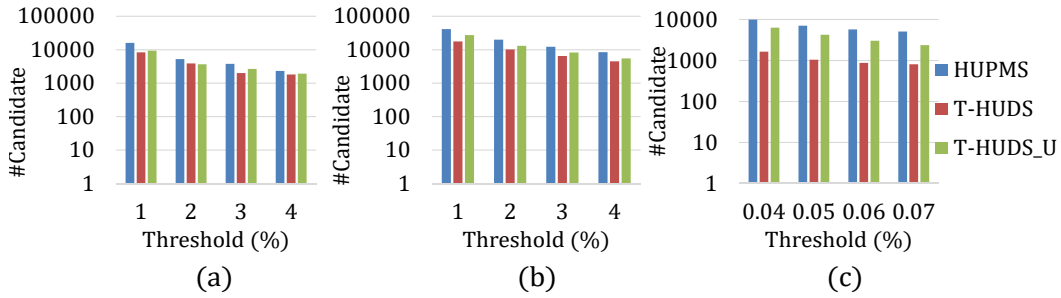



Figure 5.9: Impact of $PrefixUtil$ on the number of generated candidates on (a) IBM, (b) BMS-POS, (C) ChainStore datasets

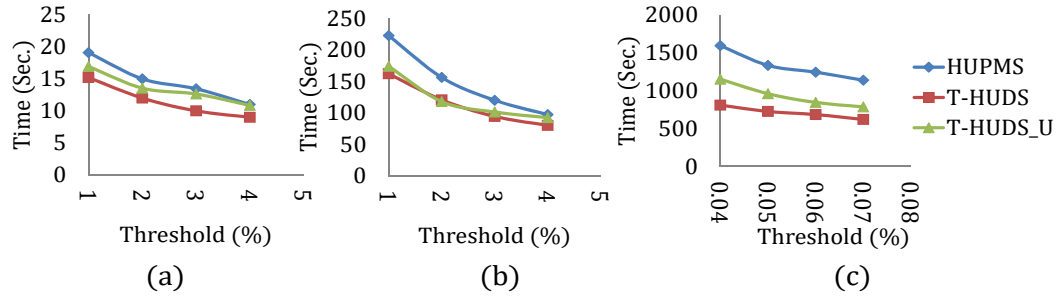



Figure 5.10: Impact of $PrefixUtil$ on run time on (a) IBM, (b) BMS-POS, (c) ChainStore datasets

impact of $PrefixUtil$.

Figures 5.9 and 5.10 present the number of generated candidates in Phase one of the three methods and their total run time with respect to different minimum utility threshold values. The minimum utility threshold is given by the percentage of total transaction utility values of the database. The reason why we chose a different range of the threshold value for the *ChainStore* dataset is that it is a sparse dataset and the number of potential

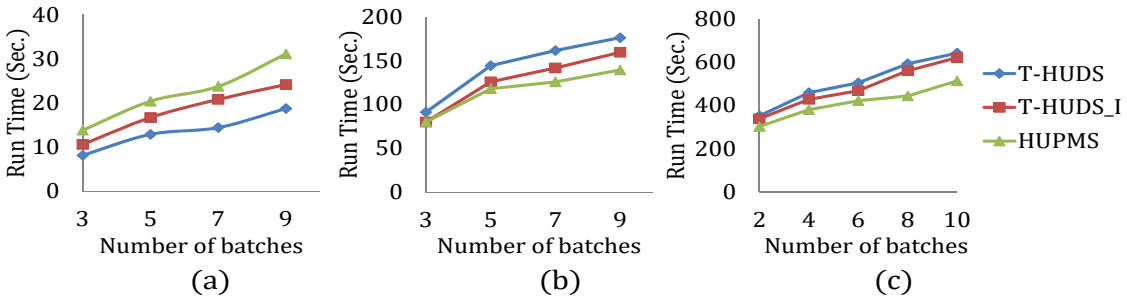



Figure 5.11: Effect of the window size on the run time: (a) IBM, (b) BMS-POS, (c) ChainStore datasets

candidates for large threshold values is too low.

These figures show that our algorithm outperforms *HUPMS* and *T-HUDS*$_U$ methods in terms of both the number of generated candidates and the run time. Moreover, these figures also demonstrate that the number of candidates and runtime differences increase in general when the minimum utility threshold decreases. As discussed earlier, the reason for $PrefixUtil$ to be more effective in pruning the search space is that it is a closer overestimate of the true utility.

#### 5.3.1.9 T-HUDS performance with different window sizes

Because *T-HUDS* dynamically updates the tree and the set of top-$k$ patterns once the window slides, its performance may vary depending on the window size parameter, $winSize$. In general, for a sliding window-based data stream mining algorithm, $winSize$ is an important factor on efficiency. Therefore, in order to determine the effect of changes in

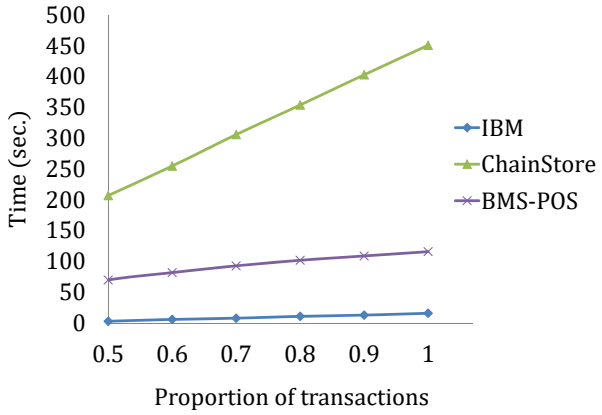



Figure 5.12: Scalability of T-HUDS on different datasets (k=500).

$winSize$ on the run time of *T-HUDS*, we analyze its performance by changing the value of this parameter. Below we present the results on the *IBM*, *BMS-POS* and *ChainStore* datasets, keeping the $k$ value fixed, but changing the number of batches in the sliding window. We compare the performance of our algorithms with the *HUPMS*$_T$ in this experiment. Figure 5.11 shows the results for k = 300. The y-axes in the graphs represent the overall run time (including tree construction time, update time, and mining time) for all the windows. The x-axes represent the window size in the number of batches. Each graph shows the trend in execution time with the variation of the window size on a dataset. On all the $winSize$ values, the proposed method is much faster than *HUPMS*$_T$, and its run time increases slowly as the window size increases.

#### 5.3.1.10 Scalability

To evaluate the scalability of the proposed algorithms, we generate a number of subsets of the $IBM$, $BMS-POS$ and $ChainStore$ datasets The size of a subset ranges from 50% to 100% transactions of the dataset it is generated from. Figure 5.12 illustrates how the run time of the algorithms for producing top-600 HUIs varies with different dataset sizes. We observe that the run time increases (almost) linearly when the number of transactions increases. This indicates that *T-HUDS* scales well with the size of dataset.

Table 5.8: Parameters of IBM data generator

| | |
|---|---|
| D | Number of sequences |
| C | Average number of transactions in a sequence |
| T | Average number of items in a transaction |
| S | Average number of itemsets in a potential maximal sequential pattern |
| I | Average number of items in an itemset of a potential maximal sequential pattern |
| N | Number of distinct items |

### 5.3.2 T-HUSP Performance Evaluation

In this section, we evaluate the performance of T-HUSP on a variety of datasets.

#### 5.3.2.1 Datasets and Performance Measures

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 [2]. The definition of parameters used by the IBM data generator are shown in the Table 5.8.

*Chainstore* is a real-life dataset acquired from [51], which already contains internal and external utilities. In order to use this dataset as a sequential dataset, we grouped transactions in different sizes such that each group represents a sequence of transactions. Another real dataset *BMS* is obtained from SPMF [25] which contains 3340 distinct items and consists of 77,512 sequences of clickstream data from an e-retailer. We follow a previous study [4] 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 5.9 shows characteristics of the datasets and parameter settings in the experiments. The *Window Size* column of Table 5.9 shows the default window size for each dataset.

We use the following measures to evaluate the performance of the algorithms:

- *Number of potential high utility sequential patterns ($\#PHUSP$)*: the total number of potential high utility sequential patterns produced by the algorithm in all sliding

Table 5.9: Details of parameter setting

| **Dataset** | **#Seq** | **#Trans** | **# Items** | **Window Size (w)** |
|---|---|---|---|---|
| DS1 | 10K | 100K | 1000 | 50K |
| BMS | 77K | 120K | 3340 | 60K |
| DS2 | 100K | 800K | 1000 | 400K |
| ChainStore | 400K | 1000K | 46,086 | 500K |

windows.

- *Run Time (sec.)*: the total execution time of the algorithms.
- *Memory Usage (MB)*: the average memory consumption per window.

#### 5.3.2.2 Methods in Comparison

Since there is no algorithm can solve the problem of top-k high utility sequential pattern mining over data streams, and it is not easy to upgrade the existing methods such as [67] either, we thus compare T-HUSP with our proposed baseline approach as described in subsection 5.2.2.1. Our preliminary experiments show that, T-HUSP$_{basic}$ cannot return results in a reasonable time for the large datasets. Hence, we implement another version of T-HUSP called T-HUSP$_{PES}$ which applies $PES$ strategy to initialize the threshold when a window forms or slides. We also use the threshold-based approach ( i.e., HUSP-

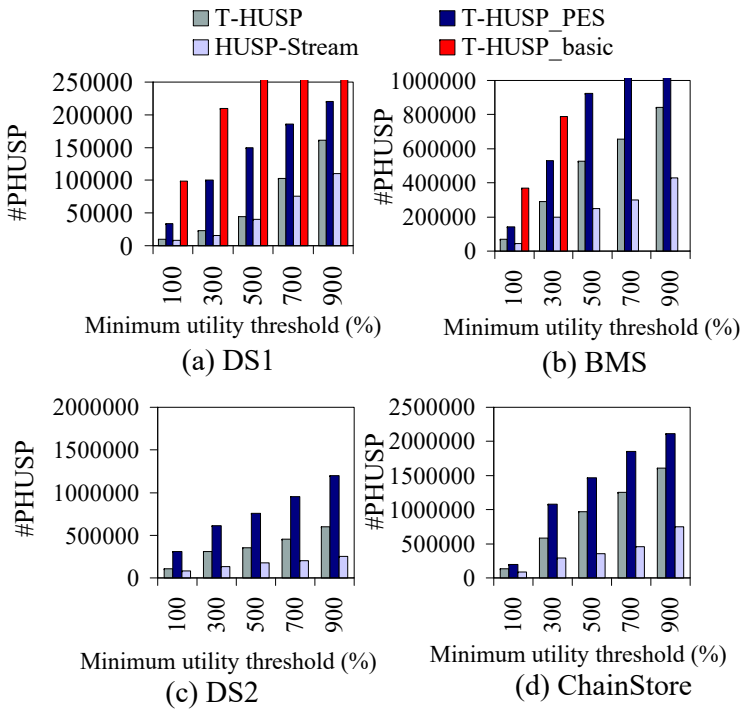



Figure 5.13: Number of potential HUSPs on (a)DS1, (b) BMS, (c) DS2, (d) ChainStore Datasets

Stream) proposed in Chapter 4 as another baseline approach. To make the top-k approaches and threshold based approaches comparable, we run top-k approaches first. After getting the utility of the k-th pattern, that is the optimal minimum utility in Definition 50, we use this value as the minimum utility threshold for running the threshold-based method.

#### 5.3.2.3 Number of generated candidates

In this section, we evaluate the algorithms in terms of the number of potential high utility sequential patterns (PHUSPs) produced by the algorithms. Figure 5.13 shows the results under different $k$ values. As shown in Figure 5.13, T-HUSP produces much fewer PHUSPs than T-HUSP$_{PES}$ and T-HUSP$_{basic}$. For example, on DS1, when the $K = 300$, the number of PHUSPs generated by T-HUSP$_{basic}$ is 8 times more than that generated by T-HUSP. On the larger datasets, i.e., DS2 and ChainStore, the number of PHUSPs grows quickly when $K$ increases and T-HUSP$_{basic}$ could not return results in a reasonable time. The main reason why T-HUSP produces much fewer candidates is that T-HUSP raises the threshold efficiently. Hence it avoids generating a large number of PHUSPs during the mining process.

#### 5.3.2.4 Efficiency of T-HUSP: Run Time

We compare T-HUSP with T-HUSP$_{basic}$, T-HUSP$_{PES}$ and HUSP-Stream on DS1, BMS, DS2 and ChainStore datasets. The run time of mining top-k high utility sequential patterns by these methods are presented in Figure 5.14. The results show that T-HUSP is generally more than 10 times faster than T-HUSP$_{basic}$. For DS2 and ChainStore, T-HUSP$_{basic}$ cannot finish the mining with a very small k (with k = 10 and 20) in 24+ hours. In addition, T-HUSP outperforms T-HUSP$_{PES}$ significantly. Besides, the gap

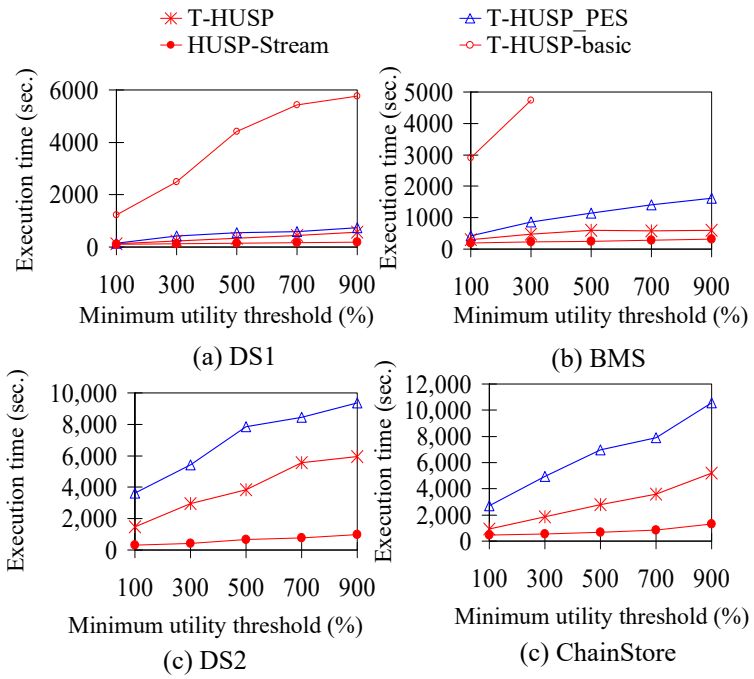



Figure 5.14: Run time on (a) DS1, (b) BMS, (c) DS2, (d) ChainStore Datasets

between T-HUSP and T-HUSP$_{PES}$ increases with the increase of k. The results indicate that the proposed strategies, including PES and PTU, are effective for top-k pattern mining.

#### 5.3.2.5 Efficiency of T-HUSP: Memory Usage

The memory consumption of the algorithms on the DS1, BMS, DS2 and ChainStore datasets is shown in Figure 5.15. It can be seen that T-HUSP uses less memory than T-HUSP$_{basic}$ and T-HUSP$_{PES}$ on the different datasets. The reason T-HUSP generates less numbers of candidates is that it applies both proposed strategies, thus increases the

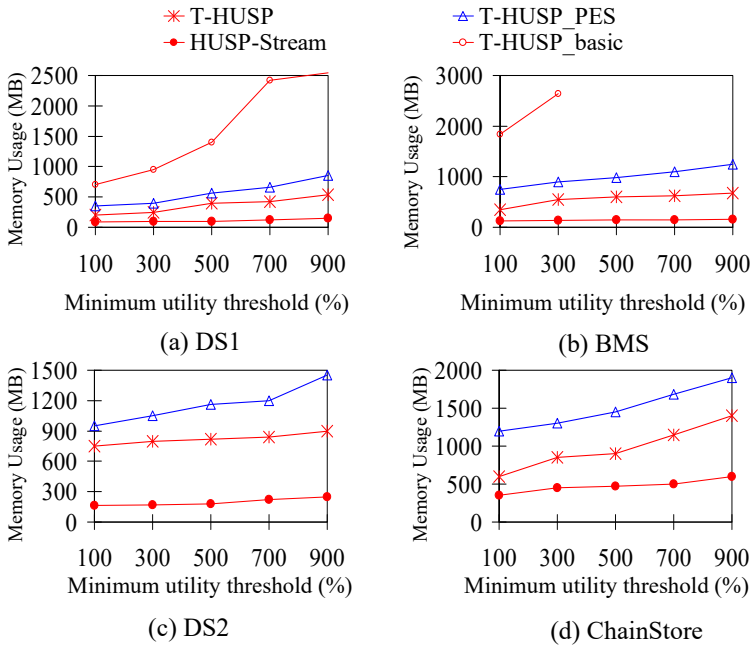



Figure 5.15: Memory usage on(a)DS1, (b) BMS, (c) DS2, (d) ChainStore Datasets

threshold quicker than the other top-k mining approaches. Since all the methods use similar pruning strategies in the mining phase, the main factor in memory consumption is the threshold used by each of them. Since HUSP-Stream uses the optimal threshold, it can prune the search space efficiently, thus its memory usage is less than the other methods.

## 5.4 Summary

In this chapter, we proposed two methods for finding top-k high utility patterns over sliding windows of a data stream. The first proposed method, called *T-HUDS*, finds top-

k *high utility itemsets* over data streams. Our contributions are summarized as follows.

- **Four efficient strategies to raise the threshold:** a major challenge in top-k HUI mining is that the number of itemsets is exponential and it is infeasible to compute the utilities of all the itemsets and identify the top-k ones. A minimum utility threshold is thus needed in the mining process to prune the search space. We proposed four strategies for initializing and dynamically adjusting the minimum utility threshold during the top-$k$ HUI mining process. We proved that using these strategies will not miss any top-$k$ HUIs.

- **Efficient search space pruning strategy:** we proposed an over-estimate of the itemset utility, which is closer to the true utility than TWU. We prove that this estimate (i.e., *prefix utility*) has a special type of *downward closure property*, which allows it to be used in the pattern growth method to effectively prune the search space. Using a closer over-estimate results in fewer candidates being generated in the first phase of the method.

- **Compact data structure:** we proposed a compact data structure (called *HUDS-Tree*) to store the information about the transactions in a sliding window. The tree is used to compute the prefix utility and to initialize and adjust the minimum utility threshold. The main differences of HUDS-Tree in comparison to the existing data structures are as follows. (1) its node stores the utility information of the itemset

for each batch in a sliding window to facilitate the update process. (2) Instead of storing TWU or other over-estimate utility values in a node, a node in a HUDS-Tree stores the *PrefixUtil* of the represented itemset for each batch, which is a closer estimate of the true utility of the itemset than TWU.

- **Efficient top-k high utility itemset mining algrithm:** using HUDS-Tree and the proposed strategies and prefix utility model, we designed a method for mining top-$k$ high utility itemsets from data streams. To the best of our knowledge, existing methods for mining HUIs over data streams do not address the issue of mining top-$k$ HUIs, and previous top-$k$ HUI mining methods do not work on data streams.

- **Extensive experiments:** we conducted an extensive experimental evaluation of the proposed method on both real and synthetic datasets, which shows that our proposed method is faster and less memory consuming than the state-of-the-art methods.

Inspired by T-HUDS, we proposed our second method, called *T-HUSP*, for discovering of top-k *high utility sequential patterns* over data streams. Our contributions are summarized as follows.

- **Three efficient strategies to raise the threshold:** we proposed three strategies for initializing and dynamically adjusting the minimum utility threshold during the top-k HUSP mining process. We proved that using these strategies will not

miss any top-k HUSPs.

- **Efficient top-k high utility sequential pattern mining algrithm:** we proposed a sliding window-based method for mining top-k high utility sequential patterns over data streams. To the best of our knowledge, existing methods for mining HUSPs over data streams do not address the issue of mining top-k HUSPs, and previous top-k HUSP mining methods do not work on data streams.

- **Extensive experiments:** we conducted extensive experiments on both real and synthetic datasets to evaluate the performance of the proposed algorithm. Experimental results show that T-HUSP serves as an efficient solution for the problem of top-k HUSP mining over data streams.

# 6 Mining Meaningful Patterns in Real Life Applications

In this chapter, we present two applications of high utility sequential pattern mining in solving real world problems. We first conduct an analysis on a real web clickstream dataset, called (*Globe*), obtained from a Canadian news web portal to extract *attractive reading behavior* (to be defined later). Then, we apply one of our proposed methods (i.e., MAHUSP[5]) to a publicly available time course microarray dataset, called *GSE6377* [47], to identify *disease-related gene expression sequences*.

## 6.1 A Utility-based Users' Reading Behavior Mining

News recommendation plays an important role in helping users find interesting pieces of information. A major approach to news recommendation focuses on modeling web users' reading behavior (reading behaviour in short). This approach discovers users' reading behaviour from web clickstreams using various data mining techniques such as *frequent pattern mining*. Nonetheless, there are some common deficiencies in the fre-

[5]For consistency, we use *MAHUSP* here as the main method to discover patterns. However, the other proposed methods are applicable for both applications

Table 6.1: 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* | $HUSP_1$ | *Retiree, 60, wonders how long her money will last*<br>*Which is better, a RRIF or an annuity? You may be surprised* | 1474 | 152 |
| | $HUSP_2$ | *Robin Williams warp − speed improvisation was almost …*<br>*CBC lays off veteran sportscasters amid budget cuts* | 1471 | 121 |
| | $HUSP_3$ | *Israel prepares to 'significantly' expand campaign as UN chief…*<br>*MH17: Disaster ratchets up Russia − Ukraine tensions* | 1212 | 116 |
| | $HUSP_4$ | *Massive explosive decompression' downed MH17: Kiev*<br>*Canada should learn from Ireland's housing crash* | 994 | 86 |
| *SPADE* | $FSP_1$ | *CBC lays off veteran sportscasters amid budget cuts*<br>*Celine Dion takes indefinite break to focus on health, family* | 576 | 286 |
| | $FSP_2$ | *La Prairie, Quebec mayor dies from wasp stings*<br>*Duffy billed taxpayers for attending funerals, RCMP allege* | 380 | 254 |
| | $FSP_3$ | *Supreme Court sides with Ottawa in multibillion − dollar EI case*<br>*MH17: Disaster ratchets up Russia − Ukraine tensions* | 536 | 247 |
| | $FSP_4$ | *Controversial First Nation chiefs salary raises concern*<br>*Harper sticks to hard line on Hamas; U.S. condemns Israel's deadly* | 836 | 220 |

quent pattern based approaches to web users' reading behavior mining. First, they discover users' reading behavior patterns based on the *frequency* of the news being viewed by users, which may not accurately capture users' interests. Second, the news domain is a dynamic environment. When users visit a news website, they are usually looking for important and up-to-the-minute information. However, the frequency based pattern mining approaches do not consider the *importance* (e.g., recency) of news articles.

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

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

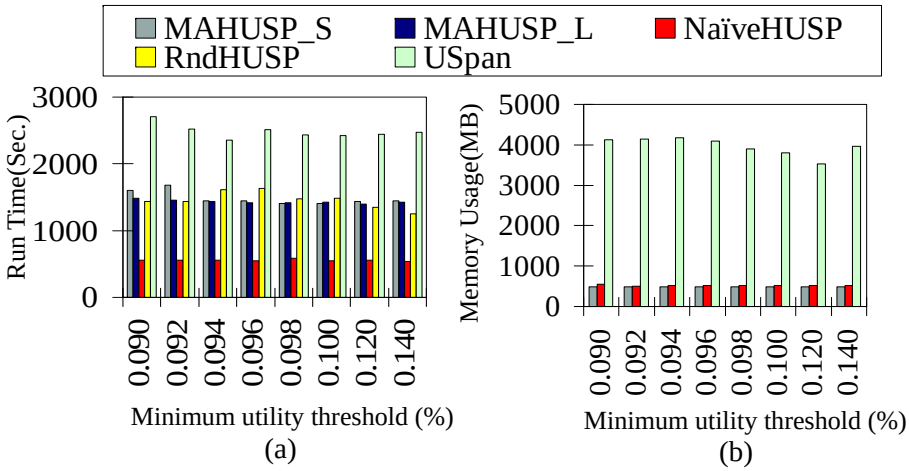



Figure 6.1: (a) Run time, (b) Memory Usage on the *Globe* dataset

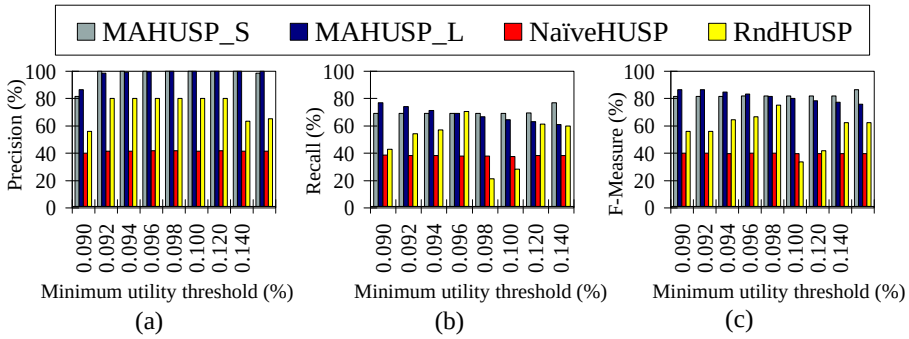



Figure 6.2: (a) Precision, (b) Recall and (c) F-Measure performance on the *Globe* dataset

in a visit.

### 6.1.1 Definitions

The most important entities involve in modeling reading behavior are *news articles* and *users*. Below, we first present definitions and propose a *news attractive model* as the utility model to take both *news importance* and *user's interest* into account. Then, we

| Session $ID$ | $\langle newsID, \{Shared, Liked, Time\ Spent\}\rangle$ | $\langle newsID, INS\rangle$ |
|---|---|---|
| $S_1$ | $\langle nw_1, \{1, 1, 14\}\rangle\langle nw_3, \{1, 0, 3\}\rangle\langle nw_5, \{1, 1, 22\}\rangle\langle nw_6, \{0, 1, 7\}\rangle$ | $\langle nw_1, 2.6\rangle\langle nw_3, 1.1\rangle\langle nw_5, 3\rangle\langle nw_6, 1.3\rangle$ |
| $S_2$ | $\langle nw_4, \{0, 0, 4\}\rangle\langle nw_5, \{1, 0, 15\}\rangle\langle nw_6, \{1, 1, 18\}\rangle$ | $\langle nw_4, 0.18\rangle\langle nw_5, 1.6\rangle\langle nw_6, 2.8\rangle$ |
| $S_3$ | $\langle nw_2, \{1, 1, 14\}\rangle\langle nw_4, \{0, 0, 1\}\rangle\langle nw_5, \{1, 0, 19\}\rangle\langle nw_6, \{0, 1, 3\}\rangle$ | $\langle nw_2, 2.6\rangle\langle nw_4, 0.04\rangle\langle nw_5, 1.86\rangle\langle nw_6, 1.1\rangle$ |
| $S_4$ | $\langle nw_1, \{1, 0, 4\}\rangle\langle nw_3, \{1, 0, 8\}\rangle\langle nw_5, \{0, 0, 13\}\rangle$ | $\langle nw_1, 1.1\rangle\langle nw_3, 1.3\rangle\langle nw_5, 0.59\rangle$ |
| $S_5$ | $\langle nw_4, \{1, 0, 9\}\rangle\langle nw_5, \{1, 0, 2\}\rangle$ | $\langle nw_4, 1.4\rangle\langle nw_5, 1.09\rangle$ |

Figure 6.3: An example of a web clickstream dataset

| *News ID* | Popularity | *Recency* | $NI(nw_i, \langle Popularity, Recency\rangle)$ |
|---|---|---|---|
| $nw_1$ | 3 | 1 | 3 |
| $nw_2$ | 5 | 4 | 20 |
| $nw_3$ | 1 | 3 | 3 |
| $nw_4$ | 7 | 2 | 14 |
| $nw_5$ | 2 | 5 | 10 |
| $nw_6$ | 4 | 3 | 12 |

Figure 6.4: The importance of news articles based on popularity and recency

define and extract *attractive reading behaviour*.

Let $NW = \{nw_1, nw_2, ..., nw_n\}$ be a set of distinct news articles. A *user session* $S$ (or sessions in short) is defined as an ordered list of visited articles $\langle nw_1, nw_2, ..., nw_z\rangle$ within a session. In this work, we define a *web clickstream* dataset as a set of sessions $\{S_1, S_2, ...., S_K\}$, where each session $S_r$ has a unique session identifier and consists of an ordered list of news articles and a set of variables associated to each visited news article. Figure 6.3 shows an example of a clickstream dataset $D$ with 5 sessions $\{S_1, S_2, S_3, S_4, S_5\}$. In this example, three variables *Shared*, *Liked* and *Time Spent* have been captured per visited article. For example, item $\langle nw_1, \{1, 1, 14\}\rangle$ in $S_1$ means that the user visited news $nw_1$ and pressed like button and also shared $nw_1$ in social media.

The user also spent 14 minutes to read $nw_1$.

**Definition 53** The **importance of news** $nw$ is a score which is calculated based on one or more domain-driven variables $var_1, var_2, ..., var_k$ of $nw$ and is defined as follows. $NI(nw) = f_{nw}(var_1, var_2, ..., var_k)$, where $f_{nw}$ is the function for calculating the importance of $nw$.

Table 6.4 shows values for two domain-driven variables *popularity* and *published date* (i.e., *recency*) of six news articles. In this table, given news $nw$ and the two variables, we calculate the importance of each news article as follows. $NI(nw) = f_{nw}(popularity, recency) = popularity \times recency$.

**Definition 54** The **interestingness of news** $nw$ **to user** $usr$ **in session** $S_r$ is a score calculated based on user engagement measures. Given set of measures $em_1, em_2, ..., em_k$, the interestingness is denoted and defined as follows. $INS(nw, S_r) = f_{ins}(em_1, em_2, ..., em_k)$, where $f_{ins}$ is the function for calculating interestingness score.

For example in Table 6.3, three measures *Shared*, *Liked* and *Time Spent* are considered to calculate the interestingness score of news $nw$ to user $usr$. In this example, $f_{ins}$ is defined as $f_{ins}(Shared, Liked, Time\ Spent) = val(Shared) + val(Liked) + Norm(val(Time\ Spent))$, where $val$ returns the value of the measure and $Norm(Time\ Spent)$ is the normalized value of Time Spent. The last column in this table shows the converted dataset based on the calculated scores.

**Definition 55** (**News Attractive Model**) Given news $nw$ and session $S_r$, *news attractive model* is defined as a combination of news importance and interestingness of $nw$ to $usr$ as follows. $nam(nw, S_r) = f_{nam}(NI(nw), INS(nw, S_r))$, where $f_{nam}$ is the function for calculating news attractiveness.

**Definition 56** Given a reading behavior pattern $P = \{nw_1, nw_2, ..., nw_L\}$ where $L$ is the length of $P$, the **attractiveness of** $P$ **in session** $S_r$ is defined and denoted as: $nam(P, S_r) = \sum_{nw \in P} nam(nw, S_r)$.

**Definition 57** The **attractiveness of reading behaviour** $P$ **in clickstream dataset** $D$ is defined and denoted as: $nam(P, D) = \sum_{S_r \in D} \sum_{nw \in P} nam(nw, S_r)$.

**Definition 58** (**Attractive reading behaviour in a web clickstream dataset** $D$) A reading behaviour $P$ is an *attractive reading behaviour* iff $nam(P, D)$ is not less than a user specified minimum attractiveness threshold.

### 6.1.2 Demonstration

Our goal is to take both news importance and interestingness into account when discovering users' reading behaviours. Here, we assume that more recent news are more important[7] and the time user spends on a news article reflects his/her interest in the news,

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

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 *interestingness* of $nw$ to $usr$ is defined as browsing time (in seconds) that $usr$ spent on $nw$[8]. In addition, since the importance of *nw* is dynamic and varying from time to time, the *importance* of *nw* is defined as:

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

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

Given news $nw$ and a session $S_r$ in the web clickstream dataset $D$, the attractiveness (i.e., utility[9]) of $nw$ in $S_r$ is defined as: $nam(nw, S_r) = timeSpent(nw, S) \times NI(nw)$.

We apply *MAHUSP* to discover HUSPs based on the above utility model (e.g., news attractive model). We also apply SPADE algorithm implemented by [26] to discover frequent sequential patterns (i.e.,*FSPs*) from the *Globe* dataset. Table 6.1 presents top-4 HUSPs (i.e., attractive reading behaviour) and top-4 FSPs of length 2, sorted by time spent and support respectively. Table 6.1 suggests that the pattern with high support is not necessarily a pattern of users' interest if we use time-spent as the interestingness

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

[9]The news attractive model can be plugged in as desired. The use of more sophisticated model may further improve the quality of the results.

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 recommendations to navigate users based on a semantic measure (e.g., news freshness and interestingness) rather than a statistical measure (e.g., support). For example, in Table 6.1, if a user reads the first article of $HUSP_1$ whose title is "*retiree, 60, wonders how long her money will last*", we can recommend the second article in $HUSP_1$ with title "*which is better, a RRIF or an annuity? You may be surprised*". These patterns are also useful for the portal designers to understand users' navigation behavior and improve the portal design and e-business strategies.

We also evaluate the performance of *MAHUSP* in comparison to the other methods implemented in Section 3.4 of Chapter 3. Figure 6.1 shows the results in terms of *run time* and *memory usage*. *NaiveHUSP* is the fastest method due to the fact that it only keeps the utility of each item over data streams. However, its utility approximation is inaccurate and causes a high rate of false positives. *USpan* is the slowest, since it re-runs the whole mining process on the current data stream to discover HUSPs. Figure 6.1(b) shows the memory usage of the methods. Since *RndHUSP*, *MAHUSP_L* and *MAHUSP_S* consume the same amount of memory, we only present the results of *MAHUSP_S*, *Naive-HUSP* and *USpan*. *NaiveHUSP* uses more memory than *MAHUSP_S* on *Globe* since it needs to keep a huge list of items and their utilities into the memory. Figure 6.2 shows the *Precision*, *Recall* and *F-Score* values for the four methods with different $\delta$ values on the

Table 6.2: (a) An example of a time course microarray dataset, (b) Fold changes of gene/probe values

| *Patient IDs* | *Genes* | $TP_1$ | $TP_2$ | $TP_3$ | $TP_4$ |
|---|---|---|---|---|---|
| $P_1$ | $G_1$ | 240 | 546 | 100 | 50 |
| | $G_2$ | 321 | 98 | 454 | 974 |
| | $G_3$ | 410 | 350 | 251 | 243 |
| $P_2$ | $G_1$ | 128 | 786 | 135 | 344 |
| | $G_2$ | 253 | 820 | 482 | 90 |
| | $G_3$ | 290 | 150 | 256 | 864 |
| $P_3$ | $G_1$ | 600 | 188 | 99 | 40 |
| | $G_2$ | 500 | 555 | 510 | 80 |
| | $G_3$ | 200 | 400 | 350 | 450 |

$(a)$

| *Patient IDs* | *Genes* | $TP_1$ | $TP_2$ | $TP_3$ | $TP_4$ |
|---|---|---|---|---|---|
| $P_1$ | $G_1$ | 1 | 2.2 | $-2.4$ | $-4.8$ |
| | $G_2$ | 1 | $-3.2$ | 1.4 | 3.0 |
| | $G_3$ | 1 | $-1.1$ | $-1.6$ | $-1.6$ |
| $P_2$ | $G_1$ | 1 | 6.1 | 1.0 | 2.6 |
| | $G_2$ | 1 | 3.2 | 1.9 | $-2.8$ |
| | $G_3$ | 1 | $-1.9$ | $-1.1$ | 2.9 |
| $P_3$ | $G_1$ | 1 | $-3.1$ | $-6.6$ | $-15$ |
| | $G_2$ | 1 | 1.1 | 1.0 | $-6.2$ |
| | $G_3$ | 1 | 2 | 1.7 | 2.2 |

$(b)$

*Globe* dataset. *MAHUSP_S* and *MAHUSP_L* outperform the other methods significantly with an average Precision, Recall and F-score value of 95%, 75% and 83% respectively, over the $Globe$ dataset.

## 6.2 A Disease-related Gene Expression Sequence Discovery

Microarray has been widely used in the biomedical field for discovering differentially expressed genes in human diseases. Recently, time course issue analysis has become critical in illness events such as cancer formation. Such diseases have to be studied and monitored for a period of time to identify abnormal alternations in gene expression. Such alternations may cause an interruption of basal condition, thus ease the cell death.

Data mining techniques such as frequency-based sequential pattern mining approach [20] have been applied on microarray datasets to identify potential gene regulations that

occur in a period of time. These methods mostly choose important gene expression sequences based on the *frequency/support* framework. However, as clinical studies have shown, the frequency alone may not be informative enough to discover gene expression sequences regarding an specific disease. For example, some genes are more important than others in causing a particular disease and some genes are more effective than others in fighting diseases. The sequences contain these highly valuable genes may not be discovered by the frequency-based approaches because they neither consider the importance of each gene, nor temporal behavior of genes under biological investigation.

As the second application, we propose a new approach to identifying disease-related gene expression sequences by taking the importance of genes with respect to a specific disease and their temporal properties under biological treatments into account. We conduct an analysis on a time course gene expression microarray dataset, called GSE6377[47], downloaded from the GEMMA database[10]. Below we first define a utility model to discover disease-related gene expression sequences effectively and then we present how a time-course microarray dataset is converted to a utility-based sequential database. Finally, we apply MAHUSP to find disease-related gene expression sequences from the dataset.

[10]http://www.chibi.ubc.ca/Gemma/home.html

### 6.2.1 Definitions

In this section, we first adopt definitions related to high utility pattern mining to the context of the application and then we define the utility model to discover disease-related gene expression sequences.

Let $G = \{g_1, g_2, ..., g_n\}$ be a set of distinct genes. A *geneset* $GI$ is a set of genes. A *time-course gene expression sequence* dataset is a set of patients $\{P_1, P_2, ...., P_K\}$, where each patient has a patient identifier $P_r$ and consists of an ordered list of *time point (TP) samples* where each $TP$ is a geneset. The time point sample $TP_d$ for patient $P_r$ is denoted as $P_r^d$. Table 6.2(a) shows an example of time course microarray dataset obtained from a biological investigation which consists of three patients whose IDs are $P_1$, $P_2$ and $P_3$. In this table, the gene expression values of three genes $G_1$, $G_2$ and $G_3$ are presented over four time point samples $TP_1$, $TP_2$, $TP_3$ and $TP_4$.

**Definition 59** The **importance of gene** $g$ is a score which is calculated based on one or more disease-related variables $var_1, var_2, ..., var_k$ which is defined as follows. $GI(g) = f_g(var_1, var_2, ..., var_k)$, where $f_g$ is the function for calculating the importance of $g$.

**Definition 60 (Temporal behavior of gene $g$ in time point sample $TP_r^d$ w.r.t. disease** $dis$.) A real value is assigned to each gene $g$ in time point sample $TP_d$ of patient $P_r$ (i.e., $P_r^d$) that specifies the relative abundance of that gene in the time point and is denoted as $EGS_{dis}(g, P_r^d)$.

**Definition 61** (**Gene-Disease Association (GDA)**) Given gene $g$ and time point $P_r^d$, *Gene-Disease Association* is defined as a combination of gene importance and temporal behavior of $g$ w.r.t. disease $dis$ as follows. $GDA(g, P_r^d) = f_{gda}(GI(g), EGS(g, S_r^d))$, where $f_{gda}$ is the function for calculating association score.

**Definition 62** The **Gene-Disease Association of a geneset** $GI$ **in a time point sample** $TP_d$ **of a patient** $P_r$ where $GI \subseteq TP_d$, is defined as $GDA(GI, P_r^d) = \sum_{g \in GI} GDA(g, S_r^d)$.

**Definition 63** (**Occurrence of a gene expression sequence** $\alpha$ **in a patient** $P_r$**)** Given a patient $P_r = \langle P_r^1, P_r^2, ..., P_r^n \rangle$ and a gene expression sequence $\alpha = \langle GI_1, GI_2, ..., GI_Z \rangle$ where $P_r^i$ is a time point sample and $GI_i$ is a geneset, $\alpha$ occurs in $P_r$ iff there exist integers $1 \leq e_1 < e_2 < ... < e_Z \leq n$ such that $GI_1 \subseteq P_r^{e_1}, GI_2 \subseteq P_r^{e_2}, ..., GI_Z \subseteq P_r^{e_Z}$. The ordered list of genesets $\langle P_r^{e_1}, P_r^{e_2}, ..., P_r^{e_Z} \rangle$ is called an *occurrence of* $\alpha$ *in* $P_r$. The set of all occurrences of $\alpha$ in $P_r$ is denoted as $OccSet(\alpha, P_r)$.

**Definition 64** (**Gene-disease association of a gene expression sequence** $\alpha$ **in a patient sequence** $P_r$**)** Let $\tilde{o} = \langle P_r^{e_1}, P_r^{e_2}, ..., P_r^{e_Z} \rangle$ be an occurrence of $\alpha = \langle GI_1, GI_2, ..., GI_Z \rangle$ in the sequence $P_r$. The gene-disease association of $\alpha$ w.r.t. $\tilde{o}$ is defined as $GDA(\alpha, \tilde{o}) = \sum_{i=1}^{Z} GDA(GI_i, P_r^{e_i})$. The gene-disease association of $\alpha$ in $P_r$ is defined as $GDA(\alpha, P_r) = \max\{GDA(\alpha, \tilde{o}) \mid \tilde{o} \in OccSet(\alpha, P_r)\}$.

**Definition 65** (**Gene-disease association of a gene expression sequence** $\alpha$ **in a time course gene expression sequence dataset** $D$**)** The gene-disease association of a gene ex-

pression sequence $\alpha$ in a time course dataset $D$ is defined as $GDA(\alpha, D) = \sum_{P_r \in D} GDA(\alpha, P_r)$.

**Definition 66 (Important Disease-related Gene Expression Sequence (IDGS))** Given a threshold $\delta$, a gene expression sequence $\alpha$ is an *Important Disease-related Gene Sequence* (IDGS) in time course dataset $D$, iff $GDA(\alpha, D)$ is no less than $\delta$.

**Utility-based Sequential Database Construction.** In order to discover disease-related gene expression sequences from a time course microarray dataset, we need to convert the dataset into a utility based sequential database.

In each time point, each gene has a temporal behavior which is expressed by a real value. We consider the first time point as a baseline to derive the temporal behaviour of each gene at each time point. Hence, the expression values in each time point are divided by the first time point values. Table 6.2(b) shows the output as a fold change matrix.

Given the fold change matrix and a threshold $\gamma$, each expression value in the dataset is transformed as *up-regulated* (representing by $+$ meaning that values are greater than $\gamma$), *down-regulated* (representing by $-$ meaning that values are less than -$\gamma$), or *normal* (neither expressed nor repressed) and only the gene expressions that are up-regulated or down-regulated are preserved. Each gene (i.e., $G_x$) in a sample can be thought of as being two $items$, one item referring to the gene being up (i.e., $G_{x^+}$), the other referring to the gene being down (i.e., $G_{x^-}$).

Given $\gamma = 1.5$, Table 6.3 shows the converted dataset. For example, in patient $P_1$,

Table 6.3: Converted utility-based sequential dataset from time course microarray dataset in Table 6.2(a)

| *Patient IDs* | *Sequence* |
|---|---|
| $P_1$ | $\{G_1{+}(2.2)G_2{-}(3.2)\}_2\ \{G_1{-}(2.4)G_3{-}(1.6)\}_3\{G_1{-}(4.8)G_2{+}(3.0)G_3{-}(1.6)\}_4$ |
| $P_2$ | $\{G_1{+}(6.1)G_2{+}(3.2)G_3{-}(1.9)\}_2\ \{G_2{+}(1.9)\}_3\{G_1{+}(2.6)G_2{-}(2.8)G_3{+}(2.9)\}_4$ |
| $P_3$ | $\{G_1{-}(3.1)G_3{+}(2.0)\}_2\ \{G_1{-}(6.6)G_3{+}(1.7)\}_3\{G_1{-}(15)G_2{-}(6.2)G_3{+}(2.2)\}_4$ |

$(a)$

| *Gene* | $G_1$ | $G_2$ | $G_3$ |
|---|---|---|---|
| *Score* | 0.8 | 0.6 | 0.1 |

$(b)$

up-regulated $G_{1+}(2.2)$ and down-regulated $G_{2-}(3.2)$ are considered to occur at the same time point (i.e., $TP_2$). In this dataset, a set of time points (i.e., TPs) for each patient forms a sequence. Given gene $g$ and time point $TP_i$, its absolute fold change value represents $EGS(g, TP_i)$. Table 6.3(b) shows the importance of genes with respect to a disease. In this work, the importance of $G_x$ represents the importance of both $G_{x^+}$ and $G_{x^-}$ gene items.

### 6.2.2 Demonstration

In order to evaluate our proposed utility model and also the performance of MAHUSP to find disease-related gene expression sequences (i.e., IDGSs) from a time course gene expression dataset, we mine the *GSE6377* dataset. McDunn et al.[47] attempted to detect 8,793 transcriptional changes in 11 ventilator-associated pneumonia patients leukocytes across 10 time points. Our goal is to decipher *pneumonia-related* gene expression se-

Table 6.4: Top-20 genes related to pneumonia

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

quences. We also consider the *score* proposed by *DisGeNET* [11] as the importance of each gene with respect to a disease. This score considers several variables such as number and type of sources (level of curation, model organisms) and the number of publications supporting the association to rank genes with respect to a specific disease.

We apply *MAHUSP* to extract HUSPs (i.e., IDGSs) based on the proposed utility model. We also run a frequency-based algorithm, *PrefixSpan*[50], to discover frequent gene expression sequences (i.e., *FGSs*) from the dataset. Table 6.5 shows top-4 HUSPs (e.g., disease-related gene expression sequences) extracted by MAHUSP and top-4 FGSs extracted by *PrefixSpan*, sorted by the utility (i.e., $GDA$) value and support respectively. Given a gene expression sequence $S_i$ and disease $dis$, we evaluate the quality of the results using *popularity of a sequence score* [13] which is defined as follows: $Pop(\alpha, dis) = \frac{\sum_{i \in alpha} w(i,dis)}{|\alpha|}$, where $w(i, dis)$ is the importance of popular gene $i$ for disease $dis$. We consider the genes presented in Table 6.4 as popular genes and $w(i, dis) = 20 - rank(i, dis) + 1$. For the genes not presented in the list, $w(i, dis) = 1$.

[11] http://www.disgenet.org/web/DisGeNET/menu

Table 6.5 suggests that the frequent gene expression sequences are not necessarily popular w.r.t. the disease even though their support value is more than 90%. This is due to the fact that these patterns are discovered based on their frequency in the dataset which is not informative enough. On the other hand, MAHUSP returns the patterns whose popularity is relatively high. These patterns help biologists select relevant sequences regarding a specific disease and also identify the relationships between important genes and the other genes.

Table 6.5: Top-4 HUSPs versus Top-4 FGSs with respect to utility and support

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

Figure 6.5 shows the performance of different methods on the *GSE6377* dataset in terms of run time and memory usage. The dataset is a dense dataset and as we expected the number of HUSPs is huge. For example, given threshold $0.07$, there are $5,6542,360$ HUSPs in the dataset. In this experiment $availMem$ is set to $2GB$. The proposed methods outperform *USpan* significantly. *NaiveHUSP* is the fastest method since it works based on item utilities in the dataset to find HUSPs. But, its false positive rate is high

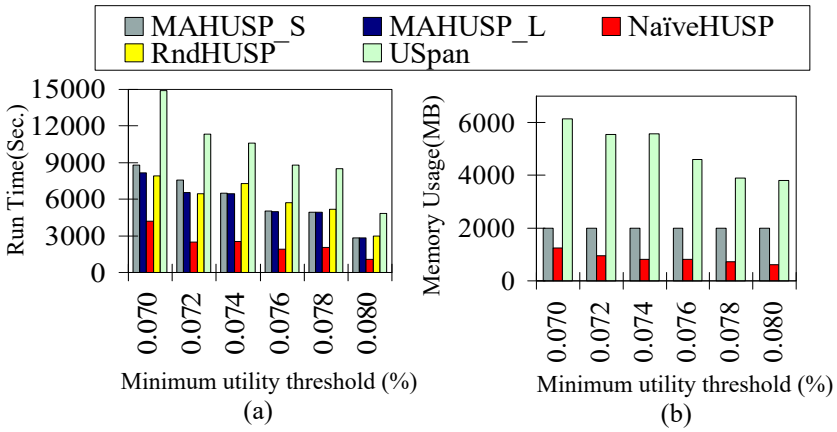



Figure 6.5: (a) Run time, (b) Memory Usage on the *GSE6377* dataset

due to its inaccurate approximate utility. Figure 6.6 shows *Precision*, *Recall* and *F-Score* values for the four methods. In general, both proposed methods outperform the other methods with an average Precision, Recall and F-score values of 75%, 67% and 71% over the *GSE6377* dataset.

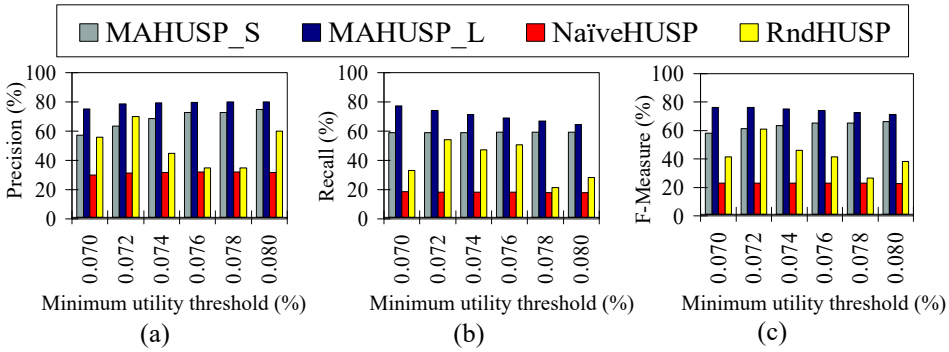



Figure 6.6: (a) Precision, (b) Recall and (c) F-Measure performance on the *GSE6377* dataset

## 6.3 Summary

In order to demonstrate the applicability of the proposed methods in practical cases, we discover meaningful patterns in two real-life applications. Our contributions are summarized as follows.

- We conducted an analysis on a real web clickstream dataset obtained from a Canadian news web portal to extract web users' reading behavior patterns.
- We analyzed a publicly available time course microarray dataset to identify gene sequences correlated with a specific disease.
- Using several quality measures, the mined utility-based sequential patterns are compared with the patterns in the frequency/support framework. The evaluation results showed that our approach can effectively discover key patterns representing user reading behavior in news domain and disease-related gene expression sequences in biomedical domain. The results of this chapter also showed the potential value of this work in real-life applications in terms of discovering meaningful patterns from sequence databases.

# 7 Conclusions and Future Work

Mining of streaming data for extracting novel insights is a fundamental task in many domains such as *market analysis*, *web mining*, *mobile computing* and *network analysis*. One of the important problems in such domains is identifying informative sequential patterns with respect to a business objective, such as patterns that represent *profitable purchase sequences* in *market analysis*, or *sequences of web pages* related to *users' interest* in *web mining*. These patterns can be discovered using *high utility sequential pattern mining (HUSP)* methods. The main objective of HUSP mining is to extract valuable and useful sequential patterns from data by considering a business objective such as *profit*, *user's interest*, *cost*, etc.

A number of studies have been conducted on mining HUSPs, but they are mainly intended for non-streaming data and thus do not take data stream characteristics into consideration. Mining HUSP from such data poses many challenges. First, it is infeasible to keep all streaming data in the memory. Second, mining algorithms need to process the arriving data in real time with one scan of data. Third, the distribution of data varies

over time, and hence analysis results need to be updated in real time. Last but not least, depending on the minimum utility threshold value, the number of patterns returned by a HUSP mining algorithm can be large and overwhelms the user. In general, it is hard for the user to determine the value for the threshold. Thus, algorithms that can find the most valuable patterns (i.e., top-k high utility patterns) are more desirable. Mining the most valuable patterns is interesting in both static data and data streams.

## 7.1 Summary of Contributions

To address these research limitations and challenges, this dissertation proposed both threshold-based and top-k-based algorithms for discovering high utility sequential patterns over data streams. We worked on mining HUSPs over both a long portion of a data stream and a short period of time. We made the following contributions. Original research has been accomplished in pursuit of this degree, and the results have been published in [72], [73], [74], [75], [76].

First, we tackled the problem of memory adaptive HUSP mining over data streams. We proposed an approximation algorithm, called *MAHUSP*, to discover HUSPs over the entire data streams. *MAHUSP* is based on a compressed tree structure and two memory adaptive mechanisms that can adapt the memory usage to the available memory by pruning the least promising part of the tree when necessary. We proved that *MAHUSP* returns all the true HUSPs under certain circumstances. 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.

Second, we presented a novel approach for *mining recent high utility sequential patterns over data streams*. We proposed an algorithm, called *HUSP-Stream*, to discover high utility sequential patterns in a transaction-sensitive sliding window over a sequence data stream. Two data structures named *ItemUtilLists* and *HUSP-Tree* were proposed to dynamically maintain the essential information of potential HUSPs over data streams. 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-Stream with two approaches that use USpan [66], a state-of-the-art algorithm for mining HUSPs in static databases, to learn HUSPs over data streams. The experiments showed that HUSP-Stream substantially outperforms 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 DS1 dataset when the minimum utility threshold is 0.1%). The experimental results also showed that our proposed SFU tree-pruning strategy is much more effective than the TSWU strategy. Our approach is scalable in both time and memory, and

serves as an efficient solution to the new problem of mining recent high utility sequential patterns over data streams.

Third, we proposed an efficient algorithm, called *T-HUDS*, for mining top-k high utility itemsets in sliding windows over streaming data. *T-HUDS* uses a novel over-estimate utility model, i.e., the *PrefixUtil* model, to effectively prune the search space for finding top-$k$ HUIs. We proved that *PrefixUtil* satisfies a special type of the downward closure property, which allows it to be effectively used to prune the search space in a pattern growth process. We also addressed a major challenge in top-$k$ pattern mining by devising several strategies for initializing and raising the minimum utility threshold during the mining process. A *FP-tree*-like data structure, *HUDS-tree*, and two auxiliary lists, *maxUtilList* and *MIUList*, are designed to store the information that is needed for computing $PrefixUtil$ and for initializing and dynamically adjusting the threshold. We also designed a strategy that uses the information from the top-$k$ patterns in the previous window to help initialize the threshold for the new window. In addition, in the second phase of top-$k$ HUI mining, the $min\_util$ threshold is also raised to help fast find the top-$k$ patterns from the candidates. We proved that using these strategies to raise the threshold and using $PrefixUtil$ to prune the search space do not miss any top-k HUIs. These strategies not only help find top-k high utility itemsets effectively, they also reduce the run time and memory consumption of the algorithm significantly. Inspired by T-HUDS, we extended HUSP-Stream and proposed a single pass algorithm, called *T-HUSP*, to incre-

mentally maintain the content of top-k HUSPs in the sliding window in a summary data structure, named *TKList*, and discover top-k HUSPs efficiently. In addition, two efficient strategies have been proposed for raising the threshold. Our experiments are conducted on both synthetic and real datasets. The results show that both methods incorporating the efficiency-enhanced strategies demonstrate impressive performance without missing any high utility itemset/sequential patterns.

Moreover, in this dissertation, we showed the effectiveness and efficiency of the proposed methods in real-life applications. We applied one of the proposed methods (i.e., MAHUSP) to a real web clickstream dataset and a real biosequence dataset to find meaningful patterns. The mined utility-based sequential patterns are compared with the patterns in the frequency/support framework. The results showed that high utility sequential pattern mining provides more meaningful patterns in real-life applications.

## 7.2 Future Work

Although we are the first to 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 dissertation, there are still ample room for exploration in the future.

1. **Dynamic landmark window:** Our proposed method in the first chapter (MAHUSP) discovers high utility sequential patterns (HUSPs) over a specific type of landmark window in which the *landmark* was set to the beginning of the data stream. As future work, we will design an approach such that users can dynamically change the *landmark* and discover patterns within the updated landmark window efficiently.

2. **Different approaches to calculate utility:** In this work, we measured the utility of a sequence by its maximum utility and did not consider other approaches to calculate the utility of a sequence. It would be worthwhile to explore new approaches to calculate utility. The major challenge is that the correctness of the proposed overestimate utility models (e.g., SFU, RSU and TSWU) in this dissertation may not hold when using other approaches. Below, we present two approaches that we will explore to calculate utility: (1) Weighted sum of occurrences' utility: in some applications (e.g., market basket analysis), the first occurrence is more important than the later ones in terms of customer acquisition factors. One approach is to assign different weights to different occurrences (e.g., based on the order of appearance in the sequence) and define utility as a weighted sum of utility of the occurrences. (2) Distance-based utility: in some applications (e.g., gene expression sequence discovery), we are dealing with long sequences. Considering only one occurrence (the one with maximum utility) may not present the true utility of a gene expression sequence and we may lose some important gene expression

sequences. For such applications, it is better to consider the sum of the maximum utility of occurrences within certain distances (e.g., the time distance among items/itemsets belonging to an occurrence is no more than a threshold) or different intervals (e.g., sum of maximum utility of occurrences for every month). In this way, we will have better insight about the utility of a sequence.

3. **Resource-aware high utility sequential pattern mining over data streams:** With the pass-through features of data streams, resources other than memory such as computation power, bandwidth or CPU, are particularly valuable in the streaming environment. For example, the available CPU will affect the processing speed and consequently the performance of the algorithms. How to keep the pace with high speed data streams while the processing speed changes is a challenging problem. As future work, we will extend our methods to consider such resources. The overall goal will be to maximize effectiveness of the methods by making the best use of available resources dynamically and adaptively.

# Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

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

[3] C. F. Ahmed, S. Tanbeer, and B. Jeong. A framework for mining high utility web access sequences. *In IETE Journal*, 28:3–16, 2011.

[4] 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.

[5] 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.

[6] C. F. Ahmed, S. K. Tanbeer, and B. Jeong. A framework for mining high utility web access sequences. *In IETE Journal*, 28:3–16, 2011.

[7] C. F. Ahmed, S. K. Tanbeer, and B. S. Jeong. Interactive mining of high utility

patterns over data streams. *Expert Systems with Applications*, 39:11979–11991, 2012.

[8] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, and Y. K. Lee. Efficient tree structures for high-utility pattern mining in incremental databases. *IEEE Transactions on Knowledge and Data Engineering*, 21:1708–1721, 2009.

[9] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *In Proc. of ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 429–435, 2002.

[10] V. S. T. B. E. Shie, P. S. Yu. Efficient algorithms for mining maximal high utility itemsets from data streams with different models. *Expert Systems with Applications*, 39:12947–12960, 2012.

[11] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[12] B. Berendt and M. Spiliopoulou. Analysis of navigation behaviour in web sites integrating multiple information systems. *The VLDB Journal*, 9(1):56–75, Mar. 2000.

[13] S. Bringay. Discovering novelty in sequential patterns: application for analysis of microarray data on alzheimer disease. In *Studies in health technology and informatics*, pages 1314–1318, 2010.

[14] A. G. Büchner and M. D. Mulvenna. Discovering internet marketing intelligence through online analytical web usage mining. *SIGMOD Rec.*, 27(4):54–61, Dec. 1998.

[15] P. P. C. Rassi and M. Teisseire. Speed : Mining maximal sequential patterns over data streams. In *In Proc. of the IEEE Int'l Conf. on Intelligent Systems*, pages 546–552, 2006.

[16] L. Cao, Y. Zhao, H. Zhang, D. Luo, C. Zhang, and E. Park. Flexible frameworks for actionable knowledge discovery. *Knowledge and Data Engineering, IEEE Transactions on*, 22(9):1299–1312, 2010.

[17] R. Chan, Q. Yang, and Y. Shen. Mining high-utility itemsets. In *Proc. of Third IEEE Int'l Conf. on Data Mining*, pages 19–26, 2003.

[18] L. Chang, T. Wang, D. Yang, and H. Luan. Seqstream: Mining closed sequential patterns over stream sliding windows. In *In Proc. of the IEEE International Conf. on Data Mining*, pages 83–92, 2008.

[19] G. Chen, X. Wu, and X. Zhu. *Mining Sequential Patterns across Data Streams*. PhD thesis, University of Vermont, 2005.

[20] C.-P. Cheng, Y.-C. Liu, Y.-L. Tsai, and V. S. Tseng. An efficient method for mining

cross-timepoint gene regulation sequential patterns from time course gene expression datasets. *BMC Bioinformatics*, 14(12):1–12, 2013.

[21] J. Cheng, Y. Ke, and W. Ng. A survey on algorithms for mining frequent itemsets over data streams. *Knowledge and Information Systems*, 16:1–27, 2008.

[22] Y. L. Cheung and A. W. Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE Transactions on Knowledge and Data Engineering*, 16:1052–1069, 2004.

[23] K. Chuang, J. Huang, and M. Chen. Mining top-k frequent patterns in the presence of the memory constraint. *The VLDB Journal*, 17:1321–1344, 2008.

[24] A. Erwin, R. P. Gopalan, and N. R. Achuthan. A bottom-up projection based algorithm for mining high utility itemsets. In *Proceedings of 2nd International Workshop on Integration Artificial Intelligence and Data Mining*, pages 3–11, 2007.

[25] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C. Wu, and V. S. Tseng. Spmf: a java open-source pattern mining library. *Journal of Machine Learning Research (JMLR)*, 15:3389–3393, 2014.

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

[27] B. Goethals and M. J. Zaki. Frequent itemset mining dataset repository, http://fimi.cs.helsinki.fi/data/, 2004.

[28] L. Golab, D. Dehaan, and E. Demaine. Identifying frequent items in sliding windows over on-line packet streams. In *Proceedings of ACM SIGCOMM internet measurement conference*, pages 173–178, 2003.

[29] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *In Proc.of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 355–359, 2010.

[30] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29:1–12, 2000.

[31] Y. Hirate, E. Iwahashi, and H. Yamana. Tf2p-growth: An efficient algorithm for mining frequent patterns without any thresholds. In *Proc. of IEEE ICDM'04 Workshop on Alternative Techniques for Data Mining and Knowledge Discoverey*, 2004.

[32] C. Ho, H. Li, F. Kuo, and S. Lee. Incremental mining of sequential patterns over a stream sliding window. In *In Proc. of the ICDM Workshops*, pages 677–681, 2006.

[33] L. Junqiang, W. Ke, and B. Fung. Direct discovery of high utility itemsets with-

out candidate generation. In *12th IEEE International Conference on Data Mining (ICDM)*, pages 984–989, 2012.

[34] G.-C. Lan, T.-P. Hong, and V. S. Tseng. Tightening upper bounds of utility values in utility mining. In *In Proceedings of the 28th workshop on Combinatorial Mathematics and Computation Theory*, pages 11–16, 2011.

[35] K. S. C. Leung and F. Jiang. Frequent itemset mining of uncertain data streams using the damped window model. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 950–955, 2011.

[36] H. F. Li, H. Y. Huang, Y. C. Chen, Y. J. Liu, and S. Y. Lee. Fast and memory efficient mining of high utility itemsets in data streams. In *Proc. of the 8th IEEE Int'l Conf. on Data Mining*, pages 881–886, 2008.

[37] Y. C. Li, J. S. Yeh, and C. C. Chang. Isolated items discarding strategy for discovering high utility itemsets. *Data and Knowledge Engineering*, 64:198–217, 2008.

[38] 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.

[39] B. Liu, W. Hsu, H.-S. Han, and Y. Xia. Mining changes for real-life applications. In *Proceedings of the Second International Conference on Data Warehousing and*

*Knowledge Discovery*, DaWaK 2000, pages 337–346, London, UK, UK, 2000. Springer-Verlag.

[40] M. Liu and J. Qu. Mining high utility itemsets without candidate generation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 55–64, 2012.

[41] M. Liu and J. Qu. Mining high utility itemsets without candidate generation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 55–64, 2012.

[42] Y. Liu, W. k. Liao, and A. Choudhary. A fast high utility itemsets mining algorithm. In *Proceedings of the 1st international workshop on Utility-based data mining*, pages 90–99, 2005.

[43] Y. Liu, W. Liao, and A. Choudhary. A two-phase algorithm for fast discovery of high utility of itemsets. In *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 689–695, 2005.

[44] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Comput. Surv.*, 43(1):3:1–3:41, 2010.

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

[46] A. Marascu and F. Masseglia. Mining sequential patterns from temporal streaming data. In *In Prc. of the ECML/PKDD Workshop on Mining Complex Data*, pages 355–359, 2005.

[47] J. McDunn, K. Husain, A. Polpitiya, A. Burykin, J. Ruan, Q. Li, W. Schierding, N. Lin, D. Dixon, and W. Zhang. Plasticity of the systemic inflammatory response to acute infection during critical illness: development of the riboleukogram. *PloS one*, 3(2):e1564, 2008.

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

[49] S. Ngan, T. Lam, R. C. Wong, and A. W. Fu. Mining n-most interesting itemsets without support threshold by the cofi-tree. *Int. J. Business Intelligence and Data Mining*, 1:88–106, 2005.

[50] J. Pei, J. Han, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *TKDE*, 16:1424–1440, 2004.

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

[52] B. Shie, H. Hsiao, and V. S. Tseng. Efficient algorithms for discovering high utility

user behavior patterns in mobile commerce environments. *In KAIS journal*, 37, 2013.

[53] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *In Proc. of the Intel Conf. on Extending Database Technology: Advances in Database Technology*, pages 3–17, 1996.

[54] F. Tao, F. Murtagh, and M.Farid. Weighted association rule mining using weighted support and significance framework. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 661–666, 2003.

[55] V. Tseng, B. Shie, C.-W. Wu, and P. Yu. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Transactions on Knowledge and Data Engineering*, 25:1772–1786, 2013.

[56] V. S. Tseng, C. J. Chu, and T. Liang. Efficient mining of temporal high-utility itemsets from data streams. In *ACM KDD Utility Based Data Mining*, pages 18–27, 2006.

[57] V. S. Tseng, C. W. Wu, B. E. Shie, and P. S. Yu. Up-growth: an efficient algorithm for high utility itemset mining. In *Proc. of Int'l Conf. on ACM SIGKDD*, pages 253–262, 2010.

[58] P. Tzvetkov, X. Yan, and J. Han. Tsp: Mining top-k closed sequential patterns. *Knowledge and Information Systems*, 7:438–457, 2005.

[59] T. H. N. Vu, K. H. Ryu, and N. Park. A method for predicting future location of mobile user for location-based services system. *Computers and Industrial Engineering*, 57(1):91 – 105, 2009. Collaborative e-Work Networks in Industrial Engineering.

[60] R. C. W. Wong and A. W. C. Fu. Mining top-k frequent itemsets from data streams. *Data Mining and Knowledge Discovery*, 13:193–217, 2006.

[61] C. W. Wu, B. E. Shie, V. S. Tseng, and P. S. Yu. Mining top-k high utility itemsets. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 78–86, 2012.

[62] B. Yang and H. Huang. Topsil-miner:an efficient algorithm for mining top-k significant itemsets over data streams. *Data Mining and Knowledge Discovery*, 23:225–242, 2010.

[63] H. Yao and H. J. Hamilton. Mining itemset utilities from transaction databases. *Data and Knowledge Engineering*, 59:603–626, 2006.

[64] H. Yao, H. J. Hamilton, and C. J. Butz. A foundational approach to mining itemset

utilities from database. In *Proceeding of the 4th SIAM International Conference on Data Mining*, pages 482–491, 2004.

[65] S. J. Yen, Y. S. Lee, C. W. Wu, and C. L. Lin. An efficient algorithm for maintaining frequent closed itemsets over data stream. In *Proceedings of IEA/AIE*, pages 767–776, 2009.

[66] 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.

[67] J. Yin, Z. Zheng, L. Cao, Y. Song, and W. Wei. Efficiently mining top-k high utility sequential patterns. In *IEEE 13th International Conference on Data Mining (ICDM)*, pages 1259–1264, 2013.

[68] U. Yun. Efficient mining of weighted interesting patterns with a strong weight and/or support affinity. *Information Sciences*, 177:3477–3499, 2007.

[69] U. Yun and J. J. Leggett. Wfim: Weighted frequent itemset mining with a weight range and a minimum weight. In *Proceedings of the fifth SIAM International Conference on Data Mining*, pages 636–640, 2005.

[70] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12:372390, 2000.

[71] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *In Machine Learning*, 42:31–60, 2001.

[72] M. Zihayat and A. An. Mining top-k high utility patterns over data streams. *Information Sciences*, 285:138 – 161, 2014. Processing and Mining Complex Data Streams.

[73] M. Zihayat, Y. Chan, and A. An. Memory-bounded high utility sequential pattern mining over data streams. Technical Report EECS-2015-04, York university, 2015.

[74] M. Zihayat, Z. Z. Hu, and A. An. Distributed and parallel high utility sequential pattern mining. Technical Report CSE-2016-04, York university, 2016.

[75] M. Zihayat, C.-W. Wu, A. An, and V. S. Tseng. Mining high utility sequential patterns from evolving data streams. In *ASE BD&SI '15*, pages 52:1–52:6, 2015.

[76] M. Zihayat, C.-W. Wu, A. An, and V. S. Tseng. Efficiently mining high utility sequential patterns in static and streaming data. In *Intelligent Data Analysis, Accepted*, 2016.