Information Sciences 285 (2014) 138–161

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Mining top-*k* high utility patterns over data streams

Morteza Zihayat, Aijun An *

Department of Computer Science & Engineering, York University, Toronto, Canada

ARTICLE INFO

Article history:
Received 23 March 2013
Received in revised form 31 December 2013
Accepted 26 January 2014
Available online 3 February 2014

Keywords:
High utility pattern mining
Data stream
Top-*k* pattern mining
Sliding window

ABSTRACT

Online high utility itemset mining over data streams has been studied recently. However, the existing methods are not designed for producing top-*k* patterns. 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 top-*k* high utility itemsets 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 paper, we propose a new method (named *T-HUDS*) for finding top-*k* high utility patterns over sliding windows of a data stream. The method 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. Our experimental results on real and synthetic datasets show that our strategies and new utility estimation model work very effectively and that *T-HUDS* outperforms two state-of-the-art high utility itemset algorithms substantially in terms of execution time and memory storage.

1. Introduction

Frequent pattern mining is an important task in data mining and has been extensively studied by many researchers [1,14,13]. Given a data set of transactions, each containing a set of items, frequent pattern mining is to find a set of itemsets whose support (i.e., the number of transactions containing the itemset) is no less than a minimum support count. 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 data set. In practice, some items or itemsets with low support in the data set may bring high profits due to their high prices or high frequencies inside transactions. Obviously, identifying such itemsets with high profits is important for business planning and operation. However, such itemsets may be missed by frequent pattern mining.

In view of this, high utility itemset mining has been studied recently [3,6,31,23]. An itemset is a *high utility itemset (HUI)* if its utility (such as the total profit that the itemset brings) in a data set is no less than a minimum utility threshold. Finding high utility itemsets has been considered to be important in various applications, such as retail marketing, web click analysis, and biological gene analysis [21,20,2]. However, mining HUIs is not as easy as 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

* Corresponding author.
E-mail addresses: zihayatm@cse.yorku.ca (M. Zihayat), aan@cse.yorku.ca (A. An).

http://dx.doi.org/10.1016/j.ins.2014.01.045
0020-0255/

search space during the HUI mining process. To deal with such a challenge, most of the HUI mining methods use an over-estimate utility, called *transaction weighted utility (TWU)* (to be defined in Section 2), to first find itemsets whose TWU is no less than the minimum utility threshold (called high TWU itemsets) and then compute the exact utilities of high TWU itemsets to identify those whose utility satisfies the minimum utility threshold. The benefit of using TWU is that TWU has the downward closure property, which allows the use of Apriori-like or FP-growth-like algorithms in the first phase of HUI mining to efficiently find high TWU itemsets.

Since data streams have become widespread in many fields, such as sensor network monitoring, trade management, and medical data analysis, methods for mining HUIs from data streams have been proposed [2,21,20,30]. In comparison to static data, data streams have some unique properties, such as very fast data arrival rate, unknown or unbounded size of data and inability to backtrack over previously arriving transactions. To deal with such challenges, a HUI mining method in [2] (named *HUPMS*) uses a compact data structure similar to FP-tree [14] to compress the transactions in the data set and uses a pattern growth method (similar to FP-growth) to efficiently identify all the high TWU itemsets with respect to a minimum utility threshold. HUIs are then identified from the set of high TWU itemsets after scanning the recent data in a sliding window for the second time to compute the exact utility of these itemsets. Although the use of TWU allows effective pruning of the search space due to its downward closure property, it is a very loose estimate of the true utility of an itemset. As a result, the number of high TWU itemsets found in the first phase of the method can be high and many of them do not satisfy the minimum utility threshold. Thus, the overall time for finding HUIs can be too long to satisfy the fast data processing requirement for data streams.

Another problem with the method in [2] and many other HUI mining methods is that the user needs to supply a 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 HUIs can be found, which is not only time and space consuming [33], 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 HUIs being found, which means that some interesting patterns are missed.

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 patterns 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 method for top-k HUI mining was proposed very recently in [33]. The method is designed for static data, not for data stream mining. 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. The method proposed in [33] initializes the threshold to 0 or the kth highest value of the lower bounds for the utility of certain 2-itemsets, and then gradually raise the threshold during the mining process to prune the search space. The authors proposed a few strategies for raising the threshold. However, their initial threshold is too low and can lead to generation of a large number of potential HUIs in the first phase of the method. In addition, their method is not designed for data streams.

In this paper, we propose more 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. A sliding window consists of a fixed number of most recent batches, each batch containing a set of transactions. When a new batch arrives, the sliding window moves forward to include the new batch and at the same time remove the oldest batch if the maximum number of batches in the window has been reached before the new batch comes. 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* in pruning the search space because it is a closer estimate of the true utility than *TWU*. The contributions of the paper are as follows:

- We are the first to 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.
- We propose an FP-tree-like 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 data sets, which shows that our proposed method is faster and less memory consuming than the state-of-the-art methods.

The paper is organized as follows. Preliminary definitions and a problem statement are given in Section 2. In Section 3, we describe the challenges in solving our problem and define some concepts used in our methods. In Section 4, we present the *HUDS-tree* structure and our algorithms for finding top-*k HUIs*. The experimental results are presented in Section 5. Related work is discussed in Section 6. In Section 7 we conclude the paper.

2. Preliminaries and problem statement

Let $I = \{i_1, i_2, \ldots, i_m\}$ 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: $D = \{T_1, T_2, \ldots, T_N\}$ 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 $T_j\}$. Fig. 1 shows an example of a data set with six transactions.

Definition 1. Utility of an item i in a transaction T_j is defined as: $u(i, T_j) = q(i, T_j) \times p(i)$ where $q(i, T_j)$ is the *quantity* of item i in transaction T_j and $p(i)$ is *external utility* of item i.

Definition 2. 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, $u(\{bc\}, T_3) = 2 \times 6 + 3 \times 5 = 27$ in Fig. 1.

Definition 3. Utility of an itemset X in a data set 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 data set D is clear in the context.

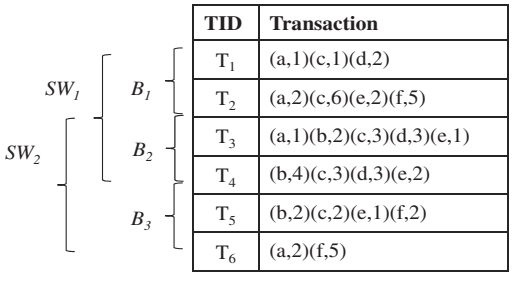
Definition 4. Utility of a transaction T_j is denoted as $TU(T_j)$ and computed as $u(T_j, T_j)$.

Definition 5. (High Utility Itemset (HUI)) An itemset X is called a high utility itemset (HUI) *on a data set D if and only if* $u_D(X) \geqslant$ *min_util where* min_util *is called a* minimum utility threshold.

A challenge in mining high utility itemsets is that the utility of an itemset does not have the downward closure (i.e., anti-monotone) property. That is, the utility of an itemset does not decrease monotonically when adding items to the itemset and it changes irregularly. Thus, unlike in frequent itemset mining, we cannot use the utility of an itemset to prune the search space in high utility itemset mining because a superset of a low utility itemset may be a high utility itemset.
To solve this problem, an over-estimate utility of an itemset (instead of the exact utility) is commonly used in the HUI mining process to prune the search space. Most of the recent methods use *transaction-weighted utility (TWU)* as the over-estimate utility.

Definition 6. Transaction-Weighted Utility (TWU) of an itemset X over a data set D is defined as: $TWU_D(X) = \sum_{X \subseteq T_j \bigwedge T_j \in D} TU(T_j)$.

Clearly, $TWU_D(X) \geqslant u_D(X)$. In addition, *TWU* satisfies the downward closure property, that is, for all $Y \subseteq X$, $TWU_D(Y) \geqslant TWU_D(X)$. Thus, most of the HUI mining methods (e.g., [21,2]) use the *TWU* values of the itemsets to prune the search space. That is, they find all the itemsets whose TWU is no less than the minimum utility threshold. Since $TWU_D(X)$ is an overestimate of $u_D(X)$, 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, these methods use a second phase to compute the exact utility of the generated itemsets and remove those whose utility is lower than the threshold.



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

Fig. 1. Example of transaction data base and external utility of items.

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 **batch** B_i consists of transactions arriving continuously in a time period, i.e., $B_i = \{T_j, T_{j+1}, \ldots, T_m\}$. For example, assuming that the dataset in Fig. 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** consists of m most recent batches, where m 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 $SW_i = \{B_i, B_{i+1}, \ldots, B_{i+winSize-1}\}$. 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 Fig. 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: $SW_2 = \{B_2, B_3\}$. Data stream mining over sliding windows is to mine patterns from each new window once a new batch is added into the new window and the oldest batch is removed from the window. The problem tackled in the paper 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.*

3. 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*,[1] is used in our HUI mining to more effectively prune the search space.

Definition 7. 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:

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

Example 1. In Fig. 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 8. 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 2. Let D be the dataset in Fig. 1. Since only T_1, T_2 and T_3 in D contain itemset $\{ac\}$, we have

$$PrefixUtil_D(\{ac\}) = PrefixUtil(\{ac\}, T_1) + PrefixUtil(\{ac\}, T_2) + PrefixUtil(\{ac\}, T_3) = 8 + 36 + 30 = 74$$

Property 1. *For any itemset X in a dataset D, the following relationship holds:*

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

Lemma 1. *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 $Z \subseteq Y$,*

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

[1] In [17], an over-estimate utility with a similar name, called *utility of full prefix extension* (U_{fpe}), was proposed. But it is different from our *Prefix Utility*. Briefly, given itemset X, the *Prefix Utility* of X in a transaction T is the sum of the utilities of those items in T that occur either before or between the items in X while u_{fpe} is the sum of the utilities of only the items before X. More discussion on their differences can be found in the related work section.

Proof 1. Let S_X be the set of transactions containing X in a data set D. According to Definition 8, 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 $PrefixUtil_{D-S_X}(Z \cup \{i\}) \geqslant 0$,

$$PrefixUtil_D(Z \cup \{i\}) \geqslant PrefixUtil_D(X). \quad \square$$

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 3. Assume that a, b and c are items in a data set and that the items in the data set are ranked in the lexicographic order. According to Lemma 1, $PrefixUtil(\{ac\}) \geqslant PrefixUtil(\{abc\})$ and $PrefixUtil(\{bc\}) \geqslant PrefixUtil(\{abc\})$. Thus, if $PrefixUtil(\{ac\})$ or $PrefixUtil(\{bc\})$ is less than a minimum utility threshold, $PrefixUtil(\{abc\})$ must be less than the threshold. Since $PrefixUtil(\{abc\}) \geqslant u(\{abc\}), u(\{abc\})$ 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 [33]. 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 9. Minimum Transaction Utility (mtu) of a transaction T is defined as: $mtu(T) = \min_{i \in T}(u(i, T))$.

For example, in Fig. 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 data set as follows.

Definition 10. Minimum Transaction Utility (MTU) of an itemset X over a data set 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 data set D is clear in the context. For example, for the data set in Fig. 1:

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

Lemma 2. *For any itemset X in a data set D, the following relationship holds:* $MTU_D(X) \leqslant u_D(X)$.

Proof 2. 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 Definitions 9 and 2, we have:

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

Hence, $mtu(T) \leqslant u(X, T)$. According to Definitions 10 and 3,

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

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

Lemma 3. *The minimum transaction utility of an itemset satisfies the downward closure property. That is, for all* $Y \subseteq X, MTU(Y) \geqslant MTU(X)$.

Proof 3. Since all the transactions containing an itemset X also contains any subset Y of $X, MTU(Y) \geqslant MTU(X)$. $\square$

The third challenge for mining top-k HUI 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 1 summarizes the concepts and notations we define in this paper.

4. T-HUDS: top-k high utility itemset mining over 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.

4.1. An Overview of T-HUDS Method

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, remove transactions in the oldest batch from the tree if the sliding window had been filled up, and updates the two auxiliary lists.

Algorithm 1 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 addition or changes of transactions in the sliding window. Once a new window is formed, *T-HUDS* calls Algorithm 3 to find top-k HUIs for the new sliding window.

Algorithm 1. 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 5 to update *HUDS-tree*, *maxUtilList* and *MIUList* using B_i and *winSize*
8: **if** batch ID $i \geqslant$ *winSize* **then**
9: Call Algorithm 3 to compute top-k HUIs on the current sliding window with the *HUDS-tree*, *maxUtilList*, *MIUList* and $minTopKUtil_{i-1}$
10: **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*.

4.2. HUDS-tree structure and construction

The structure of *HUDS-tree* is similar to that of *FP-tree* [14], *UP-tree* [31] or *HUS-tree* [2]. 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.

Table 1
Summary of notations.

Concept	Description
$u(i,T)$	Utility of item i in transaction T
$u(X)$	Utility of itemset X in a data set
$TWU(X)$	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 data set
$MIU(X)$	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 ith sliding window
PTKHUI	Potential Top-k High Utility Itemset
PTKSet	Set of Potential Top-k High Utility Itemsets

A non-root node in a *HUDS-tree* contains the following fields: *nodeName*, *nodeCounts*, *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 4. A *HUDS-tree*, built from the transactions in sliding window $SW_1 = \{B_1, B_2\}$ in Fig. 1, is illustrated in Fig. 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 the first batch are 0. The field *succ* is not illustrated for the clarity reason.

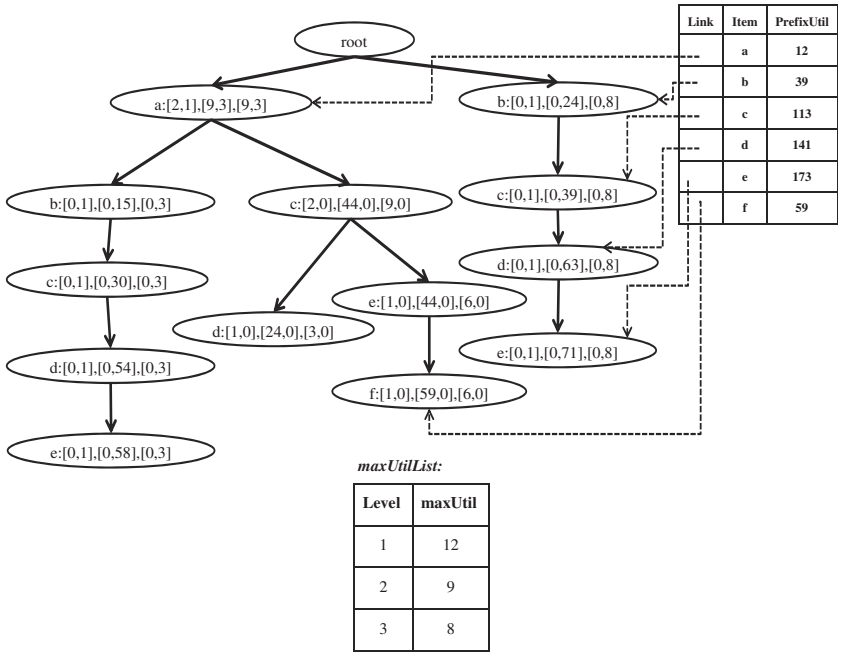


Fig. 2. HUDS-tree after inserting transaction in SW_1 in Fig. 1.

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 2. 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: update the *idx*th element, $maxUtil_{idx}$, in the *maxUtilList*
13: **if** *idx* $\neq$ the length of T **then**
14: Algorithm 2$(T, node, idx + 1, batchNumber)$
15: HUDS$-Tree \leftarrow rootNode$
16: **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 order the items in the transaction in an order (such as the lexicographic order or the descending external item utility order),[2] and then insert the items into the HUDS-tree in the way similar to building an FP-tree [14]. 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 2.[3] 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.

4.3. 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 [33]. 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

[2] In our experiments we sort the items in each transaction lexicographically. According to [16], sorting items based on their frequency in the database may give the most compact tree structure. But they also showed that using the other orders (i.e., the alphabetical order) does not affect the size of the FP-tree significantly. Since sorting the items based on either frequency or *TWU* needs to scan the database one more time to determine frequencies or *TWU* of each item in the database, we use the alphabetical order to avoid this additional scan. Given an order, items in each transaction can be sorted efficiently using one of the fastest and commonly-used sorting algorithms such as the quick sort algorithm. The average time complexity of sorting a transaction with quick sort is $O(m \log(m))$, where m is the average length of the transactions. In terms of memory usage, the extra space needed for sorting a transaction with quick sort is $O(\log(m))$ on average.

[3] In this algorithm, the way we calculate *nodePUtils* is similar to the strategy *DNU* in [29,31].

strategy for setting the threshold should satisfy the following conditions: (1) it should not miss any top-k *HUIs* and (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.[4] The fourth strategy (to be used in the second phase) will be described in Section 4.4.2.

4.3.1. 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 11. (Maximum Utility List (maxUtilList)) of a *HUDS-tree* is a list of length d:

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

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

$$maxUtil_i = \max_j\{\max(minProfit(node_{i,j}) \times nodeCount(node_{i,j}), nodeMTU(node_{i,j}))\}$$

where $node_{i,j}$ is the jth 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 Fig. 2. The level 2 has one b node and two c nodes. $maxUtil_2$ is thus computed as:

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

Lemma 4. *Let $util_k$ be the utility of the kth 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 4. 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., $LPI(X) \leqslant u(X)$ on S. Also, for all $Y \subseteq X, LPI(Y) \geqslant LPI(X)$ 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 $X_{L,j}$. According to Lemma 3, $MTU(Y) \geqslant MTU(X_{L,j})$ on set $S_{L,j}$. According to Property 2, $u(Y) \geqslant MTU(Y)$ on $S_{L,j}$. Similarly, $u(Y) \geqslant LPI(Y) \geqslant LPI(X_{L,j})$ on $S_{L,j}$. Thus,

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

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

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

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 4 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.

[4] Note that two of the three strategies, namely the use of *maxUtilList* and the use of *minTopKUtil* of the last window to be described in Sections 4.3.1 and 4.3.3 respectively, are completely novel, while the other one (i.e., the use of *MIUList* to be described in Section 4.3.2) is inspired by a strategy used in [33].

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.

4.3.2. 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 [33]:

Definition 12. 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 $\neq \exists\, T_p \in D$ such that $u(a, T_p) < u(a, T_q)$.

Definition 13. Minimum Itemset Utility of an itemset X in a dataset D is defined as:

$$MIU_D(X) = \sum_{a_i \in X} miu_D(a_i) \times SC_D(X) \text{ where } SC_D(X) \text{ is support count of } X \text{ in } D.$$

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

Property 2. *For any itemset X in dataset D, $MIU_D(X) \leqslant u_D(X)$.*

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 [33], 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 14. 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 $MIUList = \{MIU_1, MIU_2, \ldots, MIU_k\}$, where $MIU_1 \geqslant MIU_2 \ldots \geqslant MIU_k$.

Lemma 5. *Let MIU_k be the kth member of MIUList and $util_k$ be the utility of the kth highest utility itemset in the top-k HUI list. $util_k$ is no less than MIU_k.*

Proof 5. 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 2, we have:

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

According to the Definition 14, 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.

4.3.3. 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 15. Let $SW_i = \{B_i, B_{i+1}, \ldots, B_{i+winSize-1}\}$ be the ith 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 6. *Let $util_k$ be the utility of the kth highest utility itemset over sliding window SW_{i+1}, and $minTopKUtil_i$ be the minimum top-k utility of window SW_i. We have $util_k \geqslant minTopKUtil_i$.*

Proof 6. 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 $TopkHUISet_i$, $u_B(X) \leqslant u_{SW_{i+1}}(X)$. Since $minTopKUtil_i \leqslant u_B(X)$ 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 4.4.2.

4.4. 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., *PTKHUIs*) that satisfy a dynamically-changing minimum utility threshold. In the second phase, the exact utilities of the *PTKHUIs* are computed and the top-k high utility itemsets are returned.

Algorithm 3. Top-k HUI Mining

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 4 with *min_util*. The *min_util* is also dynamically updated in Algorithm 4
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** $u_{SW_i}(itemSet) \geqslant min\_util$ **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: $minTopKUtil_i \leftarrow \min\{u_{SW_i - B_i}(itemset) | itemset \in TopkHUISet\}$
14: **return** $TopkHUISet$, $minTopKUtil_i$

This 2-phase procedure is shown in Algorithm 3. At the beginning of the procedure, we initialize the minimum utility threshold, *min_util*, according to the strategies proposed in Section 4.3 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 1 if the new batch is the first one), $maxUtil_L$ is the Lth element in *maxUtilList* (where L is computed in Line 1), and MIU_k is the kth 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 4 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.

4.4.1. Phase I: Discover PTKHUIs from HUDS-tree

In Phase I, a set of potential top-k HUIs (*PTKHUIs*) is found from the *HUDS-tree*. Our objective in this phase is to find as few *PTKHUIs* 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* [14] and *HUPMS* [2]. The major differences between our Phase I procedure and the others are as follows. First, we use both *PrefixUtil* and local *TWUs* 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.

Algorithm 4. HUDS-tree mining to generate PTKHUIs (Phase I)

Input: *HUDS-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*$(t) \geqslant$ *min_util* **then**
3: Generate a potential top-k itemset: $IS \leftarrow \{t\} \cup X$
4: Add IS into the *PTKSet* set
5: **if** $MIU_{SW_i}(IS) \geqslant$ *min_util* **then**
6: Insert $MIU_{SW_i}(IS)$ into the *MIUList*
7: *min_util* $\leftarrow MIU_k$
8: *Pattern_base*$_{IS} \leftarrow$ all prefix paths of the nodes for item t with their utilities
9: Prune all items in the *Pattern_base*$_{IS}$ whose *TWU* in *Pattern_base*$_{IS}$ is less than *min_util*.
10: Construct conditional HUDS−Tree$_{IS}$ and its header table
11: **if** *HUDS-Tree*$_{IS}$ is not empty **then**
12: call Algorithm 4(*HUDS-Tree*$_{IS}$, IS, *min_util*, *MIUList*, k)
13: **return** *PTKSet*, *min_util*, *MIUList*

The pseudocode of the *HUDS-tree* mining procedure is described in Algorithm 4. 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 5–7. 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 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 4 is called recursively to discover longer *PTKHUIs* 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.

4.4.2. Phase II: Identifying top-k HUIs from PTKHUIs

HUDS-tree is a compact representation 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 3. 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_utility* 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 kth 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 1. *Given a sliding window SW_i, if X is among the top-k high utility itemsets, it is returned by Algorithm 1.*

Proof 7. 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 3:

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

  where $L = \lceil \log_2(k+1) \rceil$. According to Lemmas 4–6, $maxUtil_L \leqslant util_k, MIU_k \leqslant util_k$ and $minTopKUtil_{i-1} \leqslant util_k$. Thus, *min_util* is no larger than $util_k$.
- In Lines 5–7 of Algorithm 4, *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 5, $MIU_k \leqslant util_k$. Thus, $min\_util \leqslant util_k$.
- In Lines 11–12 in Algorithm 3, *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 4) does not miss any top-*k* HUIs. There are two places where we prune the search space in Algorithm 4.

- 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 1, $PrefixUtil(\{t\} \cup Y \cup \{i\}) \geqslant PrefixUtil(S \cup \{t\} \cup Y \cup \{i\})$ 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 $\geqslant$ *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 5. Given the transactions in Fig. 1, let $winSize = 2$ and $k = 5$. Once the first window arrives, a complete *HUDS-Tree* is constructed (Fig. 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 3, *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.

4.5. HUDS-tree update

Algorithm 5. HUDS-tree-Update

Input: *HUDS-Tree*, new batch B_i, k
Output: *HUDS-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** $\forall i\ (1 \leqslant i \leqslant winSize)\ nodePUtils[i] = 0$ **then**
7: remove the node and its subtree from the tree
8: **for** every $T \in B_i$ **do**
9: $\{HUDS\text{-}Tree, maxUtilList\} \leftarrow$ Algorithm 2(T, *HUDS-Tree*, root of *HUDS-Tree*, 1, *batchNumber*)
10: update the *miu* value of each item in T
11: 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.
12: 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.
13: **return** *HUDS-Tree*, *maxUtilList*, *MIUList*

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 5 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 2–8, 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 calling Algorithm 2. *batchNumber* is passed to Algorithm 2 so that the information about the new batch will be stored the *batchNumber*th slots in the node fields. In Algorithm 2, *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.

5. Performance evaluation

In this section, the proposed method for finding top-k high utility itemsets over data stream is evaluated. All the algorithms are implemented in Java. The experiments are conducted on an Intel (R) Core (TM) i7 2.80 GHz computer with 4 GB of RAM.

5.1. 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 [5]. 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* [11], 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 [11]. *ChainStore* is a dataset with over a million transactions, obtained from [28]. Table 2 shows details of the datasets. The *ChainStore* dataset already contains external utilities of the items and the frequency of each item in a transaction. 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 [2] 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 2 shows how many transactions are in a batch. It is set in the same way as in [2] so that each data set 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.

Table 2
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

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

5.2. 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 [33] which discovers top-*k* high utility itemsets from a static data set based on the UP-Growth method [31]. 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 [33]. 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 data set 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 [2], 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* [33]. 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 [31] 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 $T\text{-}HUDS_I$, 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. Effectiveness of the obtained threshold

Fig. 3 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 $T\text{-}HUDS_I$ 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 itemsets.

5.4. Number of generated candidates

In addition to the obtained threshold, the number of generated candidates (i.e., *PTKHUIs*) at the end of the first phase is another metric to assess the effectiveness of HUI mining methods. Table 3 presents the numbers of generated candidates on

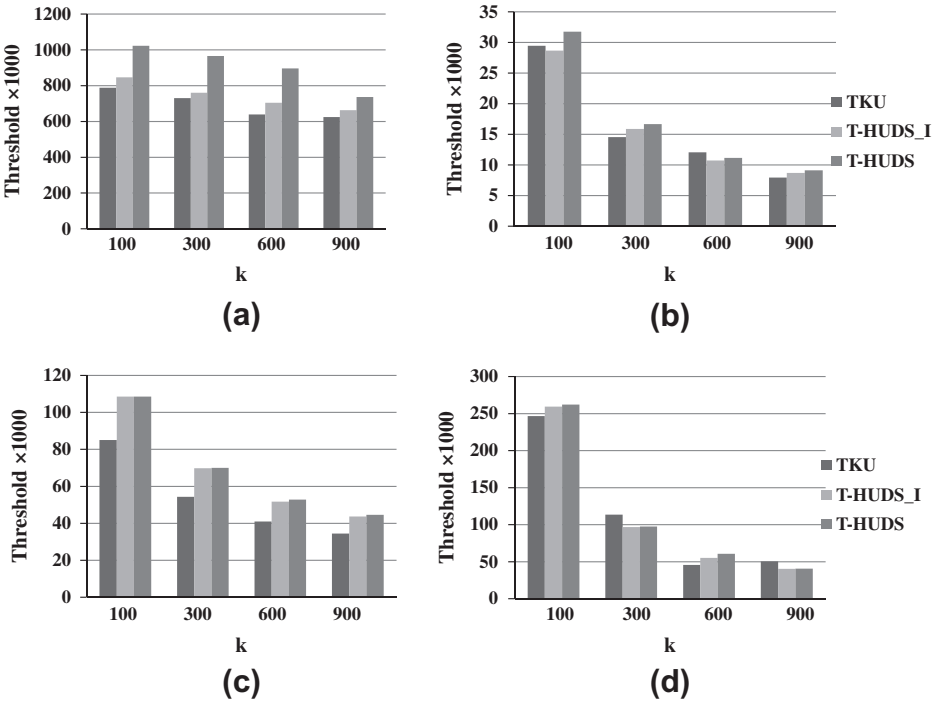


Fig. 3. Reached threshold on (a) Connect-4, (b) IBM, (c) BMS-POS, and (d) ChainStore datasets.

Table 3
Number of candidates generated in phase I.

Dataset	k	TKU	T-HUDS	$HUPMS_T$
Connec-4	100	2,280,595	1,189,624	657,934
	300	2,587,463	1,258,241	717,934
	600	2,865,490	1,315,869	857,934
	900	3,069,445	1,472,473	1,007,934
IBM	100	103,485	69,959	22,038
	300	135,998	84,898	26,668
	600	198,671	94,850	54,969
	900	276,668	100,875	217,874
BMS-POS	100	45,054	35,697	31,407
	300	59,357	37,251	35,682
	600	119,479	47,215	42,112
	900	177,725	50,189	51,463
ChainStore	100	40,419	19,751	101,435
	300	140,236	32,213	152,451
	600	258,318	102,385	211,627
	900	371,408	227,826	282,074

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 experiments 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 $T\text{-}HUDS_I$ 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 data sets. 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 data set (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*.

5.5. Efficiency of T-HUDS: run time

Fig. 4 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 $T\text{-}HUDS_I$ and *T-HUDS*. On *Connect-4* and *ChainStore*, $T\text{-}HUDS_I$ and *T-HUDS* are significantly faster than both $HUPMS_T$ and *TKU*. On the largest data set (*ChainStore*) and the most densest data set (*Connect-4*), $HUPMS_T$ is the worst, even much worse than *TKU*. The run time for $T\text{-}HUDS_I$ 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 *T-HUDS* significantly outperforms other methods in both large and dense datasets.

To see how each method works in different phases, Figs. 5 and 6 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 $T\text{-}HUDS_I$.

5.6. 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 4 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 *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.

5.7. 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 describes three different versions of the proposed method. The first method does not use *maxUtillist* values to set the

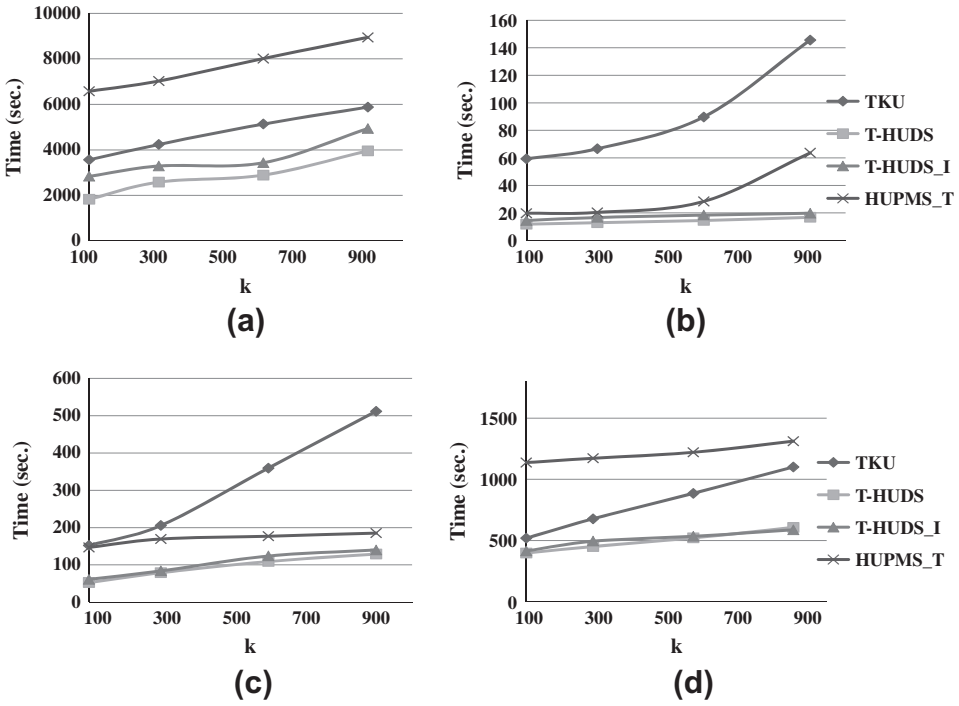


Fig. 4. Run time on (a) Connect-4, (b) IBM, (c) BMS-POS, and (d) ChainStore datasets.

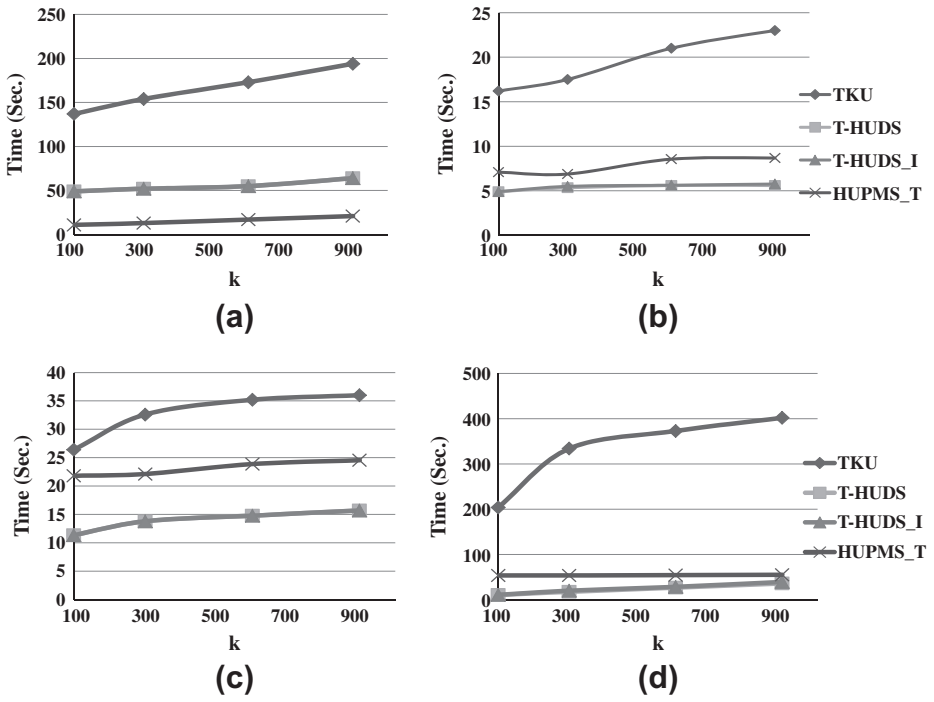


Fig. 5. Run Time for Phase I: (a) Connect-4, (b) IBM, (c) BMS-POS, and (d) ChainStore.

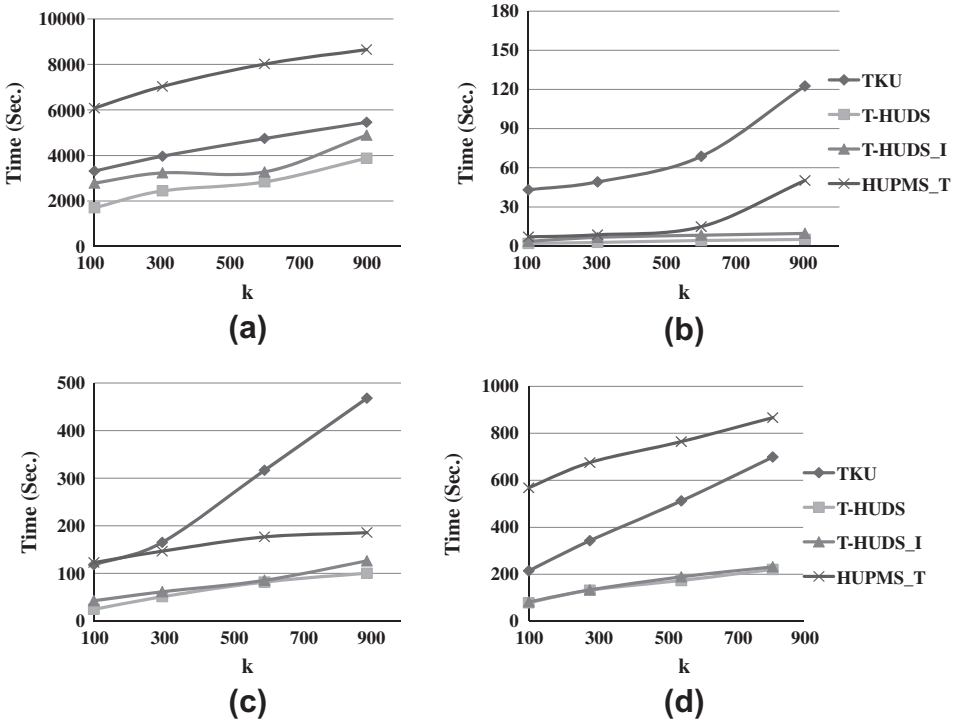


Fig. 6. Run Time for Phase II: (a) Connect-4, (b) IBM, (c) BMS-POS, and (d) ChainStore.

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 6 and Fig. 7 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

Table 4
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

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*.

5.8. 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 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 [31]. 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 [31] and *HUPMS* uses *TWU* to prune the search space. Hence, a comparison between these methods will illustrate the impact of *PrefixUtil*.

Figs. 8 and 9 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 data set and the number of potential 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 over-estimate of the true utility.

Table 5
Methods with different strategies.

Method	*maxUtilList*	*MIUList*	*minTopKUtil*
T-HUDS$_1$	×	✓	✓
T-HUDS$_2$	✓	×	✓
T-HUDS$_3$	✓	✓	×

Table 6
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	89,804	77,968	224,262
	300	110,487	87,108	365,500
	600		95,988	394,716
	900	116,889	107,163	408,102
BMS-POS	100	44,117	39,075	111,172
	300	51,201	49,870	104,019
	600	63,544	61,962	120,607
	900	80,827	80,469	149,012
ChainStore	100	24,409	21,620	61,511
	300	44,276	43,125	89,950
	600	137,794	134,363	261,534
	900	366,902	365,277	676,419

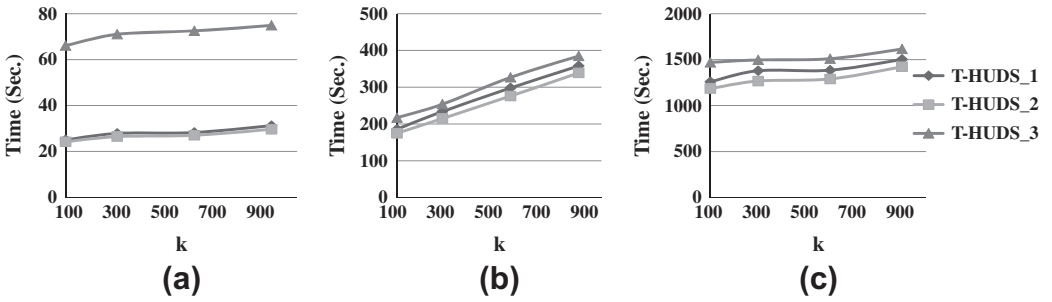


Fig. 7. Run time for different versions of T-HUDS: (a) IBM, (b) BMS-POS, and (c) ChainStore datasets.

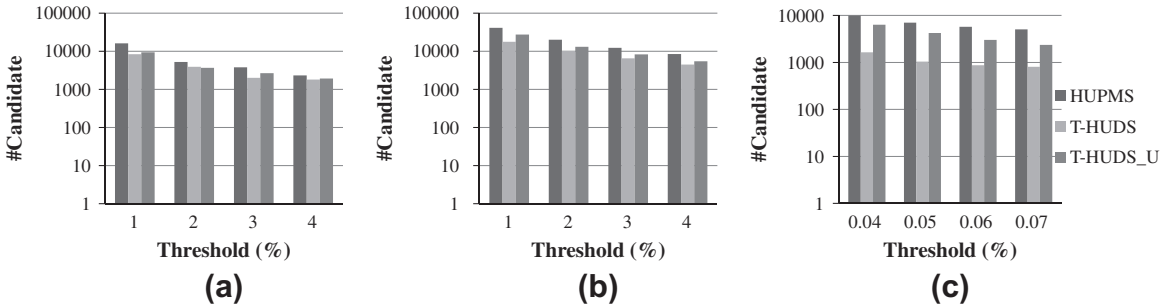


Fig. 8. Impact of *PrefixUtil* on the number of generated candidates on (a) IBM, (b) BMS-POS, and (c) ChainStore datasets.

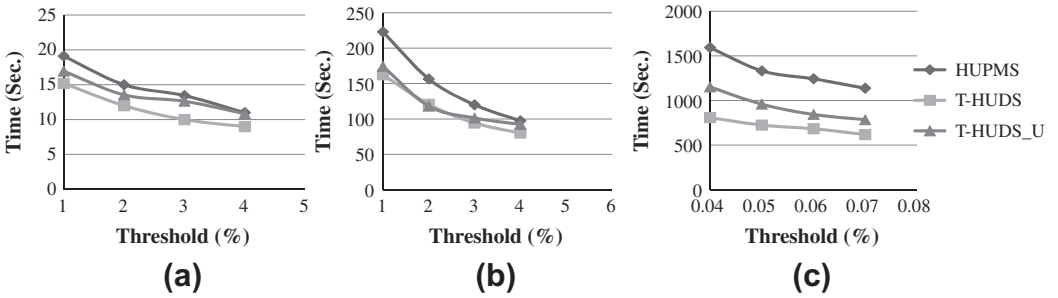


Fig. 9. Impact of *PrefixUtil* on run time on (a) IBM, (b) BMS-POS, and (c) ChainStore datasets.

5.9. 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 *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. Fig. 10 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.

5.10. 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. Fig. 11 illustrates

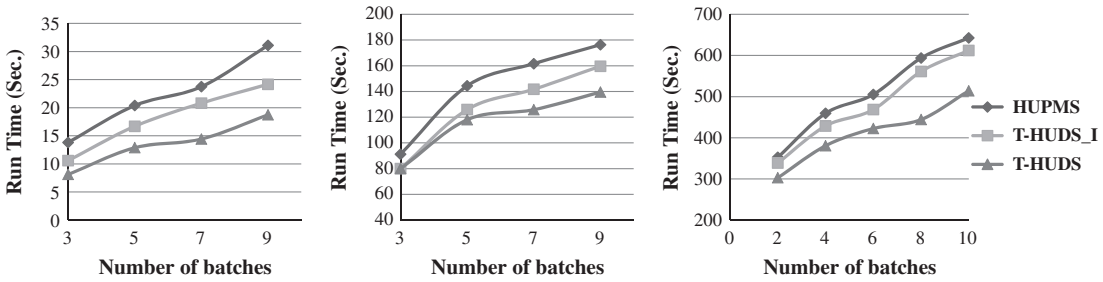


Fig. 10. Effect of the window size on the run time: (a) IBM, (b) BMS-POS, and (c) ChainStore datasets.

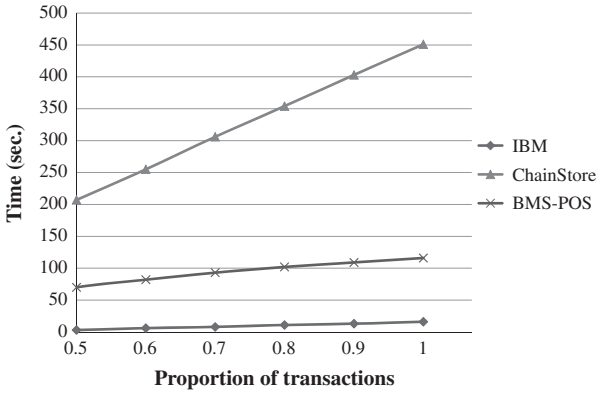


Fig. 11. Scalability of T-HUDS on different datasets (k = 500).

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.

6. Related work

6.1. High utility pattern mining

The traditional frequent pattern mining considers equal profit/weight for all items and binary occurrences (0/1) of items in transactions. To address these limitations, HUI mining methods were proposed to address these limitations. The *MEU* (Mining with Expected Utility) model [36] is the first high utility itemset mining method. *MEU* cannot use the downward closure property, hence a heuristic approach was proposed to predict if an itemset should be added to the candidate set. *MEU* checks the candidate itemsets using a prediction method with a high computational cost. Later, the *UMining* algorithm [35] improved its performance. They defined an upper bound utility for each itemset. Using this upper bound low utility itemsets are pruned during the mining process. However, these methods still do not use any downward closure property, and therefore, they suffer from excessive candidate generations and high computational costs.

The *Two-Phase* method presented in [25,26] used 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 of the high *TWU* itemsets (*HTWU*). 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* [10] was proposed that is more efficient than *Two-phase* in dense databases when the minimum utility threshold is very low. This method constructs a memory-based *CUP-tree* for mining HUIs. To reduce the number of candidates in each data base scan, the isolated item discarding strategy (*IIDS*) was proposed in [22]. Applying *IIDS*, the authors proposed two efficient algorithms *FUM* and *DCG+*. In [3], efficient tree structures were proposed to discover high utility itemsets in incremental databases. They first construct a tree based on transactions and then apply a pattern growth approach to generate potential candidates. Each node in the tree consists of an item and *TWU* value. After candidate generation in the second scan, all HUIs are discovered. The authors in [18] proposed tighter upper bounds of utility values than *TWU* values. The proposed method uses the projection technique to reduce the TWU value of an itemset. This method works based on *TWU*. But its difference from the original *TWU* estimate is that in the recursive mining process the alphabetical order is applied to the processing order of *HTWU* itemsets. That is, when itemset X is processed, only

the items appearing in the set of itemsets with X as their prefix can be kept in the projected transactions for X. The over-estimated utility of itemset X is the *TWU* value of itemset X computed on the projected transactions for X. This leads to a lower over-estimate utility than the *TWU* value on the whole dataset. However, this method needs several scans of data, which is not suitable for mining data streams.

In [31] a pattern growth approach (i.e., UP-Growth) was proposed to discover high utility patterns in two scans. UP-Growth defines a tree structure and four effective strategies *DGU*, *DGN*, *DLU* and *DLN* for mining HUIs. The strategies help the method to prune more candidates during the mining process. Similar to the other two-phase approaches in the first phase it generates candidates and in the second phase it discovers HUIs. Experiments showed that the number of candidates generated by UP-Growth in phase I can be much smaller than that from the other methods. The improved version of [31] was presented in [29]. The proposed framework consists of three main steps. First of all, it scans database twice to construct *UP-Tree* applying two proposed strategies: *DGU* and *DGN*. In the second step, *Up-Growth* or another 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. *Up-Growth* works based on two proposed strategies *DLN* and *DLU*. These strategies are the same as the strategies in [31]. But the second method, Up-Growth$^+$ works based on two new strategies *DNU* and *DNN*. However, these algorithms are neither applicable to high utility itemset mining over data streams nor are able to discover top-k high utility itemsets directly.

Recently, some works have focused on mining HUIs in one scan of data base. *HUI-Miner* (High Utility Itemset Miner) proposed by [24] 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 itemsets. 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 data set several times, *HUI-Miner* scans the data once and after that it just scans *Utility-list* several times. In [17] a high utility itemset approach is proposed that works in a single phase without generating candidates. They propose a novel data structure to maintain original utility information during mining process and then enumerate itemsets by prefix extensions. Based on the proposed data structure they could compute a tighter bound than *TWU* to prune low utility itemsets and to directly identify high utility itemsets. It is worth pointing out the proposed upper bound (the *utility of full prefix extension* (u_{fpe})) in [17] is different from *PrefixUtil* proposed in this paper. u_{fpe} of an itemset X in a transaction T is the sum of the utilities of the items in the *fpe* of itemset X. Considering the *fpe* definition in [17], an itemset Y is *fpe* of X in a transaction T containing X, if Y is a prefix extension of X derived by adding exactly all the items in T that are listed **before** X, while *PrefixUtil* considers all items in transaction T that are not ranked **after** any item in X. Although these approaches are able to discover HUIs using one scan of the data set, not only are their data structures not efficient for mining data streams, but also they are not designed to discover top-k HUIs.

6.2. High utility mining in data streams

Although several algorithms have been proposed for mining frequent patterns over data streams [7,19,37], these algorithms are not applicable to *HUI* mining. Recently, high utility itemset mining from data stream has become an active research topic in data mining [30,25,2].

THUI-Mine [30] was the first algorithm for mining temporal high utility itemsets from data streams. It is based on a non-stream HUI mining algorithm proposed in [25]. *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. However, this algorithm generates a huge number of candidates. Later, two algorithms, called *MHUI-BIT* and *MHUI-TID*, were proposed in [20] for mining high utility itemsets from data streams. This paper proposed two effective representations of item information (Bitvector and TIDlist representations) and an extended lexicographical tree-based summary data structure. The authors showed that *MHUI-TID* outperforms *MHUI-BIT*. However, the proposed representations become very inefficient when the number of distinct items become large in a window. During the mining process, they have used a tree structure to 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. Hence it needs to scan database several times. *GUIDE* is a framework proposed in [4] that mines a compact form of high utility patterns 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 during one scan of the database. 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. Then, the proposed pruning strategies are applied onto the tree to decrease the memory usage of the tree. Finally, the high utility patterns are discovered. However, *GUIDE* discovers *temporal maximal high utility itemsets* and is also not able to find top-k HUIs. *HUPMS* [2] is the recent method for HUI mining from data streams, which is based on the *TWU* model. Similar to other works, they propose a novel tree structure, called *HUS-Tree* (high utility stream tree) to keep information about itemsets and their *TWU* values. The candidates are generated in regard to the input threshold. During the second scan of the database, the exact utility of each candidate is calculated and HUIs are discovered. However, the above mentioned methods were not designed for finding top-k high utility itemsets over data streams.

6.3. Top-k pattern mining

The top-*k* high utility itemset mining was first introduced in [6]. 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 [33] to find top-*k* high utility itemsets over a static data set. The proposed approach mines top-*k* high utility itemsets without setting the minimum utility threshold. It works based on *Up-Growth* [31]. 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 different windows, 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 this paper, we designed better strategies for initializing and dynamically adjusting the minimum utility threshold over data streams.

In frequent itemsets mining, several methods were proposed to find top-*k* frequent itemsets in static data sets [8,9,15,27]. Although these algorithms are efficient, it is difficult (if not impossible) to simply adapt them to HUI mining. There are several methods for finding top-*k* frequent itemsets over data streams. Golab et al. [12] 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 [32] proposed two algorithms to address the problem of top-*k* frequent *l*-itemsets ($1 \leqslant l \leqslant L$) mining over data streams. *TOPSIL-Miner* [34] 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. A major difficulty in top-*k* HUI mining is that the utility of an itemset does not have the downward closure property. Thus, HUI mining has to work with estimated utilities. The strategies proposed for raising the frequency threshold in top-*k* frequent itemset mining cannot be applied to estimated utilities.

7. Conclusion

In this paper, we proposed an efficient algorithm, *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 prove 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. Extensive experiments were conducted to confirm the effectiveness and the high efficiency of the algorithm in finding top-*k* HUIs over data streams.

While our method proves to be efficient in both run time and memory consumption, there is room for further research and improvement. Similar to the tree structures used in *HUPMS* [2] and *TKU* [33], the *HUDS-tree* is a lossy compression of the transactions in the sliding window. The consequence of this is that a second scan of data in the sliding window is needed in the second phase of the method to obtain the exact utilities of the potential top-*k* HUIs. Although a sliding window is generally small enough to fit into the main memory, reducing the number of data scans can further improve the run time performance of HUI mining. We will look into two directions: one is to design a lossless compression data structure to store the information needed to compute the exact utility, and the other is to design an approximation method that returns an approximated list of top-*k* patterns from a lossy compression of the data.

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases, 1994, pp. 487–499.
[2] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, Interactive mining of high utility patterns over data streams, Expert Syst. Appl. 39 (2012) 11979–11991.
[3] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, Y.K. Lee, Efficient tree structures for high-utility pattern mining in incremental databases, IEEE Trans. Knowl. Data Eng. 21 (2009) 1708–1721.
[4] V.S.T.B.E. Shie, P.S. Yu, Efficient algorithms for mining maximal high utility itemsets from data streams with different models, Expert Syst. Appl. 39 (2012) 12947–12960.
[5] K. Bache, M. Lichman, UCI Machine Learning Repository, 2013.
[6] R. Chan, Q. Yang, Y. Shen, Mining high-utility itemsets, in: Proc. of Third IEEE Int'l Conf. on Data Mining, 2003, pp. 19–26.
[7] J. Cheng, Y. Ke, W. Ng, A survey on algorithms for mining frequent itemsets over data streams, Knowl. Inform. Syst. 16 (2008) 1–27.
[8] Y.L. Cheung, A.W. Fu, Mining frequent itemsets without support threshold: with and without item constraints, IEEE Trans. Knowl. Data Eng. 16 (2004) 1052–1069.
[9] K. Chuang, J. Huang, M. Chen, Mining top-*k* frequent patterns in the presence of the memory constraint, VLDB J. 17 (2008) 1321–1344.
[10] A. Erwin, R.P. Gopalan, 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, 2007, pp. 3–11.
[11] B. Goethals, M.J. Zaki., Frequent Itemset Mining Dataset Repository, 2004. <http://fimi.cs.helsinki.fi/data/>.
[12] L. Golab, D. Dehaan, E. Demaine, Identifying frequent items in sliding windows over on-line packet streams, in: Proceedings of ACM SIGCOMM Internet Measurement Conference, 2003, pp. 173–178.

[13] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data Min. Knowl. Discov. 15 (2007) 55–86.
[14] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, SIGMOD Rec. 29 (2000) 1–12.
[15] Y. Hirate, E. Iwahashi, 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 Discovery, 2004.
[16] Y.Y.J. Han, J. Pei, R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, Data Min. Knowl. Discov. 8 (2004) 53–87.
[17] L. Junqiang, W. Ke, B. Fung, Direct discovery of high utility itemsets without candidate generation, in: 12th IEEE International Conference on Data Mining (ICDM), 2012, pp. 984–989.
[18] G.C. Lan, T.P. Hong, V.S. Tseng, Tightening upper bounds of utility values in utility mining, in: Proceedings of the 28th workshop on Combinatorial Mathematics and Computation Theory, 2011, pp. 11–16.
[19] K.S.C. Leung, F. Jiang, Frequent itemset mining of uncertain data streams using the damped window model, in: Proceedings of the 2011 ACM Symposium on Applied Computing, 2011, pp. 950–955.
[20] H.F. Li, H.Y. Huang, Y.C. Chen, Y.J. Liu, 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, 2008a, pp. 881–886.
[21] H.F. Li, H.Y. Huang, S.Y. Lee, Fast and memory efficient mining of high-utility itemsets from data streams: with and without negative item profits, Knowl. Inform. Syst. 28 (2011) 495–522.
[22] Y.C. Li, J.S. Yeh, C.C. Chang, Isolated items discarding strategy for discovering high utility itemsets, Data Knowl. Eng. 64 (2008) 198–217.
[23] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, 2012a, pp. 55–64.
[24] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, 2012b, pp. 55–64.
[25] Y. Liu, W. k. Liao, A. Choudhary, A fast high utility itemsets mining algorithm, in: Proceedings of the 1st International Workshop on Utility-based Data Mining, 2005a pp. 90–99.
[26] Y. Liu, W. Liao, 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, 2005b, pp. 689–695.
[27] S. Ngan, T. Lam, R.C. Wong, A.W. Fu, Mining n-most interesting itemsets without support threshold by the cofi-tree, Int. J. Bus. Intell. Data Min. 1 (2005) 88–106.
[28] J. Pisharath, Y. Liu, B. Ozisikyilmaz, R. Narayanan, W.K. Liao, A. Choudhary, G. Memik, Nu-minebench Version 2.0 Dataset and Technical Report, 2012. <http://cucis.ece.northwestern.edu/projects/dms/minebench.html>.
[29] V. Tseng, B. Shie, C.W. Wu, P. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Trans. Knowl. Data Eng. 25 (2013) 1772–1786.
[30] V.S. Tseng, C.J. Chu, T. Liang, Efficient mining of temporal high-utility itemsets from data streams, in: ACM KDD Utility Based Data Mining, 2006, pp. 18–27.
[31] V.S. Tseng, C.W. Wu, B.E. Shie, P.S. Yu, Up-growth: an efficient algorithm for high utility itemset mining, in: Proc. of Int'l Conf. on ACM SIGKDD, 2010, pp. 253–262.
[32] R.C.W. Wong, A.W.C. Fu, Mining top-*k* frequent itemsets from data streams, Data Min. Knowl. Discov. 13 (2006) 193–217.
[33] C.W. Wu, B.E. Shie, V.S. Tseng, P.S. Yu, Mining top-*k* high utility itemsets, in: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2012, pp. 78–86.
[34] B. Yang, H. Huang, Topsil-miner: an efficient algorithm for mining top-*k* significant itemsets over data streams, Data Min. Knowl. Discov. 23 (2010) 225–242.
[35] H. Yao, H.J. Hamilton, Mining itemset utilities from transaction databases, Data Knowl. Eng. 59 (2006) 603–626.
[36] H. Yao, H.J. Hamilton, C.J. Butz, A foundational approach to mining itemset utilities from database, in: Proceeding of the 4th SIAM International Conference on Data Mining, 2004, pp. 482–491.
[37] S.J. Yen, Y.S. Lee, C.W. Wu, C.L. Lin, An efficient algorithm for maintaining frequent closed itemsets over data stream, in: Proceedings of IEA/AIE, 2009, pp. 767–776.