Knowl Inf Syst (2011) 28:495–522
DOI 10.1007/s10115-010-0330-z

REGULAR PAPER

Fast and memory efficient mining of high-utility itemsets from data streams: with and without negative item profits

Hua-Fu Li · Hsin-Yun Huang · Suh-Yin Lee

Received: 29 November 2009 / Revised: 11 May 2010 / Accepted: 9 July 2010 /
Published online: 24 July 2010

Abstract Mining utility itemsets from data steams is one of the most interesting research issues in data mining and knowledge discovery. In this paper, two efficient sliding window-based algorithms, MHUI-BIT (Mining High-Utility Itemsets based on BITvector) and MHUI-TID (Mining High-Utility Itemsets based on TIDlist), are proposed for mining high-utility itemsets from data streams. Based on the sliding window-based framework of the proposed approaches, two effective representations of item information, Bitvector and TIDlist, and a lexicographical tree-based summary data structure, LexTree-2HTU, are developed to improve the efficiency of discovering high-utility itemsets with positive profits from data streams. Experimental results show that the proposed algorithms outperform than the existing approaches for discovering high-utility itemsets from data streams over sliding windows. Beside, we also propose the adapted approaches of algorithms MHUI-BIT and MHUI-TID in order to handle the case when we are interested in mining utility itemsets with negative item profits. Experiments show that the variants of algorithms MHUI-BIT and MHUI-TID are efficient approaches for mining high-utility itemsets with negative item profits over stream transaction-sensitive sliding windows.

Keywords Data mining · Data streams · Utility mining · High-utility itemsets · Utility itemset with positive item profits · Utility itemset with negative item profits

H.-F. Li (✉)
Department of Information Management, Kainan University, Taoyuan, Taiwan
e-mail: hfli@mail.knu.edu.tw; lihuafu@gmail.com

H.-Y. Huang · S.-Y. Lee
Department of Computer Science, National Chiao-Tung University,
Hsinchu, Taiwan
e-mail: caroline27215@gmail.com

S.-Y. Lee
e-mail: sylee@cs.nctu.edu.tw

1 Introduction

In recent years, database and knowledge discovery communities have focused on mining frequent patterns from data streams. A *data stream* is an infinite sequence of data elements continuously arrived at a rapid rate. Many real-world applications generate data streams in real time such as online transaction flows from a large e-commence site, sensor data generated from sensor networks, call record flows from telecommunication company, Web record and click-streams in Web applications, performance measurement in network monitoring and traffic management, etc.

Based on the unique characteristics of data streams, methods for mining such streaming data and for mining traditional datasets are different in following aspects [10,26]. First, data elements of data streams continuously arrive in a rapid rate and the amount of data is huge. That means that the first requirement of proposed algorithms for mining data streams is real time. Second, once a data element is removed from the main memory, it is unable to backtrack over previously arrived data elements. Hence, the second requirement of stream mining algorithms is one-pass scan of data streams. Third, the memory requirement of storing all streaming data is unlimited because data stream is an infinite sequence of data element. Therefore, the proposed stream mining algorithms use limited memory usage to record the essential information about the unbounded data streams. Accordingly, the basic requirements of stream mining algorithms are single-pass, bounded space, and real time. Consequently, several efficient algorithms are proposed for mining various interesting patterns from data streams in recent years [5–8,12,14–16,19,22,23].

Mining frequent itemsets from large databases has been widely studied over the last decade [1–3,11,13,20,21]. However, traditional frequent itemsets mining model treats all items in a large database by only considering whether an item appeared in a transaction or not. However, the *count* of an itemset may be not a sufficient indicator of interestingness. Because of the count of an itemset only reflects the number of transactions in a large database that contains the itemset. It does not reveal the *utility* of an itemset. The utility can be measured in terms of cost, profit or other expressions of user preferences. Furthermore, frequent itemsets may only contribute a small portion of the overall profit, whereas non-frequent itemsets may contribute a large portion of profit. Consequently, a new model of frequent itemsets mining, i.e., *utility mining model* [4], is proposed to address the limitation of traditional association rule mining. Based on the utility mining model, utility is a measure of how useful or profitable an itemset X is. For example, in the table of Fig. 1a, each transaction is composed of items with

TID	Items with quantity (i:q)
T_1	(c:26), (e:1)
T_2	(b:6), (d:1)
T_3	(a:12), (d:1)
T_4	(b:1), (d:7)
T_5	(c:12), (e:2)
T_6	(a:1), (b:4), (e:1)
T_7	(b:10), (e:1)
T_8	(a:1), (b:1), (c:1), (d:3), (e:1)
T_9	(a:2), (b:1), (c:27), (e:2)
T_{10}	(b:6), (c:2)
T_{11}	(b:3), (d:2)
T_{12}	(b:2), (c:1)

A data stream is formed by transactions arriving in series

(a) An Example Data Stream

Item	Profit (per unit)
a	3
b	10
c	1
d	6
e	5

(b) An Example Utility Table

Fig. 1 An example data stream and an example utility table

quantity, such as, $T_1 = \{(c : 26), (e : 1)\}$, and the utility (profit) of each item is listed in the table of Fig. 1b. The utility of an itemset X, i.e., $u(X)$, is the sum of the utilities of itemset X in all the transactions containing X. An itemset X is called a *high-utility itemset* if and only if $u(X) \geq min\_utility$, where *min_utility* is a user-defined minimum utility threshold. Otherwise, it is a *low utility itemset*. For example, $u(bd) = u(bd, T_2) + u(bd, T_4) + u(bd, T_8) = (10 \times 6 + 6 \times 1) + (10 \times 1 + 6 \times 7) + (10 \times 1 + 6 \times 3) = 66 + 52 + 28 = 146$. The 2-itemset ($bd$) is considered as a high-utility itemset if the minimum utility threshold *min_utility* is 120. Therefore, the goal of utility mining is to find a set of high-utility itemsets from a large transaction database.

A utility itemset may consist of some low utility items. For example, $u(d) = u(d, T_2) + u(d, T_3) + u(d, T_4) + u(d, T_8) = (6 \times 1) + (6 \times 1) + (6 \times 7) + (6 \times 3) = 72 < min\_utility$ as shown in Fig. 1. The 1-itemset (d) is a low utility itemset. However, its superset 2-itemset (bd) is a high-utility itemset. That means if we use level-wise based algorithms (Apriori-based searching schema) to discover high-utility itemsets, all combinational of the items must be generated. The number of candidate is extremely large and the cost of computation time and memory requirement will be intolerable. The challenge of mining utility itemsets is not only in restricting the size of the generated candidate utility itemsets but also in simplify the computation time of calculating the utility of itemsets [8].

In this paper, we explore the issue of efficient mining of high- utility itemsets from data streams with and without negative profits. Hence, another challenge of utility mining from data streams is how to discover high-utility itemsets from streaming data as time advances. Two item information representations, i.e., Bitvector and TIDlist (transaction identifier list), are used in the proposed algorithms, MHUI-BIT (Mining High Utility Itemsets based on BITvector) and MHUI-TID (Mining High Utility Itemsets based on TIDlist), to improve the performance of utility mining. Both item information representations can be used to generate utility itemsets from current sliding window without rescanning the data streams. Furthermore, an effective summary data structure LexTree-2HTU is developed for maintaining a set 2-HTU (high transaction-weighted utilization)-itemsets from current transaction-sensitive sliding window. Based on two item information representations and LexTree-2HTU, two algorithms, MHUI-BIT and MHUI-TID, are proposed to mine a set of high-utility itemsets with positive item profits from data streams efficiently and effectively. Besides, a new issue of utility mining with negative item profits is addressed and two adapted approaches of algorithms MHUI-BIT and MHUI-TID are developed to discover high-utility itemsets with negative item profits from data streams.

1.1 Related works

Recently, mining high-utility itemsets from a large transaction database has become an active research problem of data mining [9,17,18,24,25]. We describe briefly as follows. A formal definition of utility mining, a theoretical model, called MEU, and a unified framework were proposed by Yao et al. [24,25]. In this work, the utility is defined as the combination of utility information in each transaction and additional resources. Since the model cannot use the downward closure property to reduce the number of candidate itemsets, a heuristic approach was proposed to predict whether an itemset should be added to the candidate set or not. However, the prediction usually overestimates, especially at the beginning stages. Moreover, the examination of candidates is impractical, either in processing cost or in memory requirement. An efficient utility mining algorithm, called Two-Phase, was proposed by Liu et al. [18]. The basic idea of Two-Phase algorithm is based on the MEU. But Two-Phase algorithm not only can prune down the number of candidate itemsets but also can find a complete

set of high-utility itemsets. In first phase of Two-Phase algorithm, a useful property, i.e., *transaction-weighted downward closure property*, is used in the Two-Phase algorithm. The size of candidates is reduced by considering the supersets of high transaction-weighted utilization itemsets. In second phase, only one extra database scan is needed to filter out the high transaction-weighted utilization itemsets that are indeed low utility itemsets. However, Two-Phase algorithm is a two-pass algorithm for mining high-utility itemsets. It is not suitable for mining data streams. Li et al. [17] proposed an isolated items discarding strategy (IIDS), which can be applied to existing level-wise utility mining methods to reduce candidates and to improve performance of utility itemset mining. Recently, Chu et al. [9] proposed efficient approach HUINIV-Mine for mining high- utility itemsets with negative item values from large databases.

The applications of mining high-utility itemsets are changed from mining traditional data sets to mining transactional data streams in recent years. Chu et al. [8] proposed the first method, called THUI-Mine, for mining temporal high-utility itemsets over data streams. The incremental mining process of THUI-Mine is based on algorithms Two-Phase [18] and SWF [13]. In the framework of THUI-Mine algorithm, a database is divided into a sequence of partitions. In the first scan of database, it employs a filtering threshold in each partition to generate a progressive transaction-weighted utilization set of itemsets. Then, it uses database reduction method to generate a set of candidate k-itemsets, where $k \geq 2$. Finally, it needs one more scan over the database to find a set of high-utility itemsets from these candidate k-itemsets. There are two performance problems of THUI-Mine algorithm: more false candidate itemsets and more memory requirement. Therefore, in this paper, we propose fast and memory efficient algorithms, MHUI-BIT and MHUI-TID, for mining high-utility itemsets with positive profits over data streams. Experimental results of this work show that the proposed algorithms MHUI-BIT and MHUI-TID both outperform than THUI-Mine algorithm.

In addition, retail business companies may sale item with negative profit for the promotion of new products. For example, customers may buy some specific products and then receive free items or goods with special discounts. At this scenario, free items or goods with special discounts result in negative item profits for business earnings. However, retail or e-commerce companies may obtain higher market profits from other products that are cross-promoted with these free items or goods with special discounts. As a result, the problem of mining high-utility itemsets with negative item profits has become too important not to address immediately. Consequently, we also propose the adapted approaches of algorithms MHUI-BIT and MHUI-TID in order to handle the case when we are interested in mining high-utility itemsets with negative item profits. Experiments show that both variants of MHUI-BIT and MHUI-TID are efficient data mining algorithm for discovering utility itemsets with negative profits from data streams over sliding windows.

1.2 Our contributions

The contributions of this work are summarized as follows.

- We proposed two efficient algorithms, MHUI-BIT (Mining High Utility Itemsets based on BITvector) and MHUI-TID (Mining High Utility Itemsets based on TIDlist), for mining high- utility itemsets from data streams with and without negative profits.
- Two item information representations, i.e., Bitvector and TIDlist (transaction identifier list), are used in the proposed methods to improve the performance of utility mining. Both item information representations can be used to generate utility itemsets from current sliding window without rescanning the data streams.

- An effective summary data structure LexTree-2HTU is developed for maintaining a set 2-HTU (high transaction-weighted utilization)-itemsets from current transaction-sensitive sliding window.
- New research issue of utility mining with negative item profits is addressed and two adapted algorithms MHUI-BIT-NIP (MHUI-BIT with Negative Item Profits) and MHUI-TID-NIP (MHUI-TID with Negative Item Profits) are developed to discover high-utility itemsets with negative item profits over continuous stream transaction-sensitive sliding windows.

1.3 Roadmap

The remainder of this paper is organized as follows. Problem of mining high-utility itemsets with and without negative item profits from data streams is defined in Sect. 2. The proposed utility mining algorithms, MHUI-BIT and MHUI-TID, and the variants proposed for utility mining with negative item profits are described in Sect. 3. Section 4 discusses the experimental results of the proposed algorithms. Finally, we conclude the work in Sect. 5.

2 Preliminary

To facilitate the presentation of this paper, some preliminaries are given in this section. Problems of mining high-utility itemsets with and without negative item profits over data stream sliding windows are defined in Sect. 2.1. An effective downward closure property used in our proposed algorithms is described in Sect. 2.2.

2.1 Problem definition

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n distinct literals called *items*. An **itemset** is a non-empty set of items. An itemset $X = (i_1, i_2, \ldots, i_k)$ with k items is referred to as k-itemset, and the value$k$ is called the **length** of X, and $i_j \in I$ for $j = 1, \ldots, n$. A **transaction** $T = \langle \text{TID}, (i_1, i_2, \ldots, i_k) \rangle$ consists of a transaction identifier (TID) and a set of items $(i_1, i_2, \ldots, i_k)$, where $i_j \in I$, $j = 1, 2, \ldots, k$. A **data stream** $DS = \{T_1, T_2, \ldots, T_m\}$ is an infinite sequence of transactions, where m is the TID of latest incoming transaction. The **transaction-sensitive sliding window** (*TransSW*) of DS is a window that slides forward for every transaction. The window at each slide has a fixed number, w, of transactions, and w is the **size** of the TransSW. Hence, *current transaction-sensitive sliding window* **CurTransSW** (or $\mathbf{TransSW}_{N\acute{a}1Dw+1}$) $= [T_{N\acute{a}1Dw+1}, T_{N\acute{a}1Dw+2}, \ldots, T_N]$, where the index $(N\acute{a}1Dw + 1)$ is the window identifier of current TransSW.

Example 1 A data stream *DS*, as shown in Fig. 1, is composed of four consecutive TransSWs, i.e.,$TransSW_1 = [T_1, T_2, \ldots, T_9]$, $TransSW_2 = [T_2, T_3, \ldots, T_{10}]$, $TransSW_3 = [T_3, T_4, \ldots, T_{11}]$ and $TransSW_4 = [T_4, T_5, \ldots, T_{12}]$, when the window size, i.e., $w = 9$, is given.

In the typical framework of frequent itemsets mining, the quantity of item purchased of each transaction is 1 or 0. However, in the framework of high-utility itemset mining, the quantity of item purchased is an arbitrary number. The *quantity* is called **local transaction utility** (LTU). The LTU, denoted as $\boldsymbol{o(i_p, T_q)}$, represents the quantity of item i_p in the transaction T_q. For example, $o(a, T_3)$ is 12 and $o(c, T_1)$ is 26 in Fig. 1.

The **external utility (EU)**, i.e., *profit*, of an item i_p, denoted as $\boldsymbol{u(i_p)}$ is the value associated with item i_p in the utility table. For example, $u(a) = 3$, $u(b) = 10$ and $u(c) = 1$. If the external utility of an item i_p is a positive value, i.e., $\boldsymbol{u(i_p)} > 0$, the item has *positive item profit*. Otherwise, the itemi_p has *negative item profit*, if $\boldsymbol{u(i_p)} \leq 0$.

The **utility of an item i_p in transaction** T_q, denoted as $\boldsymbol{u(i_p, T_q)}$, is defined as $\boldsymbol{o(i_p, T_q)} \times \boldsymbol{u(i_p)}$. For example, the utility of item a in transaction T_3 is 36, i.e., $u(a, T_3) = o(a, T_3) \times u(a) = 12 * 3 = 36$. The **utility of an itemset X in transaction** T_q, denoted as $u(X, Tq)$, is defined as $\sum_{i_p \in X} u(i_p, T_q)$, where $X = (i_1, i_2, \ldots, i_k)$ is a k-itemset and $X \subseteq T_q$. For example, the utility of 2-itemset (ce) in transaction T_1 is 31, i.e., $u(ce, T_1) = u(c, T_1) + u(e, T_1) = 26 \times 1 + 1 \times 5 = 31$, and utility of 3-itemset (abe) in transaction T_6 is 48, i.e., $u(abe, T_6) = u(a, T_6) + u(b, T_6) + u(e, T_6) = 1 \times 3 + 4 \times 10 + 1 \times 5 = 48$.

The **utility of an itemset X in TransSW**, denoted as $u(X) = \sum_{T_q \in D \wedge X \subseteq T_q} u(X, T_q)$, is the sum of the utilities of X in all the transactions of TransSW containing X as a subset. For example, the utility of 2-itemset (bd) in $TransSW_1$ is 146, i.e., $u(bd) = u(bd, T_2) + u(bd, T_4) + u(bd, T_8) = 66 + 52 + 28 = 146$, when the size of $TransSW_1$ is 9.

An itemset X is called a **high-utility itemset** if and only if $u(X) \geq min\_utility$, where *min_utility* is a user-defined minimum utility threshold. For example, 2-itemset (bd) is a high-utility itemset in $TransSW_1$ since $u(bd) = 146 \geq min\_utility$ if *min_utility* is 120 in $TransSW_1$.

Problem Definition Given a data stream *DS*, the size w of a transaction-sensitive sliding window *TransSW*, a user-defined minimal utility threshold *min_utility*, the task of this paper is to discover high-utility itemsets with and without negative item profits by one scan over the transaction-sensitive sliding windows.

2.2 Transaction-weighted downward closure property (TWDC-property)

In the framework of Boolean frequent itemset mining algorithms [3,5–7,11–16,19–21], **downward closure property**, i.e., *if an itemset is frequent then all its subsets must be frequent*, is usually used to mine all frequent itemsets from a large database. However, the downward closure property can not be used for mining high-utility itemsets. For example, the utility of 1-itemset (d) in $TransSW_1$ is 72, i.e., $u(d) = 72$, in Fig. 1. It is a low utility itemset but its superset 2-itemset (bd) is a high-utility itemset since $u(bd) = 146 > min\_utility$ if *min_utility* is 120. Therefore, we need new properties to mine high-utility itemsets.

In this section, an effective property, called **Transaction-Weighted Downward Closure Property** [8,9,17,18,24,25], is modified and used in our proposed algorithms to discover high-utility itemsets with and without negative item profits from streaming transactions over transaction-sensitive sliding windows.

Definition 1 (*Transaction utility*) **Transaction utility** of the transaction T_q, denoted as $tu(T_q)$, is the sum of the utilities of *all* items in T_q. For example, $tu(T_1) = u(c, T_1) + u(e, T_1) = 26 \times 1 + 1 \times 5 = 31$. **Transaction utility of current sliding window** *TransSW*, denoted as *tu*(*TransSW*), is the sum of transaction utilizes of all transactions of current transaction-sensitive sliding window.

For example, the transaction utility of each transaction in the example data stream *DS* of Fig. 1 is given in Fig. 3. Table of Fig. 3 is called **transaction utility table** (*TU-table*) in this paper. In this figure, we can find that $tu(TransSW_1) = 456$, $tu(TransSW_2) = 487$, $tu(TransSW_3) = 463$ and $tu(TransSW_4) = 442$ based on Definition 1.

Definition 2 (*Transaction-weighted utilization*) **Transaction-weighted utilization** of an itemset X, denoted as $twu(X)$, is the sum of the transaction utilities of all transactions

of current transaction-sensitive sliding window containing X as a subset. For example, in $TransSW_1$ of Fig. 1, the transaction-weighted utilization of 2-itemset (bd) is 146, i.e., $twu(bd) = tu(T_2) + tu(T_4) + tu(T_8) = 66 + 52 + 28 = 146$.

Definition 3 (*High transaction-weighted utilization itemset*) An itemset X is called a **high transaction-weighted utilization itemset (HTU-itemset)** if and only if $twu(X) \geq min\_utility$, where *min_utility* is a user-defined minimum utility threshold. For example, if *min_utility* is 120, the 2-itemset (bd) is a high transaction-weighted utilization itemset, i.e., *2-HTU-itemset*, since *twu*(bd) is 146 in $TransSW_1$ of Fig. 1.

Property 1 (Transaction-Weighted Downward Closure Property) *Let X be a k-itemset and Y be a $(k-1)$-itemset such that $Y \subset X$. If X is a high transaction-weighted utilization itemset, Y is also a high transaction-weighted utilization itemset.*

For example, let user-defined minimum utility threshold *min_utility* be 120 in $TransSW_1$ of Fig. 1, since the 3-itemset (abe) is a high transaction-weighted utilization itemset, its subset, i.e., {(a), (b), (e), (ab), (ae), (be)}, is also a set of high transaction-weighted utilization itemsets.

3 Efficient mining of high-utility itemsets from data streams: algorithms MHUI-BIT and MHUI-TID

In this section, two efficient utility mining algorithms, called **MHUI-BIT** (Mining High Utility Itemsets based on BITvector) and **MHUI-TID** (Mining High Utility Itemsets based on TIDlist), are proposed to discover and maintain a set of high-utility itemsets with and without negative item profits from data streams over transaction-sensitive sliding windows. Algorithms MHUI-BIT and MHUI-TID are composed of two core components: *item information* (discussed in Sect. 3.1) and *a lexicographical tree-based summary data structure* based on item information (discussed in Sect. 3.2). Based on stream sliding window-based framework [5–8,15,16,26], both utility mining algorithms are composed of three phases, i.e., *window initialization phase* (discussed in Sect. 3.2), *window sliding phase* (discussed in Sect. 3.3), and *high-utility itemset generation phase* (discussed in Sect. 3.4).

3.1 Two effective item-information representations: bitvector and TIDlist

The first component of proposed algorithms is **item-information**, i.e., *effective representations of items*. Two effective representations of item information, i.e., **Bitvector** and **TIDlist**, are developed and used in the proposed methods MHUI-BIT and MHUI-TID, respectively, to restrict the number of candidates and to reduce the processing time and memory usage needed. Item-information *Bitvector* and *TIDlist* are defined as follows.

Definition 4 (*Bitvector*) For each item x in the current transaction-sensitive sliding window *TransSW*, a bit-sequence with w bits, denoted as **Bitvector**(x), is constructed. The construction process of Bitvector is described as follows. If the item x is in the i-th transaction of current *TransSW*, the i-th bit of **Bitvector**(x) is set to be 1; Otherwise, the i-th bit of **Bitvector**(x) is set to be 0.

Definition 5 (*TIDlist*) For each item x in the current transaction-sensitive sliding window *TransSW*, a sorted list with at least w values, denoted as **TIDlist**(x), is constructed. The con-

Items	Bitvectors of $TransSW_1$	Bitvectors of $TransSW_2$
a	<001001011>	<010010110>
b	<010101111>	<101011111>
c	<100010011>	<000100111>
d	<011100010>	<111000100>
e	<100011111>	<000111110>

(a) Bitvector representations of Items

Items	TIDlists of $TransSW_1$	TIDlists of $TransSW_2$
a	{3, 6, 8, 9}	{3, 6, 8, 9}
b	{2, 4, 6, 7, 8, 9}	{2, 4, 6, 7, 8, 9, 10}
c	{1, 5, 8, 9}	{5, 8, 9, 10}
d	{2, 3, 4, 8}	{2, 3, 4, 8}
e	{1, 5, 6, 7, 8, 9}	{5, 6, 7, 8, 9}

(b) TIDlist representations of Items

Fig. 2 Item representations of Bitvectors and TIDlists in $TransSW_1$ and $TransSW_2$

struction process of TIDlist is described as follows. If the item x is in the i-th transaction of current *TransSW*, the value i is stored in the TIDlist(x).

For example, the representations of Bitvector and TIDlist of each item in $TransSW_1$ and $TransSW_2$ are given in Fig. 2, respectively. From this figure, we can find that item (a) appears in transactions T_3, T_6, T_8, and T_9 of $TransSW_1$. Hence, the Bitvector of item (a), i.e., **Bitvector**(a), and the TIDlist of item (a), i.e., **TIDlist**(a), are <001001011> and {3, 6, 8, 9} in $TransSW_1$, respectively. Furthermore, **Bitvector**(a) and **TIDlist**(a) within $TransSW_2$ are <010010110> and {2, 5, 7, 8}, respectively. Note that the construction processes of bitvector and TIDlist for each item of current transaction-sensitive sliding window are called **BITvector-Build** and **TIDlist-Build** based on Definitions 4 and 5 and used in the proposed algorithms MHUI-BIT and MHUI-TID, respectively.

Proposed algorithms are sliding window-based stream utility mining approaches. Hence, they are composed of three phases: *window initialization phase*, *window sliding phase*, and *high -utility itemset generation phase*, and described in following sections.

3.2 Window initialization phase of high-utility itemset mining of data streams

Based on the framework of sliding window-based pattern mining, first phase of the proposed algorithms is *window initialization phase*. The phase is activated while the number of transactions generated so far from data streams is less than or equal to a user-defined sliding window size w. In this phase, item-information, i.e., *Bitvector* and *TIDlist*, and the proposed data structure, called **LexTree-2HTU** (Lexicographical Tree with 2-HTU-itemsets), are constructed for high-utility itemset mining of data streams. LexTree-2HTU consists of two components: *item-information* and a set of *trees* with prefixes p_i, where item-information is bitvectors for MHUI-BIT algorithm or TIDlists for MHUI-TID algorithm, prefix p_i is an entry contained in item-information. Building algorithm of LexTree-2HTU using item information *Bitvector* structure, called *LexTree-2HTU-Build-byBitvector*, in window initialization phase is given in Fig. 4. Furthermore, another building method of LexTree-2HTU using *TIDlist* structure, called *LexTree-2HTU-Build-byTIDlist*, in window initialization phase is shown in Fig. 5.

TID	Transaction Utility	TID	Transaction Utility
T_1	31	T_7	105
T_2	66	T_8	37
T_3	42	T_9	53
T_4	52	T_{10}	62
T_5	22	T_{11}	42
T_6	48	T_{12}	21

Fig. 3 TU-table (transaction utility table) contains transaction utilities of each transaction of the example data streams as shown in Figure 1

Algorithm ***LexTree-2HTU-Build-byBitvector***
Input: A user-defined minimum utility threshold *min_utility* and a transaction-sensitive sliding window $TransSW_1$;
Output: A generated LexTree-2HTU of $TransSW_1$;
Method:
Begin
1. **foreach** transaction T_i of $TransSW_1$ **do**
2. perform *BITvector-Build* on items of T_i to construct *Bitvectors* of items;
3. perform *TU-table-Build* on T_i to construct *TU-table* of $TransSW_1$;
4. **endfor**
5. H_1 = a list of 1-HTU-itemsets <h_1, h_2, …, h_k> generated by *Bitvectors* and *TU-table*;
6. **foreach** entry h_i of H_1 **do**
7. C_2 = a set of candidate 2-itemsets with prefix h_i generated from H_1;
8. **foreach** entry c_i of C_2 **do**
9. Bitvector(c_i) = Bitvector(h_i) ⊕ Bitvector(q); /* ⊕ : bitwise AND operation */
10. calculate $twu(c_i)$ by accumulating transaction utilities of Bitvector(c_i) with TU-table;
11. **if** $twu(c_i) \geq min\_utility$ **then**
12. insert c_i into LexTree-2HTU as a node with root h_i;
13. **else** drop c_i from C_2;
14. **endif**
15. **endfor**
16. **endfor**
End

Fig. 4 Algorithm LexTree-2HTU-Build-byBitVector of window initialization phase

While first transaction-sensitive sliding window $TransSW_1$ is full, the proposed lexicographical tree-based summary data structure, called **LexTree-2HTU** (Lexicographical Tree with 2-HTU-itemsets), based on discovered item information, Bitvectors of MHUI-BIT and TIDlists of MHUI-TID, is constructed as follows. First, a set of high transaction-weighted utilization 1-itemsets, i.e., 1-*HTU-itemsets*, of $TransSW_1$ is generated using item information, Bitvecotor for MHUI-BIT algorithm but TIDlist for MHUI-TID algorithm, and the transaction utility table (in Step 5). Then, a set of candidate 2-itemsets, i.e., C_2, are generated by combining the set of 1-HTU-itemsets (in Step 7). As each candidate is generated, its corresponding transaction-weighted utility is determined immediately using item-information and the transaction utility table. If the computed transaction-weighted utilities of candidate 2-itemsets are greater than or equal to the user-defined minimum utility*min_utility*, these 2-HTU-itemsets are inserted into LexTree-2HTU as nodes (in Steps 8–15). Note that in MHUI-TID algorithm, the TIDlist of a candidate k-itemset is generated by joining the TIDlists of the two $(k-1)$-itemsets, where $k \geq 2$ and the set of two $(k-1)$-itemsets is a subset of this k-itemset, whereas in MHUI-BIT algorithm, the Bitvector of the two $(k-1)$-itemsets is generated by performing bitwise AND operation on the Bitvectors of the two $(k-1)$-itemsets.

Algorithm ***LexTree-2HTU-Build-byTIDlist***
Input: A user-defined minimum utility threshold *min_utility* and a transaction-sensitive sliding window $TransSW_1$;
Output: A generated LexTree-2HTU of $TransSW_1$;
Method:
Begin
1. **foreach** transaction T_i of $TransSW_1$ **do**
2. perform *TIDlist-Build* on items of T_i to construct *TIDlists* of items;
3. perform *TU-table-Build* on T_i to construct *TU-table* of $TransSW_1$;
4. **endfor**
5. H_1 = a list of 1-HTU-itemsets <h_1, h_2, …, h_k> generated by *TIDlists* and *TU-table*;
6. **foreach** entry h_i of H_1 **do**
7. C_2 = a set of candidate 2-itemsets with prefix h_i generated from H_1;
8. **foreach** entry c_i of C_2 **do**
9. TIDlist(c_i) = Bitvector(h_i) ⊗ Bitvector(q); /* ⊗ : item intersecting operation */
10. calculate $twu(c_i)$ by accumulating transaction utilities of TIDlist(c_i) with TU-table;
11. **if** $twu(c_i) \geq min\_utility$ **then**
12. insert c_i into LexTree-2HTU as a node with root h_i;
13. **else** drop c_i from C_2;
14. **endif**
15. **endfor**
16. **endfor**
End

Fig. 5 Algorithm LexTree-2HTU-Build-byTIDlist of window initialization phase

For example, the representations of *Bitvector* and *TIDlist* of Example 1 are given in Fig. 2, and the table of transaction utility for each transaction within two transaction-sensitive sliding windows, $TransSW_1$ and $TransSW_2$, is shown in Fig. 3. Based on item information and transaction utility table of Example 1, five 1-HTU-itemsets *a*, *b*, *c*, *d* and *e* in $TransSW_1$ are generated by the proposed methods when *min_utility* is 120. Afterward, four candidate 2-HTU-itemsets {*ab*, *ac*, *ad*, *ae*} with prefix (*a*) are generated from 1-HTU-itemset *a*. In the bitvector framework of MHUI-BIT algorithm, the Bitvector(*ab*) in $TransSW_1$ is <000001011> and generated by performing bitwise-AND operation on Bitvector(*a*)=<001001011> and Bitvector(*b*)=<010101111>. In the TIDlist framework of MHUI-TID algorithm, the TIDlist(*ab*) in $TransSW_1$ is {6, 8, 9} which is generated by intersecting TIDlist(*a*)={3, 6, 8, 9} and TIDlist(*b*)={2, 4, 6, 7, 8, 9}. Next, the transaction-weighted utility of candidate 2-HTU-itemset *ab*, i.e., *twu*(*ab*), can be obtained by summarizing the corresponding transaction utilities. Note that $twu(ab) = tu(T_6) + tu(T_8) + tu(T_9) = 48 + 37 + 53 = 138$ as given in the transaction utility table of Fig. 3. Other three 2-candidates {*ac*, *ad*, *ae*} with prefix *a* are verified in the same way, where $twu(ac) = tu(T_8) + tu(T_9) = 90 < min\_utility$, $twu(ad) = tu(T_3) + tu(T_8) = 79 < min\_utility$ and $twu(ae) = tu(T_6) + tu(T_8) + tu(T_9) = 138$. Therefore, only two 2-HTU-itemsets, (*ab*) and (*ae*), are maintained in the LexTree-2HTU with prefix *a* as shown in Fig. 6. From this figure, we can see that two gray rectangles, (*ac*) and (*ad*), after utility computing by proposed algorithms are not high transaction-weighted utility itemsets and not maintained in the LexTree-2HTU. LexTree-2HTU with prefixes *b*, *c*, and *d* are constructed using the same procedure is given in Fig. 7. Note that item *e* is the last entry of item information. Therefore, no candidate 2-itemsets with prefix *e* are generated and tested in our proposed algorithms. As a result, in window initialization phase, LexTree-2HTU of $TransSW_1$ for MHUI-BIT and MHUI-TID, respectively, is shown in Fig. 8.

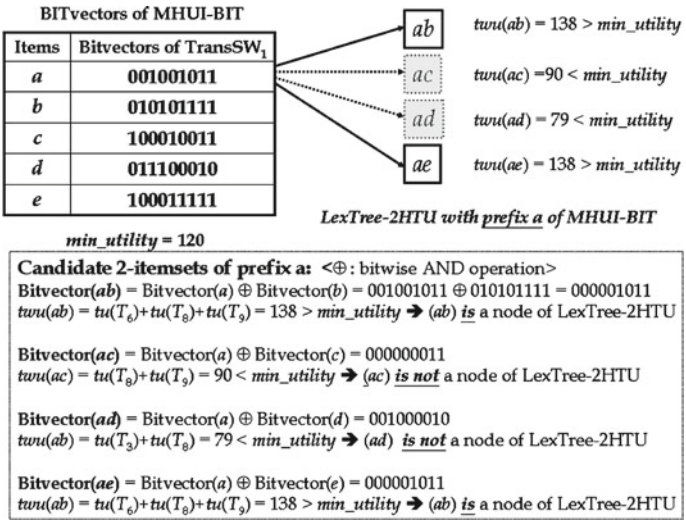

Fig. 6 LexTree-2HTU of MHUI-BIT after inserting candidate 2-itemsets with prefix *a*

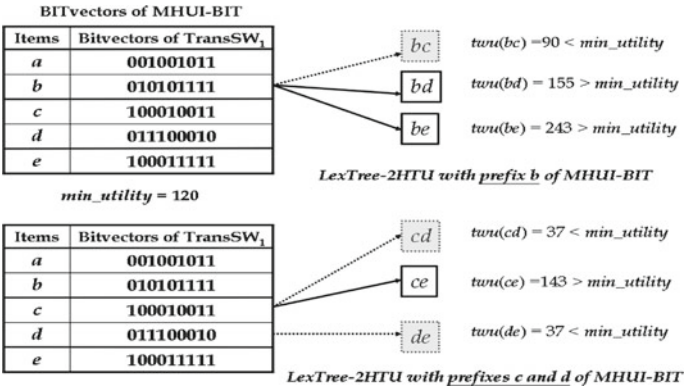

Fig. 7 LexTree-2HTU of MHUI-BIT algorithm for inserting candidate 2-itemsets with prefixes *b*, *c*, and *d*

3.3 Window sliding phase of mining high-utility itemsets from data streams

The second phase, i.e., *window sliding phase*, of mining high- utility itemsets from data streams is activated while the window is full and a new transaction arrives. In this phase, two operations are performed to update the proposed LexTree-2HTU. The first operation is to update item-information, *Bitvectors* and *TIDlists*, and the transaction utility table (TU-table) while window sliding (discussed in Sect. 3.3.1) and the second operation is to update the trees of LexTree-2HTU (discussed in Sect. 3.3.2).

Furthermore, for updating the proposed data structure LexTree-2HTU efficiently, 1-itemsets that recorded in the dropped transaction and new incoming transaction while window sliding are classified into three updated item types: **Delete-Item**, **Insert-Item** and **Intersec-Item**, using *Item-Type-Classify* algorithm. Algorithm *Item-Type-Classify* is shown in Fig. 9 and described as follows. If an item is recorded in the dropped transaction, it is a *Delete-Item* (in Steps 1–3). If an item is maintained in a new incoming transaction while

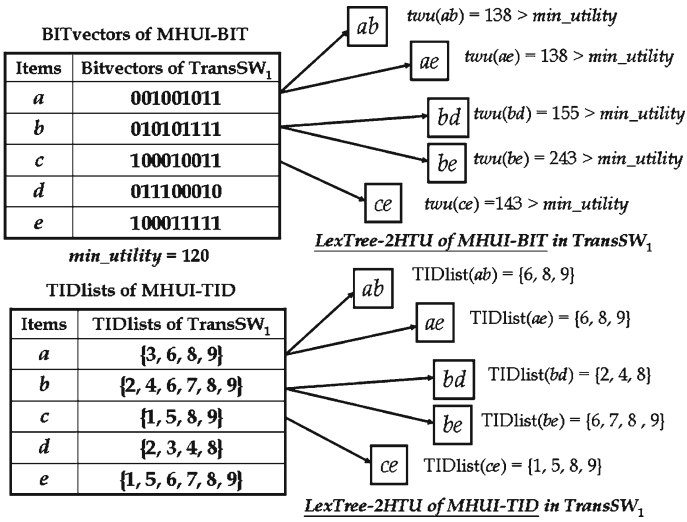

Fig. 8 LexTree-2HTUs of algorithms MHUI-BIT and MHUI-TID, respectively, after inserting all candidate 2-itemsets of $TransSW_1$

Algorithm ***Item-Type-Classify***
Input: A transaction-sensitive sliding window $TransSW_j = \{T_j, T_{j+1}, \ldots, T_{j+w-1}\}$ with w transactions, and a new incoming transaction T_{j+w};
Output: Three sets of item types: *Delete-Item-Set*, *Insert-Item-Set* and *Interesc-Item-Set* of $TransSW_{j+1} = \{T_{j+1}, T_{j+2}, \ldots, T_{j+w}\}$;
Method:
Begin
1. **foreach** item h_i of T_j **do**
2. add item h_i into *Delete-Item-Set*;
3. **endfor**
4. **foreach** item h_j of T_{j+w} **do**
5. **if** $h_j \in$ *Delete-Item-Set* **then**
6. delete item h_i from *Delete-Item-Set*;
7. add item h_i into *Intersec-Item-Set*;
8. **else** add item h_j into *Insert-Item-Set*;
9. **endif**
10. **endfor**
End

Fig. 9 Algorithm Item-Type-Classify used in window sliding phase

window sliding, it is an *Insert-Item* (in Step 8). If an item is not only in the dropped transaction but also in the new incoming transaction, the item is an *Intersec-Item* (in Step 7). For example, item *e* is a *Delete-Item*, item *b* is an *Insert-Item*, and item *c* is an *Intersec-Item* in Example 1 while sliding transaction-sensitive sliding window from $TransSW_1$ to $TransSW_2$.

3.3.1 Update item information of LexTree-2HTU and TU-table while window sliding

In the TIDlist framework of MHUI-TID algorithm, an effective TIDlist update approach, called *TIDlist-Slide*, is proposed to slide all TIDlist structures of 1-itemsets appeared in the dropped transactions for maintaining correct utility information within current transaction-sensitive sliding window. First, the transaction identifier of dropped transaction is deleted

from TIDlist of *Delete-Items* and *Intersec-Items*. After sliding, new transactions are inserted into current transaction-sensitive sliding window. Consequently, the transaction identifier of new incoming transaction is inserted into the TIDlists of *Insert-Items* and *Intersec-Items*. Furthermore, the transaction utility of new incoming transaction is computed and inserted into the transaction utility table (TU-table) for high-utility itemset mining.

For example, in Example 1, when window slides from $TransSW_1$ to $TransSW_2$, the first transaction, $T_1 = \{(c: 26), (e: 1)\}$, is deleted from and new incoming transaction, $T_{10} = \{(b: 6), (c: 2)\}$, is added into current transaction-sensitive sliding window. At this time, transaction utility of T_{10} is computed, i.e., $tu(T_{10}) = 62$, and inserted into TU-table as shown in Fig. 3, and the TIDlists of items are updated as follows. First, the TID of dropped transaction is deleted from TIDlists of *Delete-Item e* and *Intersec-Item c*. Hence, **TIDlist**(*e*) is updated from {1, 5, 6, 7, 8, 9} to {5, 6, 7, 8, 9} and **TIDlist**(*c*) is modified from {1, 5, 8, 9} to {5, 8, 9}. Then, the TID of new incoming transaction is inserted into TIDlists of *Insert-Item b* and *Intersec-Item c*. Therefore, TIDlists of items *b* and *c* are updated from {2, 4, 6, 7, 8, 9} to {2, 4, 6, 7, 8, 9, 10} and from {5, 8, 9} to {5, 8, 9, 10} for inserting T_{10} from current transaction-sensitive sliding window. TIDlists of MHUI-TID algorithm after sliding $TransSW_1$ to $TransSW_2$ using *TIDlist-Slide* is given in Fig. 2.

In the bitvector framework of MHUI-BIT algorithm, all Bitvectors of 1-itemsets are updated by performing leftmost bit shifting operation for window sliding. Then, all bitvectors of items maintained in the new incoming transaction are updated as follows. If an item h_i is recorded in the new transaction, the rightmost bit of bitvector(h_i) is set to one. The method is called *Bitvector-LMB-Shifting* (LeftMost-Bit-Shifting of Bitvector). For example, when window slides from $TransSW_1$ to $TransSW_2$, the oldest transaction T_1 is deleted and new incoming transaction T_{10} is added in the transaction-sensitive sliding window. At this time, the Bitvectors of all items from T_1 to T_{10} are updated. For instance, the **Bitvector**(*b*) and **Bitvector**(*e*) are updated from <010101111> to <101011111> and from <100011111> to <000111110> by performing leftmost bit shifting operation on **Bitvector**(*b*) and **Bitvector**(*e*), respectively.

3.3.2 Update tree nodes of LexTree-2HTU while window sliding

Based on three updated item type, the tree node updating process of LexTree-2HTU discussed in this section is composed of three situations as follows.

(a) **Item is a Delete-Item** (in Steps 5–11 of Fig. 12): If an item h_i is maintained in the dropped transaction of transaction-sensitive sliding window $TransSW_j$, the transaction-weighted utilities of its child nodes are less than or equal to that of previous window $TransSW_{j-1}$. It means that child nodes of Delete-Item may be 2-HTU-itemsets in previous transaction-sensitive sliding window but may be not 2-HTU-itemsets in current sliding window. In this case, we check transaction-weighted utilities of child nodes of the Delete-Item h_i with its item-information, *Bitvector* or *TIDlist*, based on the proposed algorithms and prune those child nodes with low transaction-weighted utilities based on a user-specified minimum utility constraint *min_utility* (in Steps 8–10 of Fig. 12).

For example, a dropped transaction T_1 of $TransSW_1$ contains two items, *c* and *e*, and a new incoming transaction T_{10} also contains two items, *b* and *c*. From this case, we can find that item *e* is a Delete-Item but item *c* is an Interesc-Item. Since there are no potential candidate 2-itemsets generated from the Delete-Item *e* in previous sliding window $TransSW_1$, no tree nodes with prefix *e* have to be checked in current LexTree-2HTU after window sliding to $TransSW_2$. Note that item *e* is a *last entry* of LexTree-2HTU in this example.

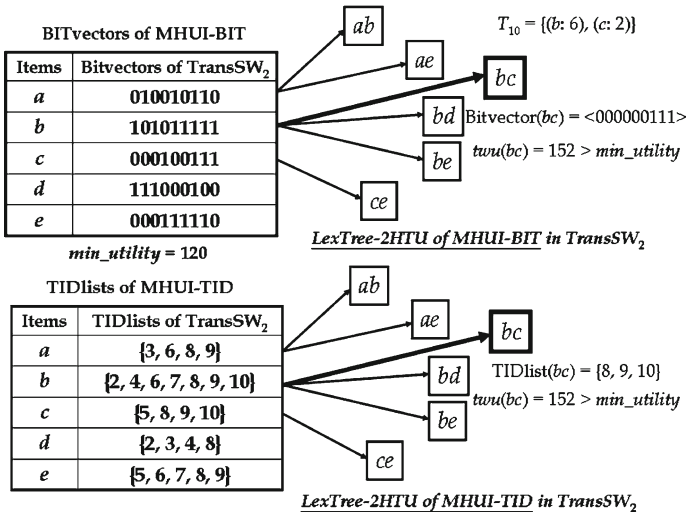

Fig. 10 Updated LexTree-2HTU after processing Delete-Item *e* and Insert-Item *b*

(b) **Item is an Insert-Item** (in Steps 12–18 of Fig. 12): If an item h_i is only contained in a new incoming transaction of current transaction-sensitive sliding window, it is an *Insert-Item*. If an item h_i is an *Insert-Item*, the transaction-weighted utilities of its child nodes with prefix h_i of LexTree-2HTU are greater than or equal to that of previous transaction-sensitive sliding window. In this case, the child nodes of an *Insert-Item* may be not 2-HTU-itemsets in previous transaction-sensitive sliding window$TransSW_{j-1}$ but may be 2-HTU-itemsets in current window $TransSW_j$. Hence, new candidate 2-itemsets are generated from this new incoming transaction and the transaction-weighted utilities of these candidate 2-itemsets are verified. If these new 2-itemsets are high transaction-weighted utilization itemsets, these 2-HTU-itemsets are inserted into the LexTree-2HTU as tree nodes with prefix h_i (in Steps 15–17 of Fig. 12).

As shown in Example 1, we can find that item *b* is an *Insert-Item*, and only one candidate 2-itemset (*bc*) is generated with prefix *b* from the new incoming transaction T_{10} = {(*b*: 6), (*c*: 2)}. After computing the transaction-weighted utility of candidate 2-itemset (*bc*), i.e., $twu(bc) = tu(T_8) + tu(T_9) + tu(T_{10}) = 152$, it is a high transaction-weighted utility itemset and inserted into the LexTree-2HTU as a branch with a prefix *b* as shown in Fig. 10. Moreover, two existing 2-HTU-itemsets, (*bd*) and (*be*), with prefix *b* are maintained in LexTree-2HTU without any transaction-weighted utility updating, because the transaction-weighted utilities of two existing nodes (*bd*) and (*be*) with prefix *b* in LexTree-2HTU are equal to that of previous transaction-sensitive sliding window.

(c) **Item is an Intersec-Item** (in Steps 19–29 of Fig. 12): If an item h_i is an Intersec-Item, original 2-HTU-itemsets with prefix h_i in previous transaction-sensitive sliding window may be not 2-HTU-itemsets in current sliding window and vice versa. In this case, we check transaction-weighted utilities of existing nodes appeared in the new incoming transaction to decide whether or not they need to be deleted from LexTree-2HTU. Furthermore, in order to decide whether or not new candidate 2-itemsets are 2-HTU-itemsets, the transaction-weighted utilities of these new candidates generated

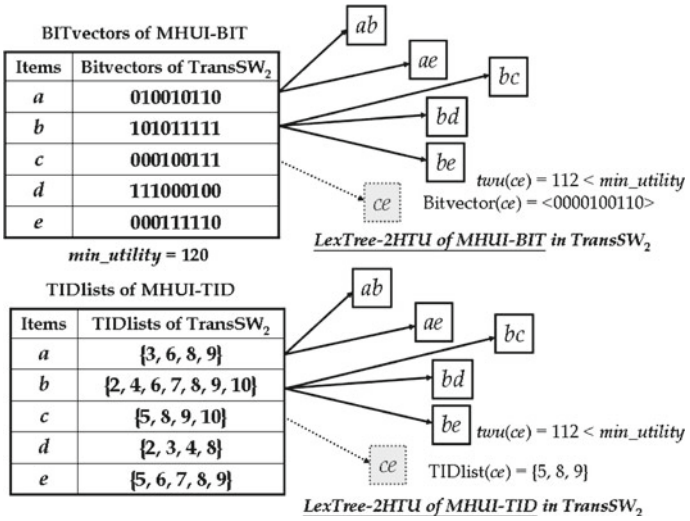

Fig. 11 Updated LexTree-2HTU after processing Delete-Items, Insert-Items and Intersec-Items while sliding windows

from the new transaction are checked. If they are new 2-HTU-itemsets, they are inserted into LexTree-2HTU as tree nodes with prefix h_i (in Steps 26–28 of Fig. 12).

For example, item c is an *Intersec-Item* of Example 1 since it appears in transactions T_1 = {(c: 26), (e: 1)} and T_{10}= {(b: 6), (c: 2)}. In this case, only one existing node, (ce), have to be checked and no new generated candidate 2-itemsets with prefix c need to be checked. After deleting the transaction T_1 from current transaction-sensitive sliding window, the transaction-weighted utility of 2-HTU-itemset (ce), i.e., $twu(ce)$, is modified from 143 to 112. New value of transaction-weighted utility of 2-itemset (ce) is less than the user-defined minimum utility *min_utility*. Consequently, the tree node (ce) with prefix c is deleted from current LexTree-2HTU. The result is given in Fig. 11. As a result, complete maintenance procedures of LexTree-2HTU in window sliding phase is given in Fig. 12.

3.4 Pattern generation phase of mining high-utility itemsets from data streams

High-utility itemset (HUI) generation phase is performed when it is needed. HUI generation phase is composed of two steps: *longer HTU-itemset generation* and *HUI-verification*. In *large HTU-itemset generation* step, both algorithms, MHUI-BIT and MHUI-TID, use level-wise-based approaches to generate a set of candidate k-HTU-itemsets, C_k, by combining previous generated $(k-1)$-HTU-itemsets, where k 2. Nota that $(k-1)$-HTU-itemsets used to generate the candidate k-HTU-itemset are subsets of the candidate k-HTU-itemset. After that, we can immediately compute the *twu* (transaction-weighted utilization) values of candidate k-HTU-itemsets using the item-information. In the proposed MHUI-BIT algorithm, it performs bitwise AND operation on corresponding bitvectors of $(k-1)$-HTU-itemsets whereas MHUI-TID algorithm uses TIDlist joining, as discussed in Sect. 3.3, to count the corresponding *twu* values of $(k-1)$-HTU-itemsets. Next, these k-HTU-itemsets, whose *twu* values are greater than or equal to the user-defined minimum utility threshold *min_utility*, are inserted into LexTree-2HTU as tree nodes. Process *longer HTU-itemset generation* stops when no candidates with longer size are generated.

Algorithm ***LexTree-2HTU-Update-byItemInformation (Bitvector, TIDlist)***
Input: A user-defined minimum utility threshold *min_utility*, a transaction utility table *TU-table*, a transaction-sensitive sliding window $TransSW_j = \{T_j, T_{j+1}, \ldots, T_{j+w-1}\}$ with w transactions, and a new incoming transaction T_{j+w};
Output: An updated LexTree-2HTU of $TransSW_{j+1} = \{T_{j+1}, T_{j+2}, \ldots, T_{j+w}\}$;
Method:
Begin
/* Line 1 is only used for MHUI-BIT algorithm */
1. perform *Bitvector-LMB-Shifting* on each **Bitvector**(h_i) of current TransSW;

/* Line 2 is only used for MHUI-TID algorithm*/
2. perform *TIDlist-Slide* on TIDlists of items within T_j and T_{j+w};

3. compute $tu(T_{j+w})$ of T_{j+w} and insert it into *TU-table*;
   /* *tu*: transaction utility */
4. perform *Item-Type-Classify* to find three item types, *Delete-Item-Set*, *Insert-Item-Set* and *Intersec-Item-Set*, from dropped transaction T_j and new incoming transaction T_{j+w};
5. **foreach** entry h_i of *Delete-Item-Set* **do** /* Case (a): if item is a Delete-Item */
6. delete the transaction utility of T_j, $tu(T_j)$, from $twu(h_i)$;
7. delete $tu(T_j)$ from *twu*(child nodes with prefix h_i);
   /* *twu*: transaction-weighted utilization */
8. **if** *twu*(child node p_j with prefix h_i) < *min_utility* **then**
9. delete node p_j with prefix h_i from LexTree-2HTU;
10. **endif**
11. **endfor**
12. **foreach** entry h_i of *Insert-Item-Set* **do** /* Case (b): if item is an Insert-Item */
13. generate a set of new candidate 2-itemsets with prefix h_i;
14. compute *twu* values of these candidate 2-itemsets;
15. **if** *twu*(candidate 2-itemsets) ≥ *min_utility* **then**
16. insert these 2-HTU-itemsets into LexTree-2HTU as tree nodes with prefix h_i;
17. **endif**
18. **endfor**
19. **foreach** entry h_i of *Intersec-Item-Set* **do** /* Case (c): if item is an Intersec-Item */
20. delete value of $tu(T_j)$ from *twu* values of all existing child nodes with prefix h_i;
21. **if** *twu*(child nodes with prefix h_i) ≤ *min_utility* **then**
22. delete these child nodes from LexTree-2HTU;
23. **endif**
24. generate a set of new candidate 2-itemsets with with prefix h_i;
25. compute *twu* values of these candidate 2-itemsets;
26. **if** *twu*(candidate 2-itemsets) ≥ *min_utility* **then**
27. insert these 2-HTU-itemsets into LexTree-2HTU as tree nodes with prefix h_i;
28. **endif**
29. **endfor**
End

Fig. 12 Maintenance Algorithms of LexTree-2HTU in the window sliding phase

For example, an itemset X is a *high-utility itemset* in Example 1 if and only if $u(X) \geq$ *min_utility*, where *min_utility* is 120. In $TransSW_1$, five 2-HTU-itemsets, {(*ab*), (*ae*), (*bd*), (*be*), (*ce*)}, as shown in Fig. 8, are generated by the proposed algorithms MHUI-BIT and MHUI-TID. But only one candidate 3-HTU-itemset (*abe*) is generated by combining three 2-HTU-itemsets, i.e., (*ab*), (*ae*) and (*be*), based on transaction-weighted downward closure property. After that, in MHUI-TID algorithm, **TIDlist**(*abe*) is {6, 7, 8} by joining **TIDlist**(*ab*) and **TIDlist**(*ae*). Note that **TIDlist**(*ab*) = {6, 8, 9} and **TIDlist**(*ae*) = {6, 8, 9} as shown in Fig. 8. Therefore, $twu(abe) = tu(T_6) + tu(T_7) + tu(T_8) = 138$ *min_utility*.

Utility Pattern Types	$TransSW_1$	$TransSW_2$
High Transaction-weighted Utilization (HTU) Itemsets generated after performing *longer HTU-itemset generation*	*a, b, c, d, e* *ab, ae, bd, be, ce* *abe*	*a, b, c, d, e* *ab, ae, bc, bd, be* *abe*
High Utility Itemsets (HUI) generated after performing *HUI-verification*	*b, bd, be*	*b, bd, be*

Fig. 13 High-Utility Itemsets generated after performing *longer HTU-itemset generation* and *HUI-verification* in $TransSW_1$ and $TransSW_2$, respectively

Consequently, candidate 3-HTU-itemset (*abe*) is a 3-HTU-itemset generated using MHUI-TID algorithm. In addition, in MHUI-BIT algorithm, the bitvector of candidate 3-HTU-itemset (*abe*), i.e., **Bitvector**(*abe*), is <000001011> by performing bitwise AND operation on **Bitvector**(*ab*) and **Bitvector**(*ae*), where **Bitvector**(*ab*) is <000001011> and **Bitvector**(*ae*) is <00001011>. Because no more new candidates with longer size are generated in this case, the first step of high-utility itemset generation phase of proposed algorithms stop.

After all HTU-itemsets are generated and inserted into LexTree-2HTU by the proposed algorithms, the second step of high-utility itemset generation phase, *HUI-verification*, is performed for these generated HTU-itemsets. In the HUI-verification step of MHUI-BIT algorithm, true utility value of HTU-itemsets generated from *longer HTU-itemset generation* step is computed by one pass over a partial transaction-sensitive sliding window*pTransSW*. The range of *pTransSW* of MHUI-BIT is determined by performing bitwise OR operation on the bitvectors of generated HTU-itemsets. In worse case, the size of *pTransSW* is equal to that of current *TransSW*. However, in the HUI-verification step of MHUI-TID algorithm, true utility value of HTU-itemsets is computed by one pass over *pTransSW*. Based on TIDlist structures, the range of *pTransSW* of MHUI-TID is determined by performing the union operation on TIDlists of HTU-itemsets generated from previous *longer HTU-itemset generation* step. In Example1, after performing *longer HTU-itemset generation* and *HUI-verification*, a set of high-utility itemsets generated from current transaction-sensitive sliding window is shown in Fig. 13.

3.5 Mining high-utility itemsets from data streams with negative item profits

In this section, our proposed algorithms, MHUI-BIT and MHUI-TID, are extended to discover high-utility itemsets with negative item profits from data streams over transaction-sensitive sliding windows. A simple but effective principle, called **AllNIP** (All Negative Item Profits)—**drop principle**, used in both algorithms is that at least one item of any generated candidate should have positive profits. Otherwise, the generated candidate should be dropped in advance for LexTree-2HTU construction in window initialization phase, window sliding phase and pattern generation phase.

In the window initialization phase of mining high-utility itemsets from data streams with negative item profits, an extended LexTree-2HTU building procedure, called *LexTree-2HTU-Build-byBitvector-NIP* (Negative Item Profits), used in MHUI-BIT algorithm is proposed and given in Fig. 14. There are two modifications in *LexTree-2HTU-Build-by-Bitvector-NIP*. First, in Step 5 of algorithm *LexTree-2HTU-Build-byBitvector-withNIP* of Fig. 14, H_1 is composed of a list of utility 1-itemsets $< u_1, u_2, \ldots, u_k >$ generated by Bitvectors and TU-table, not composed of a list of 1-HTU-itemsets as shown in Fig. 4. Second, in Steps 9 to 10, if a candidate 2-itemset c_i is composed of two items with negative item profits, it is dropped from C_2 based on AllNIP-drop principle. Note that C_2 is a set of candidate 2-itemsets with prefix h_i generated from H_1. Besides, in MHUI-TID algorithm, a

Algorithm ***LexTree-2HTU-Build-byBitvector-NIP (Negative Item Profits)***
Input: A user-defined minimum utility threshold *min_utility* and a transaction-sensitive sliding window $TransSW_1$;
Output: A generated LexTree-2HTU of $TransSW_1$;
Method:
Begin
1. **foreach** transaction T_i of $TransSW_1$ **do**
2. perform *BITvector-Build* on items of T_i to construct *Bitvectors* of items;
3. perform *TU-table-Build* on T_i to construct *TU-table* of $TransSW_1$;
4. **endfor**
5. H_1 = a list of utility 1-itemsets <u_1, u_2, …, u_k> generated by *Bitvectors* and *TU-table*;
6. **foreach** entry h_i of H_1 **do**
7. C_2 = a set of candidate 2-itemsets with prefix h_i generated from H_1;
8. **foreach** entry c_i of C_2 **do**
9. **if** c_i is composed of all items with negative item profits **then**
10. delete c_i from C_2;
11. **else**
12. Bitvector(c_i) = Bitvector(h_i) ⊕ Bitvector(q); /* ⊕ : bitwise AND operation */
13. calculate $twu(c_i)$ by accumulating transaction utilities of Bitvector(c_i) with TU-table;
14. **if** $twu(c_i) \geq min\_utility$ **then**
15. insert c_i into LexTree-2HTU as a node with root h_i;
16. **else** drop c_i from C_2;
17. **endif**
18. **endif**
19. **endfor**
20. **endfor**
End

Fig. 14 Algorithm LexTree-2HTU-Build-byBitVector-NIP of window initialization phase

modified LexTree-2HTU building procedure, called *LexTree-2HTU-Build-byTIDlist-NIP*, is also developed based on ALLNIP-drop principle and shown in Fig. 15.

In the window sliding phase, LexTree-2HTU updating procedure of proposed algorithms, called *LexTree-2HTU-Update-byItemInformation-NIP*, is proposed based on item information Bitvector or TIDlist and is given in Fig. 16. In *LexTree-2HTU-Update-byItemInformation-NIP*, items of current transaction-sensitive sliding window are classified into three types, *Delete-Item*, *Insert-Item* and *Interesc-Item*, after performing *item-Type-Classify*. Based on the AllNIP-drop principle, the existing 1-itemsets of LexTree-2HTU extracted from the dropped transaction T_j and the new incoming transaction T_{j+w}, i.e., items in *Insert-Item-Set* and *Intersec-Item-Set* in Steps 13 and 24, have to be processed for LexTree-2HTU maintenance. In both cases, only candidate 2-itemsets with *positive itemset profits* are generated for verifying their transaction-weighted utilization values. Note that if a utility itemset consists of at least one item with positive item profit, it is an itemset with *positive itemset profit*. If the transaction-weighted utilization value of a candidate 2-itemset with positive itemset profit is greater than or equal to a user-defined minimum utility constraint *min_utility*, the 2-itemset is a 2-HTU-itemset and inserted into LexTree-2HTU as tree nodes.

In the high-utility itemset (HUI) generation phase, steps *longer HTU-itemset generation* and *HUI-verification* are modified for mining high-utility itemsets from data streams with negative item profits as follows. In *large HTU-itemset generation* step, both algorithms, MHUI-BIT and MHUI-TID, use level-wise based methods to generate a set of candidate k-HTU-itemsets by combining previous generated $(k-1)$-HTU-itemsets based on AllNIP-drop principle. After that, the *twu* (transaction-weighted utilization) values of candidate

Algorithm ***LexTree-2HTU-Build-byTIDlist-NIP (Negative Item Profits)***

Input: A user-defined minimum utility threshold *min_utility* and a transaction-sensitive sliding window $TransSW_1$;
Output: A generated LexTree-2HTU of $TransSW_1$;
Method:
Begin
1. **foreach** transaction T_i of $TransSW_1$ **do**
2. perform *TIDlist-Build* on items of T_i to construct *TIDlists* of items;
3. perform *TU-table-Build* on T_i to construct *TU-table* of $TransSW_1$;
4. **endfor**
5. H_1 = a list of utility 1-itemsets <u_1, u_2, …, u_k> generated by *TIDlists* and *TU-table*;
6. **foreach** entry h_i of H_1 **do**
7. C_2 = a set of candidate 2-itemsets with prefix h_i generated from H_1;
8. **foreach** entry c_i of C_2 **do**
9. **if** c_i is composed of all items with negative item profits **then**
10. delete c_i from C_2;
11. **else**
12. TIDlist(c_i) = Bitvector(h_i) ⊗ Bitvector(q); /* ⊗ : item intersecting operation */
13. calculate $twu(c_i)$ by accumulating transaction utilities of TIDlist(c_i) with TU-table;
14. **if** $twu(c_i) \geq min\_utility$ **then**
15. insert c_i into LexTree-2HTU as a node with root h_i;
16. **else** drop c_i from C_2;
17. **endif**
18. **endif**
19. **endfor**
20. **endfor**
End

Fig. 15 Algorithm LexTree-2HTU-Build-byTIDlist-NIP of window initialization phase

k-HTU-itemsets can be computed using the item-information. In MHUI-BIT algorithm, bitwise AND operation is performed on corresponding bitvectors of ($k - 1$)-HTU-itemsets whereas TIDlist joining is performed in TIDlists of previous generated HTU-itemsets in MHUI-TID algorithm. These generated k-HTU-itemsets, whose *twu* values are greater than or equal to the user-defined minimum utility constraint *min_utility*, are inserted into LexTree-2HTU as nodes. If no more new candidates with longer size are generated, the modified first step of high-utility itemset generation phase of proposed algorithms stop. After all HTU-itemsets are generated and inserted into LexTree-2HTU by the proposed algorithms, the second step of high- utility itemset generation phase, *HUI-verification*, is performed for these generated HTU-itemsets. The methods used in step *HUI-verification* of both algorithms are equal to that of mining high-utility itemsets without negative item profits as discussed in Sect. 3.4.

4 Experimental evaluation

In this section, several experiments are evaluated to compare our proposed algorithms, MHUI-TID and MHUI-BIT, with an existing approach THUI-Mine [8]. Based on our knowledge, THUI-Mine algorithm is the first method to mine a set of temporal high-utility itemsets from data streams without negative item profits. All the programs are implemented in C++ STL and compiled with Visual C++.NET compiler. All the programs are performed on AMD Athlon(tm) 64 Processor 3,000+1.8 GHz with 1 GB memory and running on Windows XP

Algorithm ***LexTree-2HTU-Update-byItemInformation-NIP (Negative Item Profits)***
Input: A user-defined minimum utility threshold *min_utility*, a transaction utility table *TU-table*, a transaction-sensitive sliding window $TransSW_j = \{T_j, T_{j+1},\ldots, T_{j+w-1}\}$ with w transactions, and a new incoming transaction T_{j+w};
Output: An updated LexTree-2HTU of $TransSW_{j+1} = \{T_{j+1}, T_{j+2},\ldots, T_{j+w}\}$;
Method:
Begin
/* Line 1 is only used for MHUI-BIT algorithm */
1. perform *Bitvector-LMB-Shifting* on each **Bitvector**(h_i) of current TransSW;
/* Line 2 is only used for MHUI-TID algorithm*/
2. perform *TIDlist-Slide* on TIDlists of items within T_j and T_{j+w};

3. compute $tu(T_{j+w})$ of T_{j+w} and insert it into *TU-table*;
   /* *tu*: transaction utility */
4. perform *Item-Type-Classify* to find three item types, *Delete-Item-Set*, *Insert-Item-Set* and *Intersec-Item-Set*, from dropped transaction T_j and new incoming transaction T_{j+w};
5. **foreach** entry h_i of *Delete-Item-Set* **do** /* Case (a): if item is a Delete-Item */
6. delete the transaction utility of T_j, $tu(T_j)$, from $twu(h_i)$;
7. delete $tu(T_j)$ from *twu*(child nodes with prefix h_i);
   /* *twu*: transaction-weighted utilization */
8. **if** *twu*(child node p_j with prefix h_i) < *min_utility* **then**
9. delete node p_j with prefix h_i from LexTree-2HTU;
10. **endif**
11. **endfor**
12. **foreach** entry h_i of *Insert-Item-Set* **do** /* Case (b): if item is an Insert-Item */
13. generate a set of new candidate 2-itemsets with prefix h_i and *positive itemset profits*;
14. compute *twu* values of these candidate 2-itemsets;
15. **if** *twu*(candidate 2-itemsets) ≥ *min_utility* **then**
16. insert these 2-HTU-itemsets into LexTree-2HTU as tree nodes with prefix h_i;
17. **endif**
18. **endfor**
19. **foreach** entry h_i of *Intersec-Item-Set* **do** /* Case (c): if item is an Intersec-Item */
20. delete value of $tu(T_j)$ from *twu* values of all existing child nodes with prefix h_i;
21. **if** *twu*(child nodes with prefix h_i) ≤ *min_utility* **then**
22. delete these child nodes from LexTree-2HTU;
23. **endif**
24. generate a set of new candidate 2-itemsets with with prefix h_i and *positive itemset profits*;
25. compute *twu* values of these candidate 2-itemsets;
26. **if** *twu*(candidate 2-itemsets) ≥ *min_utility* **then**
27. insert these 2-HTU-itemsets into LexTree-2HTU as tree nodes with prefix h_i;
28. **endif**
29. **endfor**
End

Fig. 16 Algorithm *LexTree-2HTU-Update-byItemInformation-NIP* in window sliding phase

system. All testing data were generated by the synthetic data generator provided by Agrawal et al. in [1,3]. However, the IBM synthetic data generator only generates the quantity of 0 or 1 for each item in a transaction. In order to adapt the databases into the scenario of utility mining, the quantity of each item and the utility of each item are randomly generated. The symbols used in these experiments are shown in Table 1.

In these experiments, the quantity of each item in each transaction, Q_{item}, is generated randomly, ranging from 1 to 10. The utility of each item, U_{item}, stored in the utility table is synthetically created by assigning a utility value to each item randomly, ranging from 1 to 1,000. Observed from real-world databases that most items are in the low profit range,

Table 1 Meanings of symbols used in the experiments

Symbols	Meaning
$\|w\|$	Window size (30 K)
$\|P\|$	Partition size used in THUI-Mine (10 K)
min_utility	Minimum utility threshold (0.2% × *tu*(*curTransSW*) to 1% × *tu*(*curTransSW*))
Q_{item}	Quantity of each item in each transaction (1–5)
U_{item}	Utility of each item (1–1 K)
Item-freq	Frequency of item, i.e., *the average number of TIDlist of all items*. Item-freq is the average number of transactions each item contained.

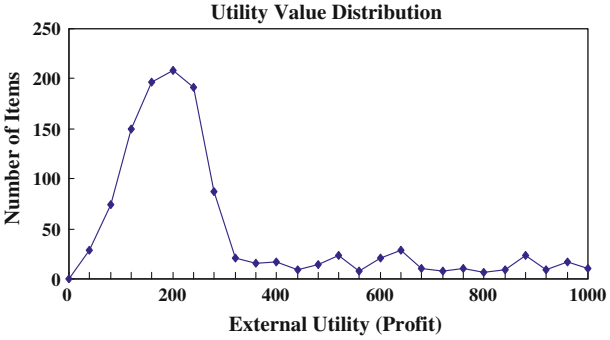

Fig. 17 Profit value distribution with distinct 1,000 items

Table 2 Name and parameter of each data set with D100 K

Synthetic data sets	Average items per transaction	Average length of maximal utility pattern	Number of items
T5I4	5	4	1,000
T10I6	10	6	1,000
T15I10	15	10	1,000
T20I15	20	15	1,000
T30I20	30	20	1,000

the utility value generated using a log normal distribution, as is similar to the model used in [8,18]. Figure 17 gives the utility table value distribution of 1,000 distinct items. Names and parameter settings of each dataset used in these experiments are given in Table 2. In all experiments, the transactions of each dataset are looked up in sequence to simulate an environment of data streams.

4.1 Evaluation of execution time

In this section, two experiments of execution time are evaluated. In first experiment, we compare the execution time of our proposed algorithms, MHUI-BIT and MHUI-TID, with an existing algorithm THUI-Mine under various minimum utility thresholds. Figure 18 show

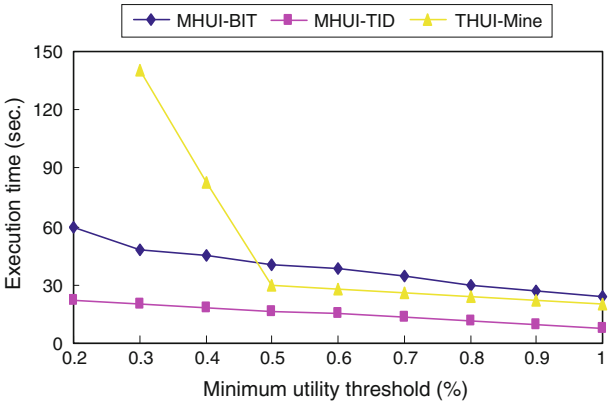

Fig. 18 Execution time for MHUI-BIT, MHUI-TID and THUI-Mine under various minimum utility thresholds on T10I6D100K ($|w| = 30$K, $|P| = 10$K)

the execution time for algorithms MHUI-BIT, MHUI-TID and THUI-Mine as the minimum utility threshold *min_utility* is increased from 0.2% × *tu*(*curTransSW*) to 1% × *tu*(*curTransSW*). From this figure, we can find that the performance of proposed algorithm MHUI-TID is better than that of THUI-Mine, especially when the minimum utility threshold *min_utility* is small. Although the execution time of MHUI-BIT algorithm is greater than that of THUI-Mine when the utility threshold is greater than 0.5% × *tu*(*curTransSW*), the proposed MHUI-BIT algorithm runs significantly faster than that of THUI-Mine when the utility threshold is smaller than 0.5% × *tu*(*curTransSW*). From this experiment, the less number of generated candidates of THUI-Mine is the reason that the execution time of THUI-Mine is less than that of MHUI-BIT while minimum utility thresholds are greater than 0.5%. Maintenance performance of item information representation used in both algorithms MHUI-BIT and MHUI-TID is the major reason that MHUI-TID algorithm always performs MHUI-BIT algorithm. Note that execution time used in these experiments is composed of building time of proposed data structure in window initialization phase, updating time of proposed data structure in window sliding phase and generation time of high-utility itemsets from proposed data structure in pattern generation phase.

In second experiment, we investigate the effects of varying dataset characteristics on the execution time of algorithms MHUI-BIT, MHUI-TID and THUI-Mine. Figure 19 shows the experimental results of algorithms MHUI-BIT, MHUI-TID and THUI-Mine under various datasets, T15I10D100K, T20I15D100K and T30I20D100K. In this experiment, the window size $|w|$ is fixed to 30,000, the partition size $|P|$ is fixed to 10,000 [8], and the user-defined minimum utility threshold *min_utility* is fixed to 1% × *tu*(*curTransSW*). From this figure, we can see that the existing algorithm THUI-Mine only runs successfully in the dataset T15I10D100K. The execution time of THUI-Mine can not be drawn in this figure since it needs much more execution time. Therefore, we can find that the relation of execution time requested is "MHUI-TID ≪ MHUI-BIT ≪ THUI-Mine" from this experiment. Although the proposed MHUI-BIT algorithm does not completely outperform the existing algorithm THUI-Mine in dataset T10I6D100K as shown in Fig. 18, it outperforms THUI-Mine in large datasets, such as T15I10D100K, T20I15D100 and T30I20D100K in Fig. 18. Consequently, MHUI-BIT maybe runs a little slower than that of THUI-Mine in small datasets, it runs significantly faster than that of THUI-Mine in large datasets. Furthermore, the proposed algorithm MHUI-TID outperforms THUI-Mine in all datasets T15I10D100K, T20I15D100 and T30I20D100K.

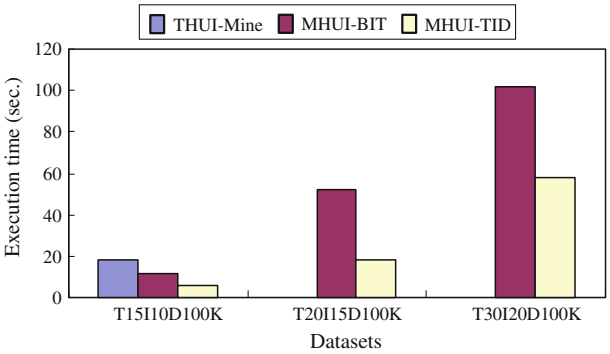

Fig. 19 Execution time of algorithms MHUI-BIT, MHUI-TID, and THUI-Mine under various datasets: T15I10D110K, T20I15D100K and T30I20D100K

Table 3 *Item-freq* in each dataset

Dataset	*Item-freq*
T15I10D100K	480
T20I15D100K	550
T30I20D100K	750

Table 3 shows the average number of transactions containing each item, i.e., *Item-freq*., where window size $|w|$ is 30 K. From this table, we can observe that *Item-freq* increases a little as the dataset become a larger one. The ratio of *Item-freq* to*window size* is changed from 1.6% (480/30,000) to 2.5% (750/30,000) in T15I10D100K to T30I20D100K, respectively. Since the ratio is apparently small, we can obtain that the performance of algorithm MHUI-TID is better than that of algorithm MHUI-BIT. Moreover, Figs. 18 and 19 also provide the performance conclusion.

Based on the experiments evaluated in Sect. 4.1, the proposed MHUI-TID algorithm is chosen to compare with THUI-Mine algorithm in following experiments, because the performance of MHUI-TID is better than that of MHUI-BIT.

4.2 Evaluation of different minimum utility thresholds

In this section, we compare the performance of the proposed algorithm MHUI-TID with an existing algorithm THUI-Mine using the dataset T5I4D100K under various minimum utility thresholds. The number of item types is fixed to 1,000, the size of sliding window w is fixed to 5,000 transactions and the partition size is fixed to one transaction. The execution time, memory requirement and the number of generated candidate utility itemsets of algorithms MHUI-TID and THUI-Mine are evaluated under different minimum utility thresholds where *min_utility* is increased from 1% × *tu*(*curTransSW*) to 6% × *tu*(*curTransSW*). The execution time comparison of algorithms MHUI-TID and THUI-Mine under various minimum utility thresholds is given in Fig. 20. From this figure, we can see that MHUI-TID runs faster than THUI-Mine under various minimum utility thresholds from 1% × *tu*(*curTransSW*) to 6% × *tu*(*curTransSW*). Hence, our proposed algorithm is more suitable for mining high-utility itemsets from data streams.

Figure 21 gives the result of memory usage comparison between MHUI-TID and THUI-Mine. From this figure, we can find that the memory requirement used in MHUI-TID algorithm is almost the same and small usage. It is obvious from that the number of candidate

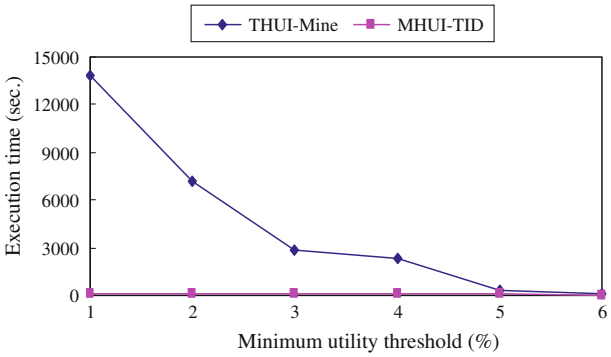

Fig. 20 Execution time comparison of MHUI-TID and THUI-Mine under various minimum utility thresholds (T5I4D100K, $|w| = 5$K, $|P| = 1$)

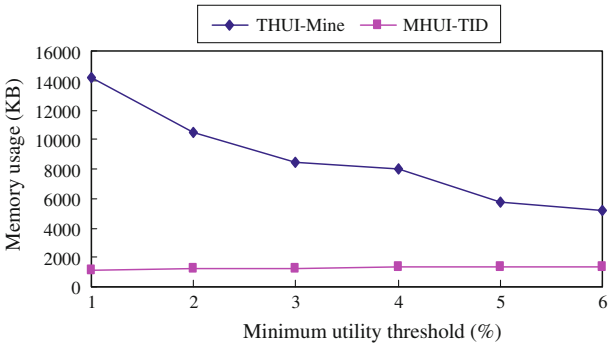

Fig. 21 Memory usage comparison of MHUI-TID and THUI-Mine under different minimum utility thresholds (T5I4D100K, $|w| = 5$K, $|P| = 1$)

itemsets generated increases smoothly under different minimum utility thresholds. But, the memory requirement of THUI-Mine algorithm increases dramatically as the minimum utility threshold decreases. This is because that, in our proposed algorithms, utility value of each candidate is computed by its item information, TIDlist or Bitvector, before it is inserted into the proposed data structure LexTree-2HTU. But the number of candidates of the THUI-Mine is generated and inserted into its data structure before THUI-Mine counts its support. Therefore, the proposed algorithm MHUI-TID uses less memory than that of THUI-Mine algorithm in an environment of data streams. Consequently, our proposed algorithm MHUI-TID runs significant faster and consumes less memory usage than that of THUI-Mine algorithm.

4.3 Performance evaluation of high-utility itemsets with negative item profits

In this section, several experiments are evaluated to compare the performance of our proposed algorithms **MHUI-BIT**, **MHUI-TID**, **MHUI-BIT-NIP** (MHUI-BIT with Negative Item Profits) and **MHUI-TID-NIP** (MHUI-TID with Negative Item Profits). In these experiments, we randomly generate the quantity of each item in each transaction of streaming data, ranging from 1 to 10, and generate the utility value of each item maintained in the utility table, ranging from −100 to 1,000. Furthermore, we generate the utility values using a log normal distribution, as used in [8,18], for simulating the low profit range and low negative item

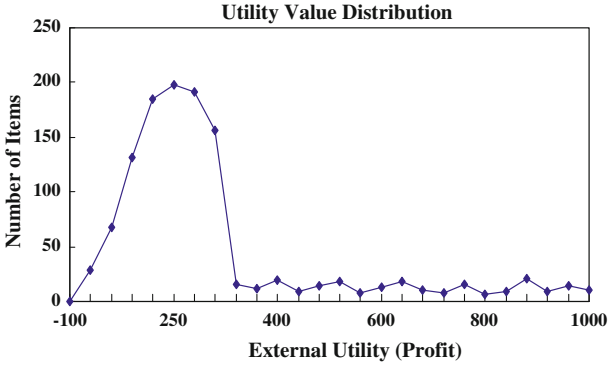

Fig. 22 Profit value distribution with distinct 1,000 items with negative item profits

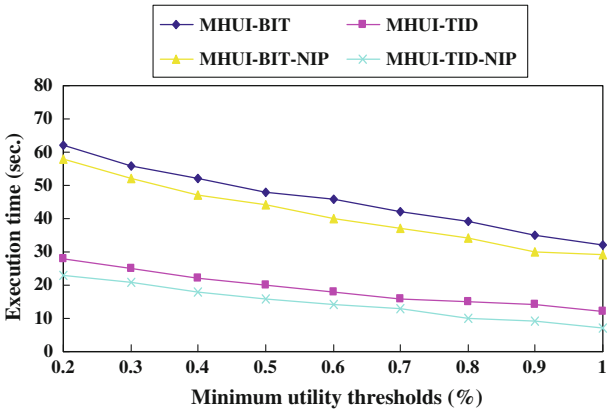

Fig. 23 Execution time for MHUI-BIT, MHUI-TID, MHUI-BIT-NIP and MHUI-TID-NIP under various minimum utility thresholds on T10I6D100K ($|w| = 30$ K, $|P| = 10$ K)

profits of utility values in real-world databases. Figure 22 shows the profit value distribution of 1,000 items with negative item profits.

Figure 23 gives the experimental result of execution time comparison of algorithm MHUI-BIT, MHUI-TID, MHUI-BIT-NIP and MHUI-TID-NIP under various minimum utility threshold *min_utility* from 0.2% × *tu*(*curTransSW*) to 1% × *tu*(*curTransSW*) on dataset T10I6D100K. From this figure, we can find that the performance of proposed algorithm MHUI-TID-NIP is better than that of MHUI-TID and the execution time of MHUI-BIT-NIP is less than that of MHUI-BIT. Consequently, the experimental result shows that the AllNIP-drop principle discussed in Sect. 3.5 is suitable for mining high-utility itemsets from data streams with negative item profits. Note that execution time used in these experiments is composed of building time of proposed data structure in window initialization phase, updating time of proposed data structure in window sliding phase and generation time of high-utility itemsets from proposed data structure in pattern generation phase.

5 Conclusions

In this paper, we addressed the problem of mining high-utility itemsets from data streams. Under the streaming environment, limited memory usage and real-time processing are major

challenges for mining utility itemsets from data streams. Two efficient algorithms MHUI-TID and MHUI-BIT are proposed for discovering high-utility itemsets from streaming data without negative profits. Two item information representations, Bitvector and TIDlist, are used in the proposed algorithms to improve the performance of utility mining. Both item information representations can be used to generate utility itemsets from current sliding window without rescanning the data streams. Moreover, a summary data structure LexTree-2HTU is developed for maintaining a set 2-HTU-itemsets from current transaction-sensitive stream sliding window effectively. Besides, a new issue of utility mining with negative item profits is addressed and two adapted approaches of algorithms MHUI-BIT and MHUI-TID are developed to discover high-utility itemsets with negative item profits from data streams. The experimental results show that our approaches can find the high-utility itemsets with higher performance by generating less candidate itemsets and reducing computation time of utility value. Furthermore, the experimental results also show that our algorithms outperform the existing algorithm THUI-Mine under different experimental conditions and are scalable with large datasets. For future work, we would extend the concepts developed in this work to discover high-utility itemsets from data streams over time-sensitive sliding windows with and without negative item profits, mining utility itemsets from data steams with millions of objects, and mining utility itemsets from data streams with a landmark window model.

References

1. Agrawal R, Imielinski T, Swami A (1993) Mining associations rules between sets of items in large Databases. In: Proceedings of ACM SIGMOD international conference on management of data, pp 207–216
2. Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo AI (1996) Fast discovery of associations rules, advances in knowledge discovery and data mining. AAAI/MIT Press, Cambridge, pp 307–328
3. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large database. In: Proceedings of the 20th international conference on very large databases (VLDB), pp 487–499
4. Chan R, Yang Q, Shen YD (2003) Mining high utility itemsets. In: Proceedings of the 3rd IEEE international conference on data mining (ICDM)
5. Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: Proceedings of the international conference on knowledge discovery and data mining (SIGKDD), pp 487–492
6. Chang J, Lee W (2004) A sliding window method for finding recently frequent itemsets over online data streams. J Inf Sci Eng (JISE) 20(4):753–762
7. Chi Y, Wang H, Yu PS, Muntz R (2004) Moment: maintaining closed frequent itemsets over a stream sliding window. In: Proceedings of the IEEE International Conference on Data Mining (ICDM), pp. 59–66
8. Chu CJ, Tseng VS, Liang T (2008) An efficient algorithm for mining temporal high utility itemsets from data streams. J Syst Softw 81(7):1105–1117
9. Chu CJ, Tseng VS, Liang T (2009) An efficient algorithm for mining high utility itemsets with negative item values in large databases. Appl Math Comput 215(2):767–778
10. Golab L, Ozsu MT (2003) Issues in data stream management. ACM SIGMOD Rec 32(2):5–14
11. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 1–12
12. Jin C, Qian W, Sha C, Yu J, Zhou A (2003) Dynamically maintaining frequent items over a data stream. In: Proceedings of the ACM 12th international conference on information and knowledge management (CIKM), pp 287–294
13. Lee CH, Lin CR, Chen MS (2001) Sliding-window filtering: an efficient algorithm for incremental mining. In: Proceedings of the ACM 10th international conference on information and knowledge management (CIKM), pp 263–270
14. Li H-F, Lee S-Y, Shan M-K (2008) DSM-FI: an efficient algorithm for mining frequent itemsets in data streams. Knowl Inf Syst Int J (KAIS) 17(1):79–97
15. Li H-F, Lee S-Y (2009) Mining frequent itemsets over data streams using efficient window sliding techniques. Expert Syst Appl (ESWA) 36(2, Part 1):1466–1477

16. Li H-F, Ho C-C, Lee S-Y (2009) Incremental updates of closed frequent itemsets over continuous data streams. Expert Syst Appl (ESWA) 36(2, Part 1):2451–2458
17. Li Y-C, Yeh J-S, Chang C-C (2008) Isolated items discarding strategy for discovering high utility itemsets. Data Knowl Eng (DKE) 64(1):198–217
18. Liu Y, Liao W, Choudhary A (2005) A fast high utility itemsets mining algorithm. In: Proceedings of the ACM international conference on utility-based data mining workshop (UBDM)
19. Manku G, Motwani R (2002) Approximate frequency counts over data streams. In: Proceedings of the 28th International Conference on very large databases (VLDB), pp 346–357
20. Park JS, Chen MS, Yu PS (1997) Using a hash-based method with transaction trimming for mining association rules. IEEE Trans Knowl Data Eng (TKDE) 9(5):813–825
21. Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large database. In: Proceedings of the 21th international conference on very large databases (VLDB), pp 432–444
22. Sun S, Huang Z, Zhong H, Dai D, Liu H, Li J (2009) Efficient monitoring of skyline queries over distributed data streams. Knowl Inf Syst Int J (KAIS). doi:10.1007/s10115-009-0269-0
23. Yang B, Huang H (2010) TOPSIL-Miner: an efficient algorithm for mining top-K significant itemsets over data streams. Knowl Inf Syst Int J (KAIS) 23(2):225–242
24. Yao H, Hamilton HJ, Butz CJ (2004) A foundational approach to mining itemset utilities from databases. In: Proceedings of 4th SIAM international conference on data mining (SDM)
25. Yao H, Hamilton H, Geng L (2006) A unified framework for utility-based measures for mining itemsets. In: Proceedings of the ACM international conference on utility-based data mining workshop (UBDM), pp 28–37
26. Zhu Y, Shasha D (2002) StatStream: statistical monitoring of thousands of data stream in real time. In: Proceedings of the 28th international conference on very large databases (VLDB), pp 358–369

Hua-Fu Li received the Ph.D. degree in Computer Science from National Chiao-Tung University of Taiwan in 2006. He also received his B.S. and the M.S. degrees in Computer Science and Engineering from Tatung Institute of Technology, and in Computer Science from National Chengchi University, in 1999 and 2001, respectively. He is currently an assistant professor at Kainan University in Taiwan. His research interests include stream data mining, web data mining, multimedia data mining, and multimedia systems.

Hsin-Yun Huang received her B.S. and the M.S. degrees in Computer Science from National Chiao-Tung University, Taiwan, in 2005 and 2007, respectively. Her research interests include database systems, data mining and utility pattern mining.

Suh-Yin Lee received the B.S. degree in electrical engineering from National Chiao-Tung University, Taiwan, in 1972, and the M.S. degree in computer science from University of Washington, U.S.A., in 1975, and the Ph.D. degree in computer science form Institute of Electronics, National Chiao-Tung University. She has been a professor in the Department of Computer Science and Information Engineering at National Chiao-Tung University since 1991 and was the chair of that department in 1991–1993. Her research interests include content-based indexing and retrieval, distributed multimedia information system, mobile computing, and data mining.