Research on improvement of high utility pattern mining algorithm over data streams

Feng Guo*, Yuqiang Li and Lin Li
Wuhan University of Technology, Computer Science and Technology, 430061 Wuhan, China

*E-mail: 2284387841@qq.com

Abstract. Aiming at the problem that the existing algorithms for high utility pattern mining over data streams based on sliding window have multiple datasets scans or redundant items, an efficient HUIGRT algorithm for mining high utility patterns over data streams based on global revision header table is proposed in this paper. First, the global revision header table and the utility tree are constructed. The global revision header table is used to store the items and transaction utility of the current data domain that need to be processed, and the utility tree is used to store all of the utility information on the item sets in the transactions to avoid multiple datasets scans. Then, this algorithm can mine all high utility patterns using the global revision header table and the utility tree. Finally, the redundant items are deleted by revising the global revision header table, meanwhile the utility tree is updated to fill in new data. This paper compares the algorithm with the existing high efficiency algorithm HUPMS and HUM-UT on the three datasets with different sparse: Mushroom, T10.I4.D100K and Retail. The results show that the space-time performance of HUIGRT algorithm is better than the two other algorithms.

1. Introduction
With the explosive growth of data, how to extract hidden useful information from massive data has become a major problem in the field of data mining [1]. Frequent pattern mining [2-4] is one of the main technologies in the field of data mining. It can mine patterns from all the datasets that exceed the user-specified threshold. However, frequent pattern mining only considers whether an item in the transaction item set appears, but ignores the nature of the item itself (profit, value, quantity, etc.), resulting in items that don’t appear many times but have high utility may be discarded.

Unlike frequent pattern mining, high utility pattern mining [5] solves the problem that high utility items may be discarded. Each data item has its own internal utility (quantity) and external utility (profit, value, etc.). Yao et al. [6] defined the high utility pattern mining for the first time and gave a specific mathematical calculation model. Liu et al. [7] proposed the two-phase algorithm. However, this algorithm generates a large number of candidates in the first phase of mining, and needs to multiple datasets scans in the second phase, which reduces the efficiency of the algorithm. Subsequently, Wang et al. [8] proposed an algorithm called HUPM-FP and Wu et al. [9] proposed an algorithm called TKU, all does not need to generate candidates, so that all high utility patterns can be mined. The former uses the pattern-growth method to mine high utility patterns, while the latter is used to solve the problem that the user is difficult to set minimum utility threshold in the actual
application process. Although these algorithms are already very efficient, they are only suitable for static datasets. With the emergence of a large number of data streams [10-11] in many fields, traditional high utility mining algorithms based on static datasets are no longer adapt to the processing of data streams. In order to solve this problem, Jin et al. [12] proposed a model for processing data streams, which can be divided into landmark model, sliding window model [13-14] and damped window model. Among them, the sliding window model is a common model for processing data streams. Based on this model, many scholars have carried out in-depth research on high utility pattern mining over data streams.

Ahmed et al. [15] proposed an algorithm over data streams based on sliding window called HUPMS. In the first phase, a large number of candidates are generated and in the second phase, it is necessary to re-verify the candidates, the space-time performance is poor. Subsequently, Wang [16] proposed an algorithm called HUM-UT. This algorithm uses the global header table (Global-HT) to store the current pending data and uses the utility Tree (UT-Tree) to store data transaction information. Through the Global-HT simulation sliding window in the excavation process, all high utility patterns over data streams can be mined through UT-Tree, which avoids the generation of candidates, and its space-time performance is obviously superior to HUPMS algorithm.

Although the efficiency of the high utility mining of the HUM-UT algorithm is already very high, the Global-HT established by the algorithm contains redundant items that are independent of the current data domain to be processed, resulting in a lot of useless time cost to traverses these redundant items for mining process. Aiming at the above problems, this paper proposes an improved high utility pattern mining based on global revision header table algorithm (HUIGRT), which uses a new global revision header table (GRHT-Table) to delete redundant items that are independent of the current data domain to be processed, thereby reducing the extra space-time consumption generated by the algorithm during the mining process. Experimental results on three different datasets show that the HUIGRT algorithm is superior to the HUPMS algorithm and the HUM-UT algorithm in both time and space efficiency. The HUIGRT algorithm will be further explained below.

2. HUIGRT algorithm
For the two-phase algorithm HUPMS, there are multiple datasets scans, and the one-phase algorithm HUM-UT has redundant items in the global header table. This paper proposes the HUIGRT algorithm, whose GRHT-Table contains a window size data, after adding a new batch of data, the redundant items in the GRHT-Table that are independent of the current data domain to be processed are removed, thereby reducing the extra space-time consumption of the algorithm in the mining process. In addition, UT-Tree is used to avoid scanning datasets multiple times and improve the space-time performance of the algorithm.

The basic process of this algorithm is shown in Algorithm 1.

| Algorithm 1 HUIGRT |
|-------------------|
| **Input** Data streams(DS) |
| **Output** high utility patterns of the data streams |
| 0: create a null GRHT-Table and a null UT-Tree |
| 1: while(DS isn’t finish) |
| 2: read a batch of data from DS |
| 3: if(sliding window is filled) |
| 4: if(sliding window is filled for the first time) |
| 5: call procedure Mining high utility patterns of DS |
| 6: else |
| 7: call procedure revision of GRHT-Table and update of UT-Tree |
| 8: call procedure Mining high utility patterns of DS |
| 9: end if |
| 10: else |
| 11: call procedure Construction of GRHT-Table and UT-Tree |
12: end if
13: end while

The HUIGRT algorithm consists of three processes: 2.1. Construction of GRHT-Table and UT-Tree; 2.2. Mining high utility pattern based on GRHT-Table and UT-Tree; 2.3. Revision of GRHT-Table and update of UT-Tree. The three processes will be further explained below.

2.1. Construction of GRHT-Table and UT-Tree

The function of this process is to ensure that the data in sliding window is filled for the first time, so that the current data domain to be processed contains the window size data, so as to realize the first time mining high utility patterns. The entire mining process uses GRHT-Table to store each data item in each batch of transactions and its corresponding transaction weighted utilization. The new batch of transactions is mapped to the tree structure of TN-Tree (Tail Node on Tree), and then multiple TN-Tree is merged into UT-Tree to store the batch utility corresponding to all batch transactions.

GRHT-Table contains different data items and each data item corresponds to the transaction weighted utilization in different batches of data and a list of link pointers, which represent the window size, that is, the number of batches contained in the window; the list of link pointers is created during the mining process and points to the node with the same name in the UT-Tree. The TN-Tree consists of only one root node, multiple general nodes and multiple tail nodes. The general nodes store each data item in the transaction, and the tail nodes not only stores its corresponding one data item, but also stores the utility information of all the data items of the data item to the root node path. When a TN-Tree corresponding to a new batch of data is created, if there have the same prefix data items between different transactions, the node corresponding to the prefix data item is shared on the TN-Tree. When multiple TN-Trees are merged into UT-Tree, the utility information of the batch of data saved by the tail node is merged into the node corresponding to the UT-Tree. At the same time, the tail node on table (TN-Table) is used to store the tail node information of each transaction in each batch of data.

In order to explain the construction process of GRHT-Table and UT-Tree to better, the concepts of utility of data item in transaction, transaction utility and transaction weighted utilization of pattern need to be defined as follows.

Let \( I = \{i_1, i_2, i_3, \ldots, i_n\} \) be a collection of all the different data items, where each data item \( i_r (1 \leq r \leq m) \) has an external utility, denoted as \( p(i_r) \). \( D = \{T_1, T_2, T_3, \ldots, T_n\} \) represents a dataset consisting of \( n \) transactions, each transaction \( T_d \in D (1 \leq d \leq n) \) is a subset of \( I \), and has a unique identifier \( TID \). Each \( i_r \) in \( T_d \) has an internal utility, denoted as \( q(i_r, T_d) \).

Definition 1 (utility of data item in transaction) The utility of data item \( i_r \) in transaction \( T_d \) is the product of the internal utility and the external utility of the data item in \( T_d \), denoted as \( u(i_r, T_d) \), which is defined as follows.

\[
u(i_r, T_d) = q(i_r, T_d) \times p(i_r)
\]  \hfill (1)

Definition 2 (transaction utility) The utility of a transaction \( T_d \) refers to the sum of the utility of all data items in \( T_d \), denoted as \( tu(T_d) \), which is defined as follows.

\[
tu(T_d) = \sum_{i_r \in T_d} u(i_r, T_d) \hfill (2)
\]

Definition 3 (transaction weighted utilization of pattern) The transaction weighted utilization \( twu(T) \) of pattern \( X \) refers to the sum of transaction utility of all transactions containing pattern \( X \) in a dataset \( D \), denoted as \( twu(X) \), which is defined as follows.

\[
twu(X) = \sum_{(X \subseteq T_d \land T_d \in D)} tu(T_d) \hfill (3)
\]

Combined with the definitions (1) to (3), and using the transaction weighted utilization \( twu(X) \) as the estimated-utility, the construction process of GRHT-Table and UT-Tree is shown in Procedure 1.
Procedure 1 Construction of GRTH-Table and UT-Tree

Input a GRHT-Table, a UT-Tree, a TN-Table, a batch of data (BD)

Output a GRHT-Table, a UT-Tree, a TN-Table

0: if (GRHT-Table has included sw batches of data) // sw is the size of window
1: call procedure revision of GRHT-Table and update of UT-Tree
2: else
3: for each (item Qi in BD)
4: calculate the estimated-utility (twu) of Qi, twu(Qi)
5: if (Qi doesn’t exist in GRHT-Table)
6: create a new item for Qi into the GRHT-Table
7: set twu(Qi) as the estimated-utility of Qi into the GRHT-Table
8: end if
9: end for
10: create a TN-Tree for BD, then merge TN-Tree to UT-Tree
11: for each (tail nodes TN on TN-Tree)
12: put TN into the TN-Table
13: end for
14: end if

This paper takes the data in table 1 and table 2 as an example to illustrate the specific execution process of the algorithm in Procedure 1. Table 1 is a pending instance of a dataset containing 8 transactions, and table 2 is the external utility corresponding to the data items in table 1, and all have been sorted according to the dictionary table order.

### Table 1. Pending instance of dataset.

| TID | Transaction item set |
|-----|----------------------|
| T1  | (a,6)(b,1)           |
| T2  | (b,4)(c,1)(e,1)(f,1) |
| T3  | (b,3)(c,6)(d,3)      |
| T4  | (a,1)(b,1)(c,8)      |
| T5  | (b,2)(c,3)           |
| T6  | (a,5)(b,3)(d,4)      |
| T7  | (c,2)                |
| T8  | (a,2)(d,3)           |

### Table 2. External utility of each data item.

| Item | External utility |
|------|------------------|
| a    | 3                |
| b    | 8                |
| c    | 5                |
| d    | 6                |
| e    | 10               |
| f    | 1                |

Set the size of window \( sw = 3 \), the size of batch data \( b = 2 \), that is, the window contains 3 batches of data, and each batch of data contains 2 transactions. Initialize GRHT-Table, UT-Tree, and TN-Table, and read in the first batch of data (composed of transactions T1 and T2). At this time, the GRHT-Table is empty, and the data in the sliding window is not filled. Calculate the \( twu \) of each data item, add non-existent data items to the GRHT-Table in the order of the dictionary table, fill the \( twu \) corresponding to each data item into the corresponding position, and fill the GRHT-Table as table 3 shows.
Table 3. GRTH-Table after reading the first batch of data.

| Item | T[1] | T[2] | T[3] |
|------|------|------|------|
| a    | 26   | null | null |
| b    | 74   | null | null |
| c    | 48   | null | null |
| e    | 48   | null | null |
| f    | 48   | null | null |

Since the first batch of data is read in, the other batch utility in the tail node after the TN-Tree merge are empty, and are marked as empty “{}”. The results as shown in figure 1.

![Figure 1](image1.png)

**Figure 1.** TN-Tree, UT-Tree, and TN-Table after reading the first batch of data.

Next, when read the second batch of data, it has the same prefix data items “ab” and “bc” as the first batch of data; the third batch of data is read, and the other two batches of data have the same prefix data items “ab” and “b”, so UT-Tree shares these nodes. Since TN-Tree is created based on the current batch data, it is not related to the specific mining process, and it will not be separately described later. The GRTH-Table is shown in table 4.

Table 4. GRTH-Table after reading the third batch of data.

| Item | T[1] | T[2] | T[3] |
|------|------|------|------|
| a    | 26   | 51   | 63   |
| b    | 74   | 123  | 109  |
| c    | 48   | 123  | 0    |
| d    | 0    | 72   | 63   |
| e    | 48   | 0    | 46   |
| f    | 48   | 0    | 0    |

The UT-Tree and TN-Table corresponding to table 4 are shown in figure 2.

![Figure 2](image2.png)

**Figure 2.** TN-Tree, UT-Tree, and TN-Table after reading the third batch of data.

Since the third batch of data is filled, and the size of window is 3, the condition that the data in sliding window is filled is satisfied, and the process ends, and then the process of mining high utility patterns is performed.

2.2. Mining high utility patterns based on GRHT-Table and UT-Tree

When the data in sliding window is filled, the mining high utility patterns is started until the data in the window is completely traversed.
In order to explain the process of high utility pattern mining to better, the utility of the pattern in the transaction, the utility of the pattern in a dataset, the total utility of the dataset and the minimum utility need to be defined as follows.

**Definition 4 (the utility of the pattern in the transaction)** The utility of the pattern $X$ in the transaction $T_d$ refers to the utility sum of each data item in $X$ in the transaction $T_d$, denoted as $u(X, T_d)$, which is defined as follows.

$$u(X, T_d) = \begin{cases} 
0 & \text{if } X \not\subseteq T_d \\
\sum_{i \in X} u(i, T_d) & \text{if } X \subseteq T_d 
\end{cases}$$  \hfill (4)

**Definition 5 (the utility of the pattern in a dataset)** The utility of the pattern $X$ in a dataset $D$ is the utility sum of $X$ in all transactions containing $X$ in $D$, denoted as $u_i(X)$, which is defined as follows.

$$u_i(X) = \sum_{T_d \in D} u(X, T_d)$$  \hfill (5)

**Definition 6 (the total utility of the dataset)** The total utility of a dataset $D$ is the sum of the utilities of all the transactions contained in $D$, denoted as $Tu(D)$, which is defined as follows.

$$Tu(D) = \sum_{T_d \in D} u_i(T_d)$$  \hfill (6)

**Definition 7 (minimum utility)** The minimum utility is the product of a user-specified minimum utility threshold ($\text{minUT}$) more than 0 and smaller than 1 and the total utility of dataset $D$, denoted as $\text{minUt}_i$, which is defined as follows.

$$\text{minUt}_i = \text{minUT} \times Tu(D)$$  \hfill (7)

High utility pattern mining refers to: for a dataset $D$, calculate the minimum utility ($\text{minUt}_i$), and mine all the patterns whose utility ($u_i$) is no smaller than $\text{minUt}_i$ from $D$.

The whole mining process is divided into main mining and sub-mining. The mining process is shown in Procedure 2.

**Procedure 2** Mining high utility patterns

**Input** a GRHT-Table, a UT-Tree, $\text{minUT}$

**Output** high utility patterns (HUPs)

1. calculate $\text{minUt}_i$, then create a link list pointer in GRHT-Table for UT-Tree
2. add an attached list (Utility-cache) to each leaf node on UT-Tree
3. for each (item $Q_i$ in GRHT-Table)
4. calculate the estimated-utility ($tw_u$) of $Q_i$, $tw_u(Q_i)$ from GRHT-Table
5. calculate the real utility ($u_i(Q_i)$) of $Q_i$, $u_i(Q_i)$ from Utility-cache
6. if ($tw_u(Q_i) \geq \text{minUt}_i$)
7. generate a data pattern, $X = \{Q_i\}$
8. if ($u_i(Q_i) \geq \text{minUt}_i$)
9. attach $X$ to HUPs
10. end if
11. create a sub global revision header table (SGRHT-Table $X$) for $X$
12. create a sub utility on tail tree (SUT-Tree $X$) for $X$
13. call sub procedure SubMining(SGRHT-Table $X$, SUT-Tree $X$, $X$)
14. end if
15. for each (node $N_i$ corresponding to $Q_i$)
16. pass the Utility-cache of $N_i$ to its parent node on UT-Tree
17. end for

Sub Procedure: SubMining(SGRHT-Table $X$, SUT-Tree $X$, $X$)
18. for each (item $Q_j$ in SGRHT-Table $X$)
19. calculate the estimated-utility($tw_u$) of $\{Q_j\} \cup X$, $tw_u(\{Q_j\} \cup X)$ from
SGRHT-TableX
20: calculate the real utility(uti) of \( \{Q_j\} \cup X \), \( \text{uti}(\{Q_j\} \cup X) \), from Utility-Information
21: if (\( \text{twu}(\{Q_j\} \cup X) \geq \text{min}Uti \))
22: generate a pattern, \( Y = \{Q_j\} \cup X \)
23: if(\( \text{uti}(\{Q_j\} \cup X) \geq \text{min}Uti \))
24: attach \( Y \) to HUPs
25: end if
26: create a sub global revision header table(SGRHT-TableY) for \( Y \)
27: create a sub utility on tail tree (SUT-TreeY) for \( Y \)
28: call procedure SubMining(SGRHT-TableY, SUT-TreeY, \( Y \))
29: end if
30: for each (node \( N_j \) corresponding to \( Q_j \))
31: pass the Utility-Information of \( N_j \) to its parent node on SUT-TreeY
32: end for
33: end for

The link pointer list exists in the last column added to the GRTH-Table, pointing to all nodes of the same name in the UT-Tree. The element of Utility_cache is the sum of the utility of the corresponding data items of each batch of data of the node, that is, the sum of the corresponding values in the three “{}”, which is recorded as “[]”. Since the TN-Table is not used in the current high utility pattern mining process, it will not be separately described later. The UT-Tree that contains the Utility_cache is shown in figure 3.

Figure 3. UT-Tree with Utility_cache attached.

Each data item is processed in turn from the last item in the created GRTH-Table. For example, to process data items “f” and “e” (the grayscale fill has been marked in figure 3), set the minimum utility \( \text{min}Uti = 80 \) and assume that the currently processed data item is the last item “f”. After calculation, \( \text{twu}(f) = 32 + 5 + 10 + 1 = 48 \), \( \text{uti}(f) = 1 \), at this time \( \text{twu}(f) < \text{min}Uti \), the condition is not satisfied. According to the transaction weight downward closure property [7], all the collections containing the data item “f” are not high utility pattern.

Next, the data item “f” corresponding to the Utility_cache on the UT-Tree is passed to the corresponding parent node “e” (the grayscale fill has been marked in figure 4). Before this, you need to remove the utility information of the current processing node. If the parent node does not contain utility information, the Utility_cache is directly passed to the parent node; otherwise, the utility information of the parent node is added to the utility information of the corresponding data item in the Utility_cache. Each time one node is processed, the Utility_cache is passed to the parent node, ensuring that each item in the processing header table has its corresponding Utility_cache, and the resulting UT-Tree is shown in figure 4.
Figure 4. The Utility_cache on the node is passed to the parent node.

As can be seen from figure 4, there are two paths contains node “e”. For the path root-b-c-e, the transaction weighted utilization \( twu \) is the sum of all the values in the Utility_cache on node “e”, which is 47. Similarly, for the path root-b-e, the value of \( twu \) is 46. It is calculated that for the transaction weight value is \( twu(e) = 93 \), the real utility of the node “e” in the sliding window is \( uti(e) = 40 \), because the set \( minUti = 80 \), satisfies \( twu(e) > minUti \), so the generated data pattern \( I = \{e\} \), and because \( uti(e) < minUti \), the data pattern \( I \) is not a high utility pattern. At this time, the current node “e” has a parent node that is not the root node. Therefore, it is necessary to create a SGRHT-Table and a SUT-Tree for data pattern \( I \), and perform sub-mining.

For the creation of SGRHT-Table in sub-mining, the sum \( twu \) of all data items on the path root-b-c-e and path root-b-e is counted. If the sum \( twu \) is no smaller than \( minUti \), the items are saved to a SGRHT-Table in the order of the dictionary table. For the creation of the SUT-Tree, you need to read all the paths and the Utility_cache on the node again. For example, from the two paths obtained in figure 4, the items not in SGRHT-Table are deleted, and the data item “b” is obtained, and \( twu = 93 \). For each of the two paths, the utility of the item “b” is 32 and 16, so the total utility is 48; the utility of the item “e” is 10 and 30, so the total utility is 40. Add data item b to the SUT-Tree, and put 48 on the corresponding to the tail node’s utility list, the result shown in figure 5.

Figure 5. Pattern \( \{e\} \) of SGRHT-TableE and SUT-TreeE.

For the data item “b” in SUT-Tree in figure 5, after calculation, \( twu(\{eb\}) = 93 \), \( uti(\{eb\}) = 40 + 48 = 88 \), since the \( minUti = 80 \), satisfies \( twu(\{eb\}) > minUti \), so the generated data pattern \( Is = \{eb\} \), and since \( uti(\{eb\}) > minUti \), the data pattern \( Is \) is a high utility pattern. Since the node “b” has only one parent and the parent is the root node, the current sub-mining process ends.

2.3. Revision of GRHT-Table and update of UT-Tree

After the first high utility pattern mining, when a new batch of data arrives, the window slides, replacing a batch of data with the longest dwell time with a new batch of data. At this time, the GRHT-Table needs to be revised and the UT-Tree is updated. The specific process is shown in Procedure 3.

Procedure 3 revision of GRHT-Table and update of UT-Tree

Input a GRHT-Table, a UT-Tree, a TN-Table, a batch of data(BD)
Output the latest GRHT-Table, the latest UT-Tree, the latest TN-Table

0: if GRHT-Table has included \( sw \) batches of data) //\( sw \) is the size of window
1: find the longest dwell time batch of data(BD) from GRHT-Table
2: for each(item \( Qi \) in BD)
3: set $Q_i$’s estimated-utility ($twu$) to 0
4: end for
5: delete link list
6: for each (item $Q_i$ in $BD$)
7: calculate the estimated-utility ($twu$) of $Q_i$, $twu(Q_i)$
8: if ($Q_i$ doesn’t exist in $GRHT$-Table)
9: create a new item for $Q_i$ into the $GRHT$-Table
10: end if
11: set $twu(Q_i)$ as the estimated-utility of $Q_i$ into the $GRHT$-Table
12: end for
13: for each (item $Q_j$ in $GRHT$-Table)
14: if (all estimated-utilities of $Q_j$ are 0)
15: delete $Q_j$ from $GRHT$-Table
16: end if
17: end for
18: find tail nodes ($TD$) of the longest dwell time batch of data from $TN$-Table
19: for each (node $N_i$ in $TD$)
20: find the path, $P$, from $N_i$ to root
21: for each (node $N_j$ on the $P$)
22: if ($N_j$ isn’t a shared node)
23: delete $N_j$ from $P$
24: end if
25: end for
26: delete $N_i$ from $TN$-Table
27: end for
28: create a $TN$-Tree for $BD$, then merge $TN$-Tree to $UT$-Tree
29: for each (tail nodes $TN$ on $TN$-Tree)
30: put $TN$ into the $TN$-Table
31: end for
32: end if

In the process of constructing $GRHT$-Table and $UT$-Tree in section 2.1, $GRHT$-Table has read three batches of data, and in section 2.2, the first mining process ends, and then read in the fourth batch of data. When reading the fourth batch of data, find the first batch of data in the $GRHT$-Table with the longest dwell time, set the $twu$ corresponding to all the data items to 0, and then clear the all pointers in the list in the $GRHT$-Table. The data items in the fourth batch of data are arranged according to the dictionary table, and then the data items that don’t exist in the table are added to the $GRHT$-Table, and the $twu$ corresponding to each data item is filled in the original position of the first batch of data, and the $GRHT$-Table is shown in table 5. As can be seen from Table 5, there is a redundancy item “$f$” that is unrelated to the current data field to be processed. Revise the $GRHT$-Table and delete the redundant item so that the algorithm does not traverse the item redundant again during the mining process. The revised $GRHT$-Table is shown in table 6.

**Table 5.** $GRHT$-Table after reading the fourth batch of data.

| Item | T[1] | T[2] | T[3] |
|------|------|------|------|
| a    | 24   | 51   | 63   |
| b    | 0    | 123  | 109  |
| c    | 10   | 123  | 0    |
| d    | 24   | 72   | 63   |
| e    | 0    | 0    | 46   |
| f    | 0    | 0    | 0    |
Table 6. Revised GRTH-Table.

| Item | T[1] | T[2] | T[3] |
|------|-----|-----|-----|
| a    | 24  | 51  | 63  |
| b    | 0   | 123 | 109 |
| c    | 10  | 123 | 0   |
| d    | 24  | 72  | 63  |
| e    | 0   | 0   | 46  |

Through the above process GRTH-Table has been revised, the update process of UT-Tree will be detailed below. Find the tail node “b” and the tail node “f” of the first batch of data from the TN-Table. First, clear the batch utility of tail node “b” corresponding to the UT-Tree. Since the node b has the child node “c”, it indicates that the node on the node “b” to the root path is shared by other batch data, so cannot be deleted. For the tail node “f”, since all batch utility on f are empty and “f” has no child nodes, the node “e” and node “f” that is not shared with other paths is deleted. Next, create a new structure TN-Tree for the fourth batch of data and merge it into UT-Tree. Clear all the tail nodes of the first batch of data in the TN-Table, and fill the tail nodes of the fourth batch of data to the corresponding position. The updated UT-Tree and TN-Table are shown in Figure 6.

Figure 6. Updated UT-Tree and TN-Table.

3. Comparative experiment and result analysis

In order to verify the performance improvement of the HUIGRT algorithm proposed in this paper in space-time. In this paper, compares the algorithm with the existing high efficiency algorithm HUPMS and HUM-UT on the three datasets with different sparse: Mushroom, T10.I4.D100K and Retail.

Three different sparse datasets are derived from the raw datasets of the FIMI website (http://www.cs.rpi.edu/~zaki/Workshops/FIMI/data/) to simulate real data streams, its characterized are shown in Table 7, where I represents the dataset contains the number of different data items, T represents the number of transactions in the dataset, AS represents the average length of the transaction, and DS represents the density of the data.

Table 7. Characteristics of three datasets.

| Dataset   | I    | T    | AS    | DS    |
|-----------|------|------|-------|-------|
| Mushroom  | 119  | 8124 | 23    | 19.33%|
| T10.I4.D100K | 1000 | 100000 | 10    | 1%    |
| Retail    | 16470| 88162| 10.3  | 0.06% |

Since the data items in the raw dataset lack internal utility and external utility, the generation method in [17] is used, and the internal utility of the items in the transaction item set are randomly generated from 0 to 10; The external utility is randomly generated from a logarithmic positive distribution between 0.01 and 10. In this paper, different minUT are used as the only change conditions, and the runtime and memory consumption are used as evaluation criteria. The experiment is implemented in C# language, the compiler is Visual Studio 2013, the hardware platform is Intel dual-core 2.60GHz, 12G memory, and the operating system is Windows7. Specific experiments are described below.
3.1. Runtime comparison experiment on different datasets
In order to mining high utility patterns for the all three algorithms on a given dataset, first you need to set a suitable minimum utility threshold (minUT) for the given dataset. If the minUT setting is too large, the dataset under the threshold does not contains the high utility patterns, and if the minUT setting is too small, it will cause the mining time to be too long. For the data streams simulated by the three datasets, set the size of window sw= 3, the size of batch data b = 200, use three algorithms respectively, and set different minUT for experiments. The runtime comparison results are shown in (A), (B) and (C) in figure 7.

![Figure 7. Runtime on different datasets.](image)

The specific runtime are shown in table 8, table 9, and table 10.

| Table 8. Specific runtime on mushroom. |
|---------------------------------------|
| minUT/| HUPMS | HUM-UT | HUIGRT |
|-----|-------|--------|--------|
| 9   | 681   | 48     | 40     |
| 8   | 815   | 52     | 42     |
| 7   | 956   | 64     | 51     |
| 6   | 1357  | 164    | 132    |
| 5   | 1787  | 321    | 281    |

| Table 9. Specific runtime on T10.14.D100K. |
|------------------------------------------|
| minUT/| HUPMS | HUM-UT | HUIGRT |
|-----|-------|--------|--------|
| 0.6 | 198   | 93     | 72     |
| 0.5 | 289   | 98     | 74     |
| 0.4 | 412   | 117    | 93     |
| 0.3 | 596   | 146    | 129    |
| 0.2 | 856   | 367    | 305    |

| Table 10. Specific runtime on retail. |
|--------------------------------------|
| minUT/| HUPMS | HUM-UT | HUIGRT |
|-----|-------|--------|--------|
| 2.2 | 861   | 291    | 117    |
| 2.1 | 955   | 306    | 125    |
| 2.0 | 1251  | 323    | 127    |
| 1.9 | 1596  | 380    | 213    |
| 1.8 | 2150  | 898    | 743    |

It can be seen from figure 7 that the HUIGRT and HUM-UT algorithm has the least runtime compared with HUPMS, this is because HUPMS needs to generate a large number of candidates at the beginning and then mine high utility patterns from the candidates, but HUIGRT and HUM-UT directly mine high utility patterns from UT-Tree to avoid generating candidates, thereby reducing the runtime of the algorithm. From the runtime curves of HUIGRT and HUM-UT, the proposed of HUIGRT algorithm has less runtime compared with HUM-UT. This is because the algorithm HUM-UT contains extraneous redundancy items in the mining process. Traversing these redundant items will generate additional time consumption, while HUIGRT removes irrelevant redundancy items, thus reducing the runtime of the algorithm.
Combined with the experimental data in table 8, table 9, and table 10, as the density of the dataset decreases, HUIGRT is more efficient in improving the runtime than the HUM-UT. On the densest Mushroom, HUIGRT has a significant improvement in runtime compared to HUM-UT, and when \( \text{minUT} = 6\% \), the runtime is the most efficient, improved by 19\%; On the T10.14.D100K with the second highest density, when \( \text{minUT} = 0.5\% \), HUIGRT has the highest efficiency in improving the runtime of HUM-UT, improved by 25\%. On the lowest density Retail, the best contrast is achieved when \( \text{minUT} = 2.0\% \), improved by 61\%. This is because the distribution of data items in a data streams transaction with a lower density is more dispersed. For the algorithm HUM-UT, the more redundant items in the data field to be processed, and the HUIGRT algorithm on the GRHT-Table, the more redundant items are deleted after the revision, reduce the more time it takes to traverse these redundant items. During the experiment, the data streams simulated by the Mushroom, and the redundancy items deleted by the algorithm HUIGRT after the GRHT-Table revision is 56; the T10.14.D100K deleted by the algorithm HUIGRT after the GRHT-Table revision reached 11817; for the Retail, the algorithm HUIGRT deleted the most redundant items, which was 213483, so the time efficiency improvement was the most obvious.

In addition, as can be seen from figure 7, as the \( \text{minUT} \) is continuously decreasing, the runtime of the three algorithms is increasing, because the number of patterns whose utility is more than the \( \text{minUT} \) will increase. You need to create more SUT-Tree and perform more mining operations, so the runtime will increase as the \( \text{minUT} \) decreases.

### 3.2. Memory consumption comparison on different datasets

In order to verify that the HUIGRT algorithm is superior in memory efficiency to the HUPMS algorithm and the HUM-UT algorithm. Next, the experimental parameters in section 3.1 are used to ensure that \( s^w = 3 \) and \( b = 200 \) are unchanged, and the memory consumption of the three algorithms on different datasets is recorded. The experimental results are shown in (A), (B) and (C) in figure 8.

![Figure 8. Memory consumption on different datasets.](image)

The specific memory consumption are shown in table 11, table 12 and table 13.

| \( \text{minUT}\% \) | HUPMS | HUM-UT | HUIGRT |
|-------------------|-------|--------|--------|
| 9                 | 125   | 20     | 18     |
| 8                 | 131   | 23     | 20     |
| 7                 | 147   | 25     | 22     |
| 6                 | 168   | 26     | 23     |
| 5                 | 191   | 28     | 24     |

| \( \text{minUT}\% \) | HUPMS | HUM-UT | HUIGRT |
|-------------------|-------|--------|--------|
| 0.6               | 198   | 43     | 34     |
| 0.5               | 218   | 48     | 36     |
| 0.4               | 235   | 52     | 37     |
| 0.3               | 247   | 61     | 49     |
| 0.2               | 258   | 63     | 50     |
Table 13. Specific memory on retail.

| minUT/% | HUPMS | HUM-UT | HUIGR | T
|---------|-------|--------|-------|
| 2.2     | 351   | 61     | 39    |
| 2.1     | 424   | 62     | 40    |
| 2.0     | 478   | 65     | 42    |
| 1.9     | 507   | 67     | 43    |
| 1.8     | 650   | 77     | 46    |

It can be seen from figure 8 that the HUIGR and HUM-UT algorithm consumes the least memory compared to HUPMS, this is because the algorithm HUIGR uses the established UT-Tree to get the real twu. When the minimum utility is more than the real twu, it is not need to create a SUT-Tree. The two-phase algorithm HUPMS uses overestimated twu, which are often more than the real twu. HUPMS creates more trees than HUIGR, resulting in HUPMS consuming more memory than HUIGR. From the memory consumption curves of HUIGR and HUM-UT, the proposed of HUIGR algorithm has less memory consumption compared with HUM-UT. This is because during the mining process, the global header table of the HUM-UT contains extraneous redundancy items. The storage of these redundant items requires additional memory, while the HUIGR removes extraneous redundancy items, thereby reducing the memory consumption of the algorithm.

Combined with the experimental data in table 11, table 12 and table 13, it can be seen that HUIGR is more efficient in memory efficiency than HUM-UT as the dataset density decreases. On the densest Mushroom, HUIGR has a significant improvement in memory efficiency compared to HUM-UT, and the memory efficiency is maximized when \( \text{minUT} = 5\% \), the memory efficiency is improved by 14%. On the T10.14.D100K with the second highest density, when \( \text{minUT} = 0.4\% \), HUIGR has a significant improvement in memory efficiency compared to HUM-UT, the memory efficiency is improved by 29%. On the lowest density Retail, when \( \text{minUT} = 1.8\% \), the best contrast is achieved, the memory efficiency is improved by 40%. This is because for the dataset with lower density, the more redundant items are included in the algorithm HUM-UT, and the more redundant items are deleted after the algorithm HUIGR modifies the GRHT-Table, reducing The redundancy memory will also occupy more memory, so the memory efficiency is more obvious.

In addition, as can be seen from figure 8, as the minUT is continuously decreasing, the memory consumption of the HUPMS algorithm will increase rapidly. This is mainly because the HUPMS algorithm uses the overestimation of the twu to determine whether to create a SUT-Tree. Therefore, in the process of decreasing the minUT, the total number of trees that the HUPMS algorithm far more than one-phase algorithm HUIGR and HUM-UT, so the memory consumption is far more than the algorithms HUIGR and HUM-UT.

4. Conclusion
Aiming at the problem that the existing high utility pattern mining algorithm has a large number of unrelated redundancy items, resulting in low space-time performance, this paper proposes HUIGR algorithm based on GRHT-Table. HUIGR adopts a new GRHT-Table stores the data items and transaction utility in the current pending data domain, avoiding the extra time consuming due to window sliding. The comparison experiments on three different sparse datasets of Mushroom, T10.14.D100K and Retail show that the HUIGR algorithm proposed in this paper is better than the existing high efficiency algorithms HUPMS and HUM-UT.

Acknowledgments
This work was supported by the National Social Science Foundation of China (15BGL048).

References
[1] Z. N. Feng, Y. N. Zhu. A Survey on Trajectory Data Mining: Techniques and Applications. IEEE Access, 4, 2056-67(2017)
[2] G. Pyun, U. Yun, K. H. Ryu. Efficient frequent pattern mining based on Linear Prefix tree. Knowledge-Based Systems, 55, 125-39(2014)

[3] P. Claude, S. Jérémie, F. Frédéric, et al. Frequent pattern mining in attributed trees: algorithms and applications. Knowledge & Information Systems, 46(3), 491-514(2016)

[4] A. Khubaib, A. Rodina, K. Kwangman. Enabling far-edge analytics: performance profiling of frequent pattern mining algorithms. IEEE Access, 5, 8236-49(2017)

[5] Zida, Souleymane, Fournier-Viger, et al. EFIM: a fast and memory efficient algorithm for high-utility itemset mining. Knowledge & Information Systems, 51(2), 1-31(2017)

[6] H. Yao, H. Hamilton, C. Butz. A foundational approach to mining itemset utilities from databases. Proceedings of the 2004 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 482-86(2004)

[7] L. Ying, W. K. Liao, A. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining. Springer-Verlag, 689-95(2005)

[8] L. Wang, S. Q. Xiong, Y. F. Chang, et al. An Algorithm for Mining High Utility Patterns Based on Pattern-growth. ACTA AUTOMATICA SINICA, 41(9), 1616-26(2015)

[9] W. W. Wu, B. E. Shie, P. S. Yu, et al. Mining top-K high utility itemsets. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 78-86(2012)

[10] H. Li, H. Q. Chen, Q. Y. Yang, et al. Real-time Visualization Method for Log Data streams. Journal of System Simulation, 28(9), 2133-37(2016)

[11] H. F. Li, H. Y. Hung, and S. Y. Lee. Fast and memory efficient mining of high-utility itemsets from Data streams: with and without negative item profits. Knowledge & Information Systems, 28(3), 495-522(2011)

[12] C. Q. Jin, W. N Qian, A. Y Zhou. Analysis and Management of Streaming Data: A Survey. Journal of Software, 15(8), 1172-81(2004)

[13] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, et al. Efficient Mining of High Utility Patterns over Data streams with a Sliding Window Method. Studies in Computational Intelligence, 295, 99-113(2010).

[14] H Ryang, and U. Yun. High utility pattern mining over Data streams with sliding window technique”, Expert Systems with Applications, 57(C), 214-31(2016)

[15] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, et al. Choi. Interactive mining of high utility patterns over Data streams. Expert Systems with Applications, 39(15), 11979-91(2012)

[16] L. Wang. Pattern Mining Algorithms over Data streams. Dalian: Dalian University of Technology(2013)

[17] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, et al. Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases. IEEE Transactions on Knowledge & Data Engineering, 21(12), 1708-21(2009)