Article

Mining High Utility Itemsets Based on Pattern Growth without Candidate Generation

Yiwei Liu ^{1,2}, Le Wang ³, Lin Feng ^{1,2} and Bo Jin ^{1,2,*}

¹ School of Computer Science and Technology, Dalian University of Technology, No.2 Linggong Road, Ganjingzi District, Dalian 116024, China; liuyiwei@dlut.edu.cn (Y.L.); fenglin@dlut.edu.cn (L.F.)

² School of Innovation and Entrepreneurship, Dalian University of Technology, No.2 Linggong Road, Ganjingzi District, Dalian 116024, China;

³ College of Digital Technology and Engineering, Ningbo University of Finance and Economics, 899 Xueyuan Road, Haishu District, Ningbo 315175, China; lelewater@nbufe.edu.cn

* Correspondence: jinbo@dlut.edu.cn; Tel.: +86-13084100305

Abstract: Mining high utility itemsets (HUIs) has been an active research topic in data mining in recent years. Existing HUI mining algorithms typically take two steps: generating candidates and identifying utility values of these candidate itemsets. The performance of these algorithms depends on the efficiency of both steps, both of which are usually time-consuming. In this study, we propose an efficient pattern-growth based HUI mining algorithm, called tail-node tree-based high-utility itemset (TNT-HUI) mining. This algorithm avoids the time-consuming candidate generation step, as well as the need of scanning the original dataset multiple times for exact utility values, as supported by a novel tree structure, named the tail-node tree (TN-Tree). The performance of TNT-HUI was evaluated in comparison with state-of-the-art benchmark methods on different datasets. Experimental results showed that TNT-HUI outperformed benchmark algorithms in both execution time and memory use by orders of magnitude. The performance gap is larger for denser datasets and lower thresholds.

Keywords: utility mining; tail-node tree; efficient algorithm; market-basket analysis

Citation: Liu, Y.; Wang, L.; Feng, L.; Jin, B. Mining High Utility Itemsets Based on Pattern Growth without Candidate Generation. *Mathematics* **2021**, *9*, 35. <https://dx.doi.org/10.3390/math9010035>

Received: 2 October 2020

Accepted: 18 December 2020

Published: 25 December 2020

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Pattern discovery from a transactional database has been an important topic in data mining [1,2]. Since the development of the Apriori algorithm for mining frequent itemsets [3], newer algorithms [4–10] have been continually proposed for various formulations and performance enhancements. Traditional frequent itemset mining treats each item in a transaction as binary. In other words, each itemset either occurs or does not occur in a transaction. However, in the real-world, real-valued quantities, such as profit, may be practically important. The unit profit and purchase quantity of items are vital for finding the most valuable itemsets that contribute most to the overall profit. To address this problem, mining high utility itemsets (HUIs) was proposed and studied extensively in the data mining literature [11–18].

Existing HUI mining algorithms typically take two steps: generating candidates and identifying utility values of these candidate itemsets. The performance bottleneck of these algorithms has primarily been in the candidate generation process; thus, lots of research efforts have been focused on improving this step by reducing the number of candidates or avoiding the candidate generation step completely. For example, a number of algorithms that generate candidate itemsets based on the apriori method [19–22] or map the transaction itemsets to utility lists [14,15] may generate non-existing candidate itemsets (i.e., itemset that may co-occur but never co-occurred in the dataset), which results in unnecessary computing that negatively impacts the computational performance. In contrast, pattern-growth-based algorithms [11,16,17] can avoid generating non-existing candidates; thus, they are promising for superior computation performance in HUI mining.

Although the pattern-growth approach can effectively exclude non-existing itemsets from candidate itemsets, they still need to generate candidate itemsets and require additional scans of the original dataset to calculate the exact utility value of these candidates and identify HUIs. The reason is that, after mapping transaction itemsets to a tree structure, they use an overestimated utility value to generate candidate itemsets. Unlike in frequent pattern mining, the downward closure property of the support measure is no longer applicable in HUI mining and cannot be used for effectively removing low utility patterns from the candidates. Therefore, using an overestimated utility value that has computation friendly downward closure property has been a commonly adopted strategy in HUI mining [19–26]. The greater overestimation, the more candidates will be produced and, thus, the less efficiency. Without the ability of directly retrieving the exact utility values from the tree, existing pattern-growth-based HUI mining methods need to scan the original dataset to identify HUIs, which required additional passes of data I/O, resulting in much computation overhead. To address this issue, we propose a novel tree structure, called tail-node tree (TN-Tree), from which we can retrieve the exact utility values without re-scanning the original dataset. The basic idea is that, when mapping the transaction itemsets to the tree, we maintain the utility of each individual item in the itemset in a special node, called the tail node. Correspondingly, a tail-node tree-based HUIs mining algorithm, named tail-node tree-based high-utility itemset (TNT-HUI), is proposed for discovering HUIs efficiently. With this simple enhancement, our TNT-HUI algorithm can find all HUIs efficiently through only two scans of the dataset. Experimental results with both dense and sparse datasets also verified the effectiveness of the proposed method.

Our contributions may be summarized as follows. First, we designed a novel tree structure in which the tail nodes are used to store item-specific utility information, so that the exact utility value of an itemset may be easily retrieved later. Second, based on the pattern-growth method, we designed an HUI mining algorithm that can avoid the need of generating candidate itemsets. Finally, our experiments with both sparse and dense datasets, including synthetic and real data, demonstrated that the performance of the proposed algorithm outperformed the state-of-the-art algorithms.

The rest of this paper is organized as follows. Section 2 describes related work for HUI mining. Section 3 describes the background. Section 4 describes our proposed data structure TN-Tree. Section 5 describes the proposed algorithm TNT-HUI. Section 6 reports our experimental results. Section 7 draws the conclusions and points out possible future work.

2. Related Work

Existing HUI mining algorithms may be categorized into two groups: apriori-based and pattern-growth-based.

2.1. Apriori-Based HUI Mining Algorithms

Yao et al. proposed the mathematical model for mining HUIs [22]. The authors estimated an expected utility value to determine whether an itemset should be a candidate itemset for high utility itemsets. However, the number of candidates may approach the number of all the combinations of items if the minimum utility value is very small and a dataset contains many distinct items, so the mining process might be time-consuming. Later, Yao et al. proposed two new algorithms for mining HUIs: UMining and UMining_H algorithms [21]. The algorithm UMining employs the utility upper bound property for pruning. The algorithm UMining_H employs a heuristic method for pruning. These two algorithms may prune some HUIs, as well as suffer from excessive candidates.

Liu et al. proposed the algorithm Two-Phase [20] for mining HUIs. The authors firstly proposed the transaction-weighted-utilization (twu) model. The model maintains a twu downward closure property. In this model, an itemset can be considered as a candidate itemset for HUIs if its twu value is not less than a minimum utility value. The algorithm Two-Phase consists of two phases. In the first phase, Two-Phase finds all the candidate

itemsets. In the second phase, the algorithm discovers the actual HUIs from the candidate itemsets by an additional dataset scan. In the first dataset scan, the algorithm finds all the 1-element candidate itemsets and, based on that, generates the 2-element itemsets. In the second dataset scan, the algorithm finds all the 2-element candidate itemsets from these 2-element itemsets, uses this information to generate the 3-element candidates, and so on. This algorithm outperforms the algorithm proposed in Reference [22]. However, this algorithm still generates too many candidates in the first phase and needs multiple scans of a dataset.

To reduce the number of candidates in the first phase of the algorithm Two-Phase, Li et al. proposed an isolated items discarding strategy (IIDS) to reduce the number of candidates and applied the strategy to the two existing algorithms, and get two new algorithms renamed FUM and DCG+ [19]. The performance of these two new algorithms outperforms their original algorithms. Although IIDS effectively reduces candidates, it still scans dataset multiple times and generates candidate itemsets for HUIs.

Apriori-like algorithms generate candidate itemsets layer-by-layer, and then they scan the original dataset to get the utility value of the candidates. Their main shortcoming is that they generate a large number of candidates, including non-existing itemsets from dataset, and they need multiple scans on the original dataset.

2.2. Pattern-Growth-Based HUI Mining Algorithms

Pattern-growth-based algorithms avoid multiple scans of the dataset and reduce, as well as taking those itemsets that are not in the dataset as candidate itemset. HUP-Growth [25] creates HUP-Tree in a way like FP-Tree. When mapping a transaction itemset to a tree, it stores the utility values of this node, as well as the node's ancestors into a list (this list is called "utility list"). If the node's utility list already exists, the itemset's utility values are added up to the list. This way the utility values of all possible itemsets of the dataset can be calculated from the tree. HUP-Growth takes a bottom-up approach to process each item, collecting items along the path, generates all possible combinations containing this item, and calculates their utility values, thereby determining all HUIs for current item. The merit of this algorithm is that utility values of itemsets can be calculated efficiently from the tree. But, it still generates too many candidate itemsets.

Algorithm IHUP [11] also adopts FP-Tree's approach to create IHUP-Tree. When it maps a transaction itemset to the tree, the utility value of this transaction itemset is stored on each node of this itemset. If the node already contains a utility value, a new value is simply added to it. IHUP utilizes pattern-growth approach (FP-Growth method [6]) to generate a candidate itemset and uses the sum of all utility values of the corresponding nodes of the current item as the over-estimated threshold to determine whether this itemset is a promising candidate. Compared with HUP-Growth's approach that combines items along the path to get candidate itemsets, IHUP's candidates are lessor, and the mining efficiency is increased.

IHUP cannot retrieve an itemset's utility value directly after it maps the transaction itemset to a tree. Instead, it gets the sum of all utility values of the transactions containing this itemset (or over-estimated utility value). Therefore, it needs to scan the original dataset to calculate candidates' utility after these candidates are generated. Algorithm UP-Growth [16] is an improvement of IHUP. When it maps a transaction itemset to a tree, it registers the utility values of the corresponding node and this node's ancestors in the transaction. If a node has already registered with a utility value, the algorithm just adds the new value to it. Sub-trees are constructed by the same way, i.e., each node does not contain utility values of its children nodes. So, UP-Growth's over-estimated utility value (used for judging whether an itemset is a candidate) is lower than that of IHUP. This effectively reduces the number of candidate and improves the time efficiency of the judging of candidates.

The pattern-growth-based algorithms mentioned above discard utility values of individual item of a transaction. They cannot retrieve the exact utility value of an itemset

and must utilize an over-estimated utility value to generate candidate itemsets. It is obvious that the smaller the over-estimated utility threshold is, the lesser the candidates will be, and the better the performance of the mining algorithm may achieve. If we can get the exact utility value of an itemset, we can identify directly whether it is an HUI without bothering the processing of candidates. Based on this reasoning, we propose to construct a novel tree structure such that, after mapping transaction itemsets to the tree, the itemsets' exact utility values can be retrieved from the tree. In summary, our study adopts this pattern-growth approach to mine HUIs without generating candidate itemsets.

3. Preliminaries

In this section, we give the definition of the HUI mining.

3.1. Basic Concepts

Given a set of m unique items $I = \{i_1, i_2, \dots, i_m\}$, an itemset $X \subseteq I$ containing k distinct items is called a k -itemset. A transaction dataset $DB = \{T_1, T_2, \dots, T_n\}$ contains n transactions. Each transaction T_d ($d = 1, 2, \dots, n$) involves a subset of all unique items in I , called a transaction itemset. For convenience, we use the notation T_d represent the transaction itemset.

For a utility-valued transaction database, each item i_r ($r = 1, 2, \dots, m$) has a unit profit $p(i_r) \in \mathbb{R}$, and each item i_r in a transaction T_d is attached with a quantity $q(i_r, T_d) \in \mathbb{R}$ with its occurrence in the transaction (e.g., quantity purchased, dollar amount paid, or profit from the transaction).

Definition 1 (Item Utility). The utility of the item i_r in a transaction T_d is denoted as $u(i_r, T_d)$ and calculated as

$$u(i_r, T_d) = p(i_r) * q(i_r, T_d), \quad (1)$$

where $p(i_r)$ is the unit profit of item i_r , and $q(i_r, T_d)$ is the quantity of item i_r 's occurrence in transaction T_d , $\forall i = 1, 2, \dots, m, \forall d = 1, 2, \dots, n$.

Definition 2 (Itemset Utility). The utility of an itemset X in a transaction T_d is denoted as $u(X, T_d)$ and defined as

$$u(X, T_d) = \begin{cases} 0, & \text{if } X \not\subseteq T_d; \\ \sum_{i_r \in X} u(i_r, T_d), & \text{if } X \subseteq T_d; \end{cases} \quad (2)$$

where $u(i_r, T_d)$ is the utility of the item i_r in transaction T_d . The utility of the itemset X in the whole transaction dataset $DB = \{T_1, T_2, \dots, T_n\}$ is denoted as $u(X)$ and defined by

$$u(X) = \sum_{T_d \in DB} u(X, T_d). \quad (3)$$

Since a transaction corresponds to a transaction itemset, the transaction utility is a special case of itemset utility. More specifically, the utility of a transaction T_d is denoted as $tu(T_d)$ and defined by

$$tu(T_d) = \sum_{i_r \in T_d} u(i_r, T_d). \quad (4)$$

Definition 3 (Support Number). The support number (sn) of an itemset X is the number of transaction itemsets containing X .

Definition 4 (Transaction-Weighted Utility). The transaction-weighted utility of an itemset X is denoted as $twu(X)$, and is defined by

$$twu(X) = \sum_{T_d \in \{T \in DB: X \subseteq T\}} tu(T_d). \quad (5)$$

$twu(X)$ is the sum of the transaction utilities of all transaction itemsets containing X .

Example 1 (Utility-Valued Transaction Database). The first two columns in Table 1 and the first two columns in Table 2 provide an example utility-valued transaction database. More specifically, Table 1 is a dataset containing 7 transaction itemsets, and Table 2 shows the unit profit value of each item in Table 1.

Table 1. An example database.

TID	Items and Quantities	$tu(T_i)$	$u(B, T_i)$	$u(C, T_i)$	$u(\{B, C\}, T_i)$
T_1	(B, 4)(C, 3)(D, 3)(E, 1)	24	12	3	15
T_2	(B, 2)(C, 2)(E, 1)(G, 4)	15	6	2	8
T_3	(B, 3)(C, 4)	13	9	4	13
T_4	(A, 1)(C, 1)(D, 2)	15	0	1	0
T_5	(A, 2)(B, 2)(C, 2)(D, 2)(E, 1)(F, 9)	44	6	2	8
T_6	(A, 1)(C, 6)(D, 2)(E, 1)(G, 8)	31	0	6	0
T_7	(A, 2)(C, 4)(D, 3)	30	0	4	0

Table 2. Profit table.

Item	Profit	twu	sn
A	10	120	4
B	3	96	4
C	1	172	7
D	2	144	5
E	3	114	4
F	1	44	1
G	1	46	2

Definition 5 (Promising Itemset). An itemset/item X is called a promising itemset/item for high utility itemsets/item if $twu(X) \geq \min_uti$; otherwise, it is an unpromising itemset/item. A promising itemset is also called a candidate itemset for HUIs.

Lemma 1 (Transaction-Weighted Downward Closure Property). Any subset of a promising itemset is a promising itemset, and any superset of an unpromising itemset is an unpromising itemset.

Lemma 1 has been proved in Reference [20]. For example, if $\{ACD\}$ is a promising itemset, the itemset $\{AC\}$ (or any sub itemset of $\{ACD\}$) is also a promising itemset. On the other hand, if $\{AC\}$ is unpromising, all its super itemsets (such as $\{ACD\}$) are unpromising.

Theorem 1. Let item Q be an unpromising item in dataset DB ; then, any itemset X containing Q is not a high utility itemset [16].

Proof. According to Lemma 1, itemset X is an unpromising itemset. According to Definition 2 and Definition 4, $u(X) \leq twu(X)$, the utility of itemset X is less than the minimum utility value; thus, itemset X is not an HUI. $\square$

3.2. Problem Definition

In a transaction dataset, an itemset is a high utility itemset if its utility is not less than a user-specified minimum utility value, where the utility of an item in a transaction is defined as its internal utility multiplied by its external utility. The utility of an itemset in a transaction is defined as the sum of its all items' utility in the transaction. For example, the utility of an itemset X in a transaction dataset is defined as the sum of its utility in each transaction containing X .

Definition 6 (High Utility Itemset). An itemset X is called a high utility itemset if its utility ($u(X)$) is not less than a user-specified minimum utility value.

Given a transaction database DB , the problem of HUI mining aims at finding all HUIs. Mining HUIs from the databases refers to finding all itemsets in which utility is not less than the minimum utility value.

4. TN-Tree for HUI Mining

To facilitate the mining process and avoid scanning the dataset many times, a tree structure is employed to maintain the dataset in our algorithm. In this section, we firstly introduce a new tree structure called TN-Tree to maintain a transaction dataset, and then we describe the algorithm of mining HUIs from the TN-Tree.

4.1. The Structure of TN-Tree

In this study, we propose TN-Tree, a new data structure to store critical information from the dataset for HUI mining. TN-Tree can be used to store the utility values of itemsets, which can be used to determine whether an itemset is an HUI.

Like other tree structures for pattern generation, in a TN-tree, each node N contains the following fields:

- $N.name$ records item name of the node N ,
- $N.parent$ records parent node of the node N , and
- $N.children$ records a set of the children nodes of N .

Moreover, a subset of nodes in the TN-tree are called *tail-nodes*. Each tail node and all nodes on its path to the root node correspond to an itemset X . A tail node contains the following fields, in addition:

- $N.sn$ is the number of path-itemset of the node N ;
- $N.piu$ is a list which records path-itemset utility;
- $N.su$ is the path-utility of all elements in $N.piu$;
- $N.bu$ is the utility of the base-itemset.

$N.sn$, $N.bu$, $N.piu$, and $N.su$ are called the *tail-information* of node N . The tail-information is important because the utility value of any itemset can be calculated according to its tail-information of TN-Tree.

The TN-Tree structure maintains all candidate itemset information. In other words, all itemsets that potentially have a utility score above the minimum utility threshold can be found by using information stored on the tree. Each tail node corresponds to an itemset that consists of all items on its path to the root. Each tail node stores piu , the utility of each item in the itemset, and su , the total utility of the set.

Figure 1 illustrates an example TN-Tree, which was constructed based on data in Tables 1 and 2. For example, the leftmost node B is a tail-node of the itemset $\{C, D, E, B\}$, and the sequence of numbers, “3, 6, 3, 12” on the node, represent the total utility values of items C , D , E , and B in the entire dataset, respectively.

Figure 1. Example tail-node tree (TN-Tree) constructed from the toy data in Tables 1 and 2.

4.2. TN-Tree Construction

A TN-Tree can be constructed by two scans of a transaction dataset. The pseudocode is provided in Algorithm 1.

Algorithm 1: TN-tree Construction

```

Input :DB: transactions data;
         $\eta$ : minimum threshold for staying on the header table.
Output :T: the TN tree.
// First scan of the database DB
1 Initiate a header table  $H$  containing the fields of item ID,  $twu$ ,  $sn$ , and links;
2 for each transaction  $TS$  of DB do
3   for each item  $I$  in  $TS$  do
4      $H \mapsto I.sn++$ ;
5      $H \mapsto I.twu+ = tu(TS)$ ;
6   end
7 end
8 Delete unpromising items from  $H$  based on threshold  $\eta$ ;
9 Sort  $H$  according in the descending order of  $sn$  or  $twu$  of  $H$ ;
// Second scan of the database DB
10 Initialize a TN-Tree  $T$  with an empty root node;
11 for each transaction  $TS$  of DB do
12   Delete unpromising items from  $TS$ ;
13   Sort items of  $TS$  according to  $H$ , with utility values, to  $X$ ;
14   Insert  $X$  to  $T$ ;
// Process the tail information for the tail node
15    $N.sn++$ ;
16    $N.piu+ = X.piu$ ; // element-by-element addition
17    $N.su+ = X.su$ ;
18 end

```

In the first scan, we create a header table. We first collect compute the twu value and support number of each unique item in the dataset. The items of the header table were then arranged in the descending order of support numbers (or twu values). Unpromising items are then deleted from the header table.

In the second scan, transaction itemsets are added into the TN-Tree. The TN-Tree was first initialized as an empty root node (i.e., its parent node and item name are null). For each transaction in the dataset, we take the following process. First, we delete unpromising items from the transaction itemset (Line 12). Then, we sort the remaining promising items according to their ordering in the header table and create a sorted itemset X (Line 13). Next, we add itemset X into the TN-Tree, and store the number of itemset X , the utility of each item in X , and the sum utility of all items in X to the tail-information of the tail-node of X on the TN-Tree. and link the new node of TN-Tree to the corresponding item in the header table (Line 14).

Note that the field bu on each tail-node is initialized as 0 in this (global) TN-Tree. Its value will be updated in the HUI mining process when subtrees are constructed (see Section 5.2).

Example 2 illustrates the construction process of a TN-Tree using the dataset in Tables 1 and 2.

Example 2 (TN-Tree Construction). Suppose the minimum utility value min_uti is 70.

Firstly, a header table H is created by one scan of the dataset. The result is shown in Figure 2a.

Figure 2. Construction of a TN-Tree.

Then, a TN-Tree is initialized as a root node in which its parent node and item name are null. A second scan of the dataset will add all transactions to the TN-Tree by the following process.

1. For the first transaction itemset $\{B, C, D, E\}$, we remove unpromising items from the itemset and sort items of the itemset according to the order of H. Then, we get the itemset $\{C, D, E, B\}$, add the itemset to a TN-Tree, and store the *piu* values $(3, 6, 3, 12)$ to the field *piu* on the tail-node. The TN-Tree is shown in Figure 2b after T_1 is added to the tree, where node B is a tail-node, and $1; 0; 3, 6, 3, 12; 24$ shows its *sn* = 1, *bu* = 0, *piu* = $\{3, 6, 3, 12\}$, and *su* = 24 (i.e., $3 + 6 + 3 + 12$).
2. For the second transaction itemset $\{B, C, E, G\}$, we remove unpromising item G, sort items of the itemset according to the order of H, and get the itemset $\{C, E, B\}$. Figure 2c shows the TN-Tree after T_2 was added to the tree.
3. For the third transaction itemset $\{B, C\}$, we obtained the sorted itemset $\{C, B\}$. Figure 2d shows the TN-Tree after T_3 was added to the tree.
4. By the above method, the first six transactions were added to the TN-Tree. The result is shown in Figure 2e.
5. After all the transaction itemsets were added to the tree, the TN-Tree was shown as Figure 1. When T_7 was added to the tree, there was a path “root-C-D-A” on the TN-Tree and node A was also the tail-node in Figure 2e. Therefore, we just needed to modify the tail-information on the tail-node A. The modified tail-information is shown in Figure 1.

5. Mining HUIs from A TN-Tree

In this section, we first introduce the concepts about sub-tree, describe, and analyze the proposed algorithm.

5.1. Important Concepts about Sub Trees

The TNT-HUI algorithm is a recursive algorithm that iterates over sub-tree of the global TN-Tree initially constructed. To clarify the description of the algorithm, we firstly give the following definitions.

Definition 7 (base-itemset and Conditional Tree). A conditional tree (also called a sub-tree) [6] of itemset X is a tree that is constructed using all transaction itemsets containing itemset X (X is removed from these transactions itemsets before they are added to the conditional tree). Itemset X is called the base-itemset of this conditional tree.

A tree that is constructed by all transaction itemsets of a dataset and, in which its base-itemset is null, is called a global tree. A global tree also is called a conditional tree in which its base-itemset is null. $u(X, t)$ in a transaction itemset t containing X is also called base-utility (abbreviated as bu) of transaction itemset t in the conditional tree T .

Definition 8 (Sub Dataset). In a conditional tree T in which its base-itemset is X (if X is null, T is a global tree), suppose item Q appears in k tail nodes, and the corresponding path-itemsets are $Y_1, Y_2, \dots, Y_k$, the itemsets $Y_1 \cup X, Y_2 \cup X, \dots, Y_k \cup X$ (along with their utility values) constitute the sub dataset of itemset $\{Q\} \cup X$. Each record in sub dataset is called sub transaction-itemset.

Definition 9 (Local Candidacy). If the twu value of an item in a sub dataset is less than the minimum utility value, it is called a local unpromising item; otherwise, it is called a local promising item.

According to Theorem 1, the algorithm TNT-HUI removes all unpromising items from original transaction itemsets when it creates the TN-Tree with transaction itemsets. In addition, according to Theorem 1, the algorithm TNT-HUI removes all local unpromising items of a sub dataset when it creates a sub TN-Tree.

5.2. Algorithm Description

The algorithm of mining HUIs from a TN-Tree is shown in Algorithm 2.

We process each item (denoted as Q) in the header table H , starting from the last item, by the following steps.

First, we find the total BU , SU , and NU values for each node of item Q . Since nodes are added to the tree according to the order of the header table, these nodes were all *tail nodes* and each of them contains *tail information*. We count the sum of those nodes' base-utility (Line 4) and the sum of those nodes' path-utility, (Line 5). For each tail-node N_i , we find item Q 's utility from list $N_i.piu$ and denote it as $N_i.nu$. Then, we sum them up as NU (Line 6).

Then, if $(BU + SU)$ is less than the predefined minimum utility value, go to Step 3; otherwise, we add item Q to a base-itemset (which is initialized as an empty set) and generate HUI and create sub TN-Tree to perform mining recursively (Lines 13–14). More specifically, if $(BU + NU)$ is not less than the predefined minimum utility value, then the current base-itemset is an HUI (Lines 10–12). We remove the item Q from the current base-itemset after we perform a recursive mining process on the new sub TN-Tree (line 16).

Finally, for each of these k tail-nodes (which we denote as $N_i, i = 1, 2, \dots, k$), we modify its tail-information by deleting item Q 's utility from list $N_i.piu$ and modifying $N_i.su$ as: $N_i.su = N_i.su - N_i.nu$. If its parent node contains a tail-information, then accumulate this tail-information to its parent's tail-information (lines 27–30); otherwise, move this tail-information to its parent (lines 22–25).

The constructing process of sub tree of the Algorithm 2, summarized in Algorithm 3, is as follows. First, we create a new header table subH by scanning the corresponding path-itemsets in the current TN-Tree (lines 1–8), including deleting unpromising items from subH and sorting its items in the descending order of sn/twu (lines 7–8). Second, we process each path-itemsets in the current TN-Tree, including deleting unpromising items (line 14), sorting items according to subH (line 15), and inserting the path-itemsets to a new TN-Tree subT (lines 16–20).

Algorithm 2: TNT-HUI

Input : T : the TNT-tree constructed from the transactions data;
 H : the header table constructed from the transactions data;
base-itemset:

Output : HUIs

```

1 for each item  $Q$  in  $H$  (with a bottom-up sequence) do
  // Step 1: Calculate BU, SU, and NU
2   $BU \leftarrow 0, SU \leftarrow 0, NU \leftarrow 0$ ;
3  for each node  $N$  for the item  $Q$  in  $T$  do
4     $BU \leftarrow BU + N.bu$ ; // base-itemset utility
5     $SU \leftarrow SU + N.su$ ; // path-itemset utility
6     $NU \leftarrow NU + N.nu$ ; //  $N.nu$  is the utility of item  $Q$  in the
      list  $N.piu$ 
7  end
  // Step 2: Generate HUIs and create sub TN-tree
8  if  $BU + SU \geq min\_util$  then
9    base-itemset  $\leftarrow$  base-itemset  $\cup \{Q\}$ ;
10   if  $BU + NU \geq min\_util$  then
11     Copy base-itemset to HUIs;
12   end
13   Create a sub TN-tree  $subT$  and a header table  $subH$  for base-itemset
      (see Algorithm 3);
14   TNT-HUI( $subT, subH, base-itemset$ ); // recursive call
15   Remove item  $Q$  from itemset base-itemset;
16 end
  // Step 3: Remove item  $Q$  from tree  $T$  by moving each
      tail-node's tail-information to its parent
17 foreach node  $N$  for item  $Q$  in  $T$  do
18    $u \leftarrow$  the utility of item  $Q$  in list  $N.piu$ ;
19   remove utility value of item  $Q$  from list  $N.piu$ ;
20    $N.su \leftarrow N.su - u$ ;
21   if  $N.parent.su == NULL$  then
22      $N.parent.bu \leftarrow N.bu$ ;
23      $N.parent.su \leftarrow N.su$ ;
24      $N.parent.piu \leftarrow N.piu$ ;
25      $N.parent.sn \leftarrow N.sn$ ;
26   else
27      $N.parent.bu \leftarrow N.parent.bu + N.bu$ ;
28      $N.parent.su \leftarrow N.parent.su + N.su$ ;
29      $N.parent.piu \leftarrow N.parent.piu + N.piu$ ;
30      $N.parent.sn \leftarrow N.parent.sn + N.sn$ ;
31   end
32   Remove node  $N$  from  $T$ ;
33 end
34 end
35 return HUIs;

```

Example 3 (HUI Mining based on TN-Tree). For example, in Figure 1, item “B” is the last item in the header table. We firstly calculate its $BU = 0$, $SU = 24 + 35 + 11 + 13 = 83$, $NU = 12 + 6 + 6 + 9 = 33$. Because $BU + SU = 83 > 70$, we add item “B” to a base-itemset (initialized as null), resulting base-itemset B. Because $BU + NU = 33 < 70$, this itemset B is not a high utility itemset. However, as $BU + SU = 83 > 70$, we still construct a sub-header table and a sub TN-Tree for the current base-itemset {B}.

A sub-header table is created by the following. From the path “root-C-D-E-B: 1,0;3,6,6,12;24” in Figure 1, get an itemset {C,D,E,B}, the support number of each item (i.e., 1) and utility of items C, D, E, B and itemset {C,D,E,B} (i.e., 3, 6, 3, 12 and 24), respectively. See the first sub transaction-itemset of the sub dataset in Figure 3a. Similarly, we can get other three sub transaction-itemset from the other three paths, respectively: root-C-D-A-E-B: 1,0;2,4,20,3,6;35, root-C-E-B: 1,0;2,3,6;11, and root-C-B: 1,0;4,9;13. See the sub dataset in Figure 3a (the digital associated with each item, such as 3 in (C,3), is the utility value of this item).

Figure 3. After removing node B.

A sub-header table is created by scanning the sub dataset in Figure 3a, the result is shown in Figure 3d. A sub-header table just maintains all local promising items. A sub TN-Tree is created by the method of TN-Tree in Section 4.2, except that the utility values of itemset {B} of each sub transaction-itemset in Figure 3a are accumulated to the field bu on the tail-node and the item B is not added to the sub TN-Tree. The result is shown in Figure 3d.

Then, we perform a recursive mining process on the new sub-header table and sub TN-Tree. For the last item E in the header table in Figure 3d, $BU = 24$, $SU = 16$, $NU = 9$. Because $BU + SU < 70$, item E is not added to the base-itemset because $BU + NU < 70$, no new sub TN-Tree or HUI is generated. It is the same for item C of the header table in Figure 3d.

After processing all items in Figure 3d, we go on processing remaining items of the header table in Figure 1. Figure 3b is the sub dataset of itemset {A}, and Figure 3e is the corresponding sub TN-Tree of itemset {A}. For the last item D in the header table in Figure 3e, $BU = 50$, $SU = 31$, $NU = 18$. Based on the same judgment as already described above, item D is added to the current base-itemset, resulting base-itemset {A,D}; although {A,D} is still not an HUI, it satisfies the condition of constructing a sub TN-Tree.

From Figure 3e, we can get the sub dataset of itemset {A,D} in Figure 3c, and the new sub-tree is shown in Figure 3f. Now, $BU = 68$, $SU = 13$, $NU = 13$, and because $BU + SU = 81 > 70$, add item C to the base-itemset and get base-itemset {ADC}. Now, $BU + NU = 81 > 70$, which means {A,D,C} is an HUI.

Next, we go on processing the remaining items of the sub-header table in Figure 3e. Then, we go on processing the remaining items of the header table in Figure 1.

The “add/move” process (Step 3 of Algorithm 2.) is a key operation of this algorithm. When a transaction itemset (or sub transaction-itemset) is added to a TN-Tree, its support number, base-utility and its each item’s utility are stored in its tail-node, not in the node itself. Moreover, since a node can appear in multiple branches, its support number, base-utility, utility, etc., should be the sum of the corresponding values of all its tail-nodes. So, tail-information of one node should be passed to its parent node after this node is processed. For example, after processing node B: 1;0;3,6,3,12;24 in Figure 1, according to Step 3, remove B’s utility (12) from B.piu (3,6,3,12) and subtract it from B.su (24), resulting a new tail-information 1;0;3,6,3;12. Since B’s parent node E does not contain tail-

information, we move this new tail-information to this node E , resulting in $E: 1;0;3,6,3;12$ (see Figure 4). In the same manner, tail-nodes $B: 1;0;2,3,6;11$ and $B: 1;0;4,9;13$ in Figure 1 were processed and moved to their parent nodes, resulting in $E: 1;0;2,3;5$ and $C: 1;0;4;4$. Tail-node $B: 1;0;2,4,20,3,6;35$ was added to its parent node (because its parent node contains tail-information), resulting in $E: 2;0;8,8,30,6;52$; see Figure 4.

Figure 4. The tail-node tree-based high-utility itemset (TNT-HUI) algorithm for mining high utility itemsets.

Algorithm 3: Create sub-header table and sub TN-Tree

Input : the previous TN-Tree T , item Q
Output : subT and subH
 // Construct subH
 1 **foreach** node N of item Q in T **do**
 2 **foreach** item I in path-itemset of N in T (excluding Q) **do**
 3 $H \mapsto I.sn \leftarrow H \mapsto I.sn + N.sn$;
 4 $H \mapsto I.twu \leftarrow H \mapsto I.twu + N.su + N.bu$;
 5 **end**
 6 **end**
 7 Delete unpromising items from subH;
 8 Sort subH in the descending order of sn or twu of subH;
 // Construct subT
 9 Initialize subT with an empty root node;
 10 **foreach** node N for item Q in T **do**
 11 $X \leftarrow$ path-itemset of N (excluding Q);
 12 $u \leftarrow$ the utility of item Q in $N.piu$;
 13 $Tin \leftarrow N \mapsto$ tail-information;
 14 Delete unpromising items that is not in subH from X and modify Tin ;
 15 Sort items of X according to subH and modify Tin ;
 16 Insert X to subT;
 // Process the tail information of the tail node sN of itemset X
 17 $sN.sn \leftarrow sN.sn + Tin.sn$;
 18 $sN.piu \leftarrow sN.piu + Tin.piu$;
 19 $sN.su \leftarrow sN.su + Tin.su$;
 20 $sN.bu \leftarrow sN.bu + Tin.bu + u$;
 21 **end**

5.3. Analysis of the Algorithm

Property 1 (TN-Tree Completeness). *Given a transaction dataset DB and a minimum utility value min_uti , its corresponding TN-Tree contains the complete information of DB in relevance to HUIs mining.*

Proof. Based on a TN-Tree construction process, all transactions itemsets that contain the same (local) promising items are mapped to one path (for example, T_4 and T_7 in Table 1 are mapped to one path in Figure 1) and have shared the same tail-node. The sum of utility of each item in those transactions are stored to the field piu on the tail-node. Thus, the utility of an itemset X in DB can also be retrieved from the corresponding tail-nodes. $\square$

Property 2. *Let DB be a dataset, $subDB$ be a sub dataset of itemset X , and itemset Y be in $subDB$ and $X \cap Y = \emptyset$. Then, the utility of $X \cup Y$ in DB is equivalent to the utility of $X \cup Y$ in $subDB$, and itemset $X \cup Y$ is an HUI if and only if it is an HUI in $subDB$.*

Proof. Based on the sub dataset construction process in Example 3 and Definition 8, all transactions containing itemset $X \cup Y$ are mapped to $subDB$. Thus, the utility of itemset $X \cup Y$ in DB is equivalent to the utility of $X \cup Y$ in $subDB$. So, itemset $X \cup Y$ is an HUI in DB if and only if its utility in $subDB$ is not less than the minimum utility value in $subDB$. $\square$

Property 3 (TNT-HUI Correctness). *Given a base-itemset X , in which base utility is BU , for any promising item Q in the $subDB$, (1) if $BU + SU < min_util$, then any superset of itemset $X \cup \{Q\}$ is not an HUI; (2) if $BU + NU \geq min_util$, then itemset $X \cup \{Q\}$ is an HUI; otherwise, it is not an HUI.*

Proof. (1) Firstly, based on the (sub) header table construction process, the twu value in a (sub) header table includes the utility values of (local) unpromising items in the corresponding transactions. Secondly, after an item in a (sub) header table is processed, the algorithm TNT-HUI have mined all HUIs containing this item. So, this algorithm needs not consider those processed items when it processes the remaining items in a (sub) header table. Based on these two reasons mentioned above, we need re-calculated the twu value of an item in a (sub) header table. In the algorithm TNT-HUI, the value $\sum_{i=1}^k (N_i.bu + N_i.su)$ is the new twu of itemset $X \cup \{Q\}$, and it does not include the utility values of the two kinds of items mentioned above (unpromising items and processed items). According to Theorem 1, any superset of itemset $X \cup \{Q\}$ is not an HUI if $\sum_{i=1}^k (N_i.bu + N_i.su)$ is less than the minimum utility value.

(2) Let $subDB$ be the sub dataset of itemset X (if X is null, $subDB$ is the original dataset). Based on the sub TN-Tree construction process, the value $\sum_{i=1}^k (N_i.bu + N_i.nu)$ is the utility of itemset $X \cup \{Q\}$ in $subDB$. According to the Property 2, itemset $X \cup \{Q\}$ is a high utility itemset if and only if $\sum_{i=1}^k (N_i.bu + N_i.nu)$ is not less than the minimum utility value. $\square$

Property 3 guarantees all itemsets mined by the algorithm TNT-HUI are HUIs. For example, in Example 3, the utility value of each new base-itemset ($BU + NU$) is obtained from the tree, so it is an HUI if its utility value is not less than the minimum utility value. Note that, in the special case where X is null, a sub TN-Tree is a global TN-Tree.

Property 4 (TNT-HUI Completeness). *The result found by TNT-HUI is complete. In other words, all HUIs can be discovered by TNT-HUI.*

Proof. Assume $X = \{x_1, x_2, x_3, \dots, x_k\}$ to be an HUI, and any sub-itemset of X is promising itemsets. The algorithm TNT-HUI creates sub-header table and sub-tree for $\{x_1\}$, $\{x_1, x_2\}$, $\dots$, $\{x_1, x_2, \dots, x_{k-1}\}$. Finally, it can get $u(X)$ from sub-tree of the itemset $\{x_1, x_2, \dots, x_{k-1}\}$. So, TNT-HUI can mine all HUIs. $\square$

Property 5. *The algorithm TNT-HUI may not construct a new sub-tree and generate a HUI when it processes an item of a header table.*

Proof. When the algorithm TNT-HUI is processing an item of a header table, the utility of the item (NU), the utility of the current base-itemset (BU), and the sum of its corresponding path-utility (SU) are obtained from its tail-information (shown in Step 1 and 2 in Figure 2). Because this sum of BU and SU does not include the utility of (local) unpromising items and processed items, it will be less than its twu value in the header table. If the sum of BU and SU is less than the minimum utility value, TNT-HUI will not construct a new sub-tree. If the sum of NU and BU is less than the minimum utility value, a HUI is not generated. $\square$

For example, when processing the item E in the header table in Figure 3d, $BU = 24$, $SU = 16$, $NU = 9$, because $(BU + SU)$ and $(BU + NU)$ are less than the predefined minimum utility value 70, the algorithm TNT-HUI does not construct a new sub-tree for the itemset $\{B, E\}$ and generate an HUI $\{B, E\}$.

5.4. Comparison with Existing HUI Mining Algorithms

Tree structures have been used to represent transaction databases for pattern mining. For example, for the dataset in Table 1 and the profit table in Table 2, a global IHUP-Tree is shown in Figure 5, in which items are arranged in the descending order of twu values. In the second step, IHUP generates candidates for HUIs from the IHUP-Tree by employing the FP-Growth method [6]. In the third step, IHUP scans the dataset to find all HUIs from the candidates. During the construction of an UP-Tree, the unpromising items and their utilities are eliminated from the transaction utilities, and the utilities of its descendants of any node are discarded from the utility of the node. For any itemset X , the value of $TWU(X)$ in the UP-Tree is not bigger than that in the IHUP-Tree, so the number of candidates created by the algorithm UP-Growth is not bigger than that created by the algorithm IHUP.

The structures of the header table in algorithms IHUP and UP-Growth contains item, twu value and link information, as shown in Figure 5. The structures of IHUP-Tree and UP-Tree are identical: each node on them contains item, support number, twu (or a value derived from twu value), link to parent, link to children, and link to the next node.

When a transaction itemset is inserted to an UP-tree, each node does not contain utility values of its children nodes. So, UP-Growth's over-estimated utility value (used for judging whether an itemset is a candidate) is lower than that of IHUP. So, this effectively reduces the number of candidate and improves the time efficiency of the judging of candidates. After mapping transaction itemsets to a TN-tree, the itemsets' exact utility values can be retrieved from the tree, so TNT-HUI mines HUIs without generating candidates.

Property 6. *In the sub-trees and sub-header tables of the same base-itemset, the number of items (promising items) in the sub-header table and the number of nodes on the sub-tree created by the algorithm TNT-HUI are not more than that created by the algorithm UP-Growth [16] or the algorithm IHUP [11].*

Proof. When a transaction itemset (or sub transaction-itemset) is added to a tree by the algorithms UP-Growth or IHUP, the utility value of each item is stored in the corresponding node (shown in Figure 5c,d), so the actual utility of each item of each path on the UP-Tree or IHUP-Tree cannot be obtained when a sub-tree is constructed. An over-estimated utility value of each item is used to construct the sub-tree and the new sub-header table. On the other hand, TNT-HUI creates a sub-tree and a sub-header table with the actual utility of each item of each path on a TN-Tree. Since the over-estimated twu is not less than the actual value, the number of (local) promising items in the sub-header table, and the number of nodes on a sub-tree, will be not larger than that in the algorithms UP-Growth or IHUP. $\square$

Figure 5. Example tree structures based on toy data in Tables 1 and 2.

6. Experimental Results

We evaluated the performance of the proposed algorithm on real-life transaction datasets (mushroom, chess, connect, retail, and chain-store) and two synthetic transaction datasets T10.I4.D100K and T10.I6.D100K. Table 3 shows the characteristics of these transaction datasets, where second column (I) shows the number of distinct items, third column (AS) shows the average size of transactions, fourth column (T) shows the total number of transactions, and the last column (DS) shows the percentage of total distinct items that appear in each transaction. The last column (DS) in Table 3 provides a measure of whether a dataset is dense or sparse. In general, a sparse dataset contains fewer items per transaction, but the set of items is relatively large. A dense dataset, in contrast, has many items per transaction, but the set of items is relatively small. Therefore, when the value of DS parameter of a dataset is relatively low (e.g., less than or equal to 10.0), a dataset is said to be sparse [27]. For example, both the datasets chess and mushroom are dense datasets, and the other four are sparse datasets.

Table 3. Dataset characteristics.

Dataset	I	AS	T	DS
chess	76	37	3196	48.68%
mushroom	119	23	8124	19.33%
connect	129	43	67,557	33.33%
T10.I4.D100K	1000	10	100,000	1%
T10.I6.D100K	1000	10	100,000	1%
retail	16,470	10.3	88,162	0.06%
Chain-store	46,086	7.2	1,112,949	0.0156%

The dataset chain-store is a real-life dataset, which was adopted from a major grocery store chain in California (NU-MineBench Version 2.0 Score Code and Datasets, available at <http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html>, accessed July 2017),

and provides the real profit for each item. For other six datasets that do not have utility information, we adopted the method of artificially generating utility values as applied in previous literature [11,16,19,20,23]: the quantity for each item of each transaction itemset was randomly assigned, ranging from 1 to 10; and the external utility tables were generated by assigning a utility value to each item randomly, ranging from 0.0100 up to 10.0000. Since most items are in the low profit range in real world dataset, we also adopted the method of generating the utility values by a log-normal distribution like the existing works [11,16,19,20,23]. The original transaction datasets chess, mushroom, connect, retail, and T10.I4.D100K were obtained from FIMI Repository (<http://fimi.cs.helsinki.fi/data/>), accessed in October 2010. The dataset T10.I6.D100K was generated by the IBM data generator (<http://www.almaden.ibm.com/software/quest/Resources/index.shtml>). The source code and testing datasets (the first five datasets) used in our experiments can be downloaded from the following address: <http://code.google.com/p/tnt-hui/downloads/list>.

We compare the performance of the algorithm TNT-HUI with the algorithm UP-Growth [16]. Based on the two algorithms, we design four compared algorithms and give them new notations as follows. As stated in Section 4.2, we can structure a TN-Tree in descending order of their support numbers or their *twu* values. To compare the performance of these two types of trees, we annotate them as follows. A TN-Tree in which nodes are arranged in descending order of their support numbers is denoted as TNTsn, and a TN-Tree in which nodes are arranged in descending order of their *twu* values is denoted as TNTtwu. Our algorithm based on TNTsn is denoted as TNT-HUIsn and the algorithm based on TNTtwu is denoted as TNT-HUITwu. The algorithm UP-Growth [16] is called UP-UPG when using UP-Growth and is called UP-FPG when using FP-Growth [6]. All algorithms were written in Java programming language.

The configuration of the testing platform is as follows: Windows 7 operating system, 8G Memory, Intel(R) Core(TM) i7-3612 CPU @ 2.10 GHz.

6.1. Evaluation of Computational Efficiency

We first compare runtime of four algorithms. Figure 6 shows the comparison of running time on each dataset under various minimum utility thresholds. The smaller the minimum utility threshold (η) is, the longer the algorithm will be running. On dense dataset chess and connect, when minimum utility threshold is too small, UP-UPG and UP-FPG would cause memory overflow or run too much time, so the corresponding data points are omitted in Figure 6.

It can be seen from Figure 6 that algorithm TNT-HUI is superior than UP-UPG and UP-FPG, and it can achieve the level of about 4 orders of magnitude on dense datasets and about 2 orders of magnitude on sparse datasets. The dense dataset chess contains only 107 HUIs when threshold is 27%, and no HUI when threshold is 29%, but UP-UPG takes 18,976.511 seconds even when no HUI is mined, while TNT-HUI takes only 1.264 seconds. Similar result occurs on dataset connect: UP-UPG takes 217,026.636 seconds and TNT-HUI takes 2.4 s. when threshold is 40%, and no HUI exists in this case. Figure 6 also shows that TNT-HUI not only outperforms significantly in terms of time efficiency but also develops more smoothly when the threshold decreases.

The reasons why TNT-HUI performs so well in terms of runtime are as follows.

(1) TNT-HUI maps transaction itemsets to TN-tree, and the itemsets' exact utility values can be retrieved from the tree. So, it can find all HUIs from the tree using pattern-growth approach, while UP-FPG and UP-UPG can only find candidate itemsets using over-estimated values and need extra scans on original dataset to calculate the actual utility value. The more candidates generated, the more time-consuming UP-FPG and UP-UPG will be.

Figure 6. Comparison of running time.

(2) The number of trees constructed by the algorithm TNT-HUI is not more than that created by the algorithm UP-Growth or IHUP. Firstly, the number of items in the first header table constructed by 4 algorithms is the same and the twu value of the corresponding item is also the same. Secondly, since the over-estimated twu is not less than the actual value, the number of (local) promising items in the sub header table, and the number of nodes on a sub-tree, will be not larger than that in the algorithms UP-Growth (see Property 6). Lastly, in the algorithms UP-Growth and IHUP, each item of a header table (or sub header table) must be added to the base-itemset to generate a new base-itemset, and a sub header table must be constructed for the new base-itemset. In this case, a new sub-tree must be constructed if the sub header table is not empty. When the algorithm TNT-HUI is processing an item of a header table, it may not construct a new sub-tree, a new sub header table, and a high utility itemset (see Property 5). Summarizing the above cases, we can conclude that the number of trees created by the algorithm TNT-HUI is not more than that created by UP-Growth or IHUP. The fewer trees are created, the less the executive time and space the algorithm costs. As shown in Table 4, although the number of HUIs is 0, the number of candidates is large in algorithm UP-Growth, even approaching 3,640,999, and the number of its trees is 1,745,321. Table 4–10 show that algorithm UP-UPG and UP-FPG generate a large number of candidate itemsets and more trees in the mining process.

Table 4. Details analysis on the dataset chess.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
33	0	93	96	394,029	1,476,703	834,105	2,953,671
31	0	178	185	855,972	2,702,584	1,796,242	5,405,523
29	0	361	387	1,745,321	4,891,370	3,640,999	9,783,237
27	107	1,196	1,241	3,460,518		7,191,033	
25	1745	5066	5205	6,756,642		13,986,739	
23	9805	20,563	20,900				
21	47,926	79,866	80,791				

Table 5. Details analysis on the dataset mushroom.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
19	0	36	107	108	1000	5899	2198
17	0	77	209	212	5109	10,878	11,760
15	0	241	609	622	17,013	24,977	36,948
13	0	1526	3863	3880	26,539	29,064	53,575
11	0	6719	9831	9900	41,767	50,475	85,583
9	1745	18,254	19,642	19,679	101,782	201,684	216,024
7	99,269	51,076	60,321	59,960	318,297	389,648	643,620

Table 6. Details analysis on the dataset connect.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
49	0	9	9	9	159,473	185	318,974
46	0	14	14	5686	824,367	15,523	1,648,758
43	0	22	23	190,159	3,082,099	425,432	6,164,221
40	0	41	44	1,323,594	9,375,217	2,841,948	18,750,456
37	0	83	88				
34	0	239	245				
31	209	1620	1630				
28	34,613	80,893	80,958				

Table 7. Details analysis on the dataset T10I4D100k.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
0.32	106	336	366	1499	1798	3722	4208
0.28	178	548	609	2054	2661	4965	5972
0.24	479	1058	1159	3163	3895	7335	8511
0.2	1027	1474	1581	4588	5505	10,455	11,974
0.16	2014	2283	2433	6666	7583	15,047	16,517
0.12	3993	3545	3741	9216	10,142	20,827	22,302
0.08	8658	6550	6889	13,415	14,367	30,664	32,149

Table 8. Details analysis on the dataset T10I6D100k.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
0.12	277	552	607	1986	3347	5931	8239
0.11	344	680	731	3454	4105	9370	10,526
0.1	524	1011	1103	4399	7974	12,867	19,454
0.09	906	1540	1665	9058	20,224	24,785	45,538
0.08	1690	2412	2503	23,823	33,709	56,803	74,783
0.07	3960	4892	5075	41,833	58,777	97,560	128,010
0.06	10,504	11,754	12,250	72,789	85,935	163,060	186,203

Table 9. Details analysis on the dataset retail.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
0.14	243	898	981	1167	1315	4846	5015
0.12	318	1315	1371	1709	1975	6201	6493
0.1	456	1803	1891	2547	2930	8134	8549
0.08	652	2622	2710	3889	4413	11,263	11,873
0.06	1101	3732	3791	6073	7136	16,708	18,099
0.04	2203	5431	5486	12,075	17,569	31,599	41,003
0.02	6902	9809	9926	195,794	483,875	416,416	982,370

Table 10. Details analysis on the dataset Chain-store.

η (%)	# HUIs	# Trees Created in the Process of Mining				# Candidates	
		TNT-HUIsn	TNT-HUITwu	UPG	FPG	UPG	FPG
0.021	1072	2321	2344	2408	2656	20,941	21,291
0.019	1263	2760	2838	2925	3280	24,030	24,535
0.017	1544	3374	3437	3561	4097	28,157	28,950
0.015	1914	4111	4201	4361	5212	34,152	35,406
0.013	2440	4995	5137	5396	6772	42,913	45,035
0.011	3266	6186	6337	6753	9097	56,579	60,296
0.009	4582	7663	7820	8636	12,742	79,330	86,345
0.007	6925	9894	10,035	11,655	20,725	122,446	139,303

In summary, algorithm TNT-HUI can directly obtain high utility itemsets from the tree using pattern-growth approach, and it creates less trees than UP-UPG and UP-FPG, so its performance has been greatly improved in term of runtime.

6.2. Evaluation of Memory Usage

We measured the actual memory consumption of the algorithms and compared the maximum memory usage, as summarized in Figure 7.

Figure 7 is the memory consumption of four algorithms on each dataset and under various thresholds. On dense datasets chess and connect, UP-UPG and UP-FPG could run out of memory or ran too long finish when the minimum threshold is smaller than a certain level, so there are some missing values on Figure 7 for these two datasets. It can also be seen from Figure 7 that the spatial efficiency of TNT-HUI is significantly improved on dense datasets, while the memory consumption of UP-UPG and UP-FPG increase greatly while the thresholds decrease. The main reason is that UP-UPG and UP-FPG generate a large number of candidates and much more trees in the mining process (as shown in Tables 5–10). In sparse datasets, because the tree structure of TN-Tree is more complex (with more utility values) than UP-Tree, the memory consumption is more than UP-UPG and UP-FPG when threshold is big. But with the decreasing of the threshold, the number

of trees generated by UP-UPG and UP-FPG is significantly more than that of TN-Tree, and the number of candidate itemsets increases dramatically, the overall memory consumption of TNT-HUI would be less than that of other two algorithms in these cases.

By the above experiments, we can conclude that the algorithm TNT-HUI is effective in time and space for different types of datasets, especially for the small minimum utility threshold and dense datasets. Moreover, compared with the algorithm TNT-HUITwu, the number of trees constructed by TNT-HUIsn is smaller, so TNT-HUIsn outperforms both UP-Growth and TNT-HUITwu.

Figure 7. Comparison of maximum memory usage.

7. Conclusions

In this paper, we proposed an efficient algorithm, called TNT-HUI, for mining HUIs from transaction dataset. Based on pattern-growth approach, it can mine HUIs directly without generating candidate itemsets through only two scans of the dataset. A novel data structure, called TN-Tree, was proposed for maintaining the transactional dataset. TN-Tree maintains the utility of each individual item of the itemset in its tail-node, so TNT-HUI can retrieve the utility value of an itemset and the algorithm can find HUIs from TN-Tree without generating candidate itemset. In the experiments, dense datasets, sparse datasets, synthetic dataset, real-life dataset, and a dataset containing many long transaction itemsets are used to evaluate the performance of our algorithm. The mining performance is enhanced significantly since TNT-HUI does not generate candidate itemsets and creates fewer trees. The results are nearly the same on 7 different datasets: the algorithm TNT-HUI not only outperforms other algorithms in terms of execution time by 4 orders of magnitude on dense datasets and 2 orders of magnitude on sparse datasets, but also in terms of space,

and the runtime of TNT-HUI increases smoothly, along with the decrease of the minimum utility threshold.

This work may be extended or improved in various ways [28–30]. A TN-Tree can firstly arrange items in descending order of support number, and those items are arranged in their *twu* values descending order when support numbers of the header table are same. The algorithm TNT-HUI can be improved by this strategy. The algorithm TNT-HUI can also adopt the idea of tree reconstruction [29,30]. In this case, we may first construct a tree and a header table by scanning dataset once, and then reconstruct the tree by the order of header table. So, HUIs are mined with only once scan of a dataset. TNT-HUI can be adopted for parallel computing, as well. After the header table is constructed, the items in the header table can be processed in parallel. For example, as shown in Figure 2a, there are five items in the header table, and we can build five sub-trees in the order of the header table for these five items, respectively, and these trees can be created and processed in parallel. If the sub-tree is also a big one, we can deal with it in the same way.

Author Contributions: Conceptualization, B.J. and L.W.; methodology, Y.L.; validation, L.F., L.W. and B.J.; writing—original draft preparation, Y.L.; writing—review and editing, L.W., L.F. and B.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by the Science Challenge Project (No.TZ2016006-0107-02), Ningbo Natural Science Foundation Project (No.2017A610122), Ningbo Soft Science Research Project (No.2014A10008).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Djenouri, Y.; Lin, J.C.W.; Nørnvåg, K.; Ramampiaro, H. Highly efficient pattern mining based on transaction decomposition. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, 8–11 April 2019; pp. 1646–1649.
2. Nguyen, L.T.; Nguyen, P.; Nguyen, T.D.; Vo, B.; Fournier-Viger, P.; Tseng, V.S. Mining high-utility itemsets in dynamic profit databases. *Knowl.-Based Syst.* **2019**, *175*, 130–144. [\[CrossRef\]](#)
3. Agrawal, R.; Srikant, R. Fast algorithms for mining association rules in large databases. In Proceedings of the International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile, 12–15 September 1994; pp. 487.
4. El-hajj, M.; Zaiane, O. COFI-tree mining: A new approach to pattern growth with reduced candidacy generation. In Proceedings of the IEEE International Conference on Frequent Itemset Mining Implementations (FIMI), Melbourne, FL, USA, 19 December 2003.
5. Grahne, G.; Zhu, J. Fast algorithms for frequent itemset mining using FP-trees. *IEEE Trans Knowl. Data Eng.* **2005**, *10*, 1347–1362. [\[CrossRef\]](#)
6. Han, J.; Pei, J.; Yin, Y.; Mao, R. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Min. Knowl. Discov.* **2004**, *8*, 53–87. [\[CrossRef\]](#)
7. Song, M.; Rajasekaran, S. A Transaction Mapping Algorithm for Frequent Itemsets Mining. *IEEE Trans Knowl. Data Eng.* **2006**, *4*, 472–481. [\[CrossRef\]](#)
8. Wang, E.T.; Chen, A.L. Mining frequent itemsets over distributed data streams by continuously maintaining a global synopsis. *Data Min. Knowl. Discov.* **2011**, *23*, 252–299. [\[CrossRef\]](#)
9. Djenouri, Y.; Comuzzi, M. Combining Apriori heuristic and bio-inspired algorithms for solving the frequent itemsets mining problem. *Inf. Sci.* **2017**, *420*, 1–15. [\[CrossRef\]](#)
10. Lin, J.C.W.; Zhang, Y.; Zhang, B.; Fournier-Viger, P.; Djenouri, Y. Hiding sensitive itemsets with multiple objective optimization. *Soft Comput.* **2019**, *23*, 12779–12797. [\[CrossRef\]](#)
11. Ahmed, C.; Tanbeer, S.; Jeong, B.; Lee, Y. Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases. *IEEE Trans Knowl. Data Eng.* **2009**, *21*, 1708–1721. [\[CrossRef\]](#)
12. Guo, G.; Zhang, L.; Liu, Q.; Chen, E.; Zhu, F.; Guan, C. High utility episode mining made practical and fast. In Proceedings of the International Conference on Advanced Data Mining and Applications, Guilin, China, 19–21 December 2014; pp. 71–84.
13. Hu, J.; Silovic, A. High-utility Pattern Mining: A Method for Discovery of High-Utility Item Sets. *Pattern Recognit.* **2007**, *40*, 3317–3324. [\[CrossRef\]](#)
14. Liu, J.; Wang, K.; Fung, B. Direct Discovery of High Utility Itemsets without Candidate Generation. In Proceedings of the 2012 IEEE 12th International Conference on Data Mining (ICDM), Brussels, Belgium, 10–13 December 2012; pp. 984–989.
15. Liu, M.; Qu, J. Mining high utility itemsets without candidate generation. In Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM), Maui, HI, USA, 29 October–2 November 2012; pp. 55–64.

16. Tseng, V.; Shie, B.; Wu, C.; Yu, P. Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases. *IEEE Trans Knowl. Data Eng.* **2013**, *25*, 1772–1786. [\[CrossRef\]](#)
17. Tseng, V.; Wu, C.; Shie, B.; Yu, P. UP-Growth: An Efficient Algorithm for High Utility Itemset Mining. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 24–28 July 2010; pp. 253–262.
18. Wu, C.; Shie, B.; Tseng, V.; Yu, P. Mining top-K High Utility Itemsets. In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Beijing, China, 12–16 August 2012; pp. 78–86.
19. Li, Y.; Yeh, J.; Chang, C. Isolated Items Discarding Strategy for Discovering High Utility Itemsets. *Data Knowl. Eng.* **2008**, *64*, 198–217. [\[CrossRef\]](#)
20. Liu, Y.; Liao, W.; Choudhary, A. A Two-Phase Algorithm for Fast Discovery of High Utility Itemsets. In Proceedings of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD), Hanoi, Vietnam, 18–20 May 2005; pp. 689–695.
21. Yao, H.; Hamilton, H. Mining Itemset Utilities from Transaction Databases. *Data Knowl. Eng.* **2006**, *59*, 603–626. [\[CrossRef\]](#)
22. Yao, H.; Hamilton, H.; Butz, G. A Foundational Approach to Mining Itemset Utilities from Databases. In Proceedings of the 4th SIAM International Conference on Data Mining (SDM), Orlando, FL, USA, 29 April–1 May 2004; pp. 482–486.
23. Erwin, A.; Gopalan, R.; Achuthan, N. CTU-mine: An efficient high utility itemset mining algorithm using the pattern growth approach. In Proceedings of the 7th IEEE International Conference on Computer and Information Technology, Fukushima, Japan, 16–19 October 2007; pp. 71–76.
24. Lin, C.; Hong, T.; Lan, G.; Wong, J.; Lin, W. Mining High Utility Itemsets Based on the Pre-large Concept. *Adv. Intell. Syst. Appl.* **2013**, *1*, 243–250.
25. Lin, C.; Hong, T.; Lu, W. An Effective Tree Structure for Mining High Utility Itemsets. *Expert Syst. Appl.* **2011**, *38*, 7419–7424. [\[CrossRef\]](#)
26. Tseng, V.S.; Wu, C.W.; Fournier-Viger, P.; Yu, P.S. Efficient Algorithms for Mining Top-K High Utility Itemsets. *IEEE Trans Knowl. Data Eng.* **2016**, *28*, 54–67. [\[CrossRef\]](#)
27. Ye, F.; Wang, J.; Shao, B. New Algorithm for Mining Frequent Itemsets in Sparse Database. In Proceedings of the International Conference on Machine Learning and Cybernetics, Guangzhou, China, 18–21 August 2005; pp. 1554–1558.
28. Cheng, J.; Zhu, L.; Ke, Y.; Chu, S. Fast algorithms for maximal clique enumeration with limited memory. In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Beijing, China, 12–16 August 2012; pp. 1240–1248.
29. Koh, J.; Shieh, S. An Efficient Approach for Maintaining Association Rules Based on Adjusting FP-Tree Structures. *Database Syst. Adv. Appl.* **2004**, 2973, 417–424.
30. Tanbeer, S.; Ahmed, C.; Jeong, B.; Lee, Y. CP-Tree: A Tree Structure for Single-Pass Frequent Pattern Mining. In Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD), Osaka, Japan, 20–23 May 2008; pp. 1022–1027.