Appl Intell (2017) 47:114–131
DOI 10.1007/s10489-016-0890-z

Efficient algorithm for mining high average-utility itemsets in incremental transaction databases

Donggyu Kim[1] · **Unil Yun**[1]

Published online: 14 February 2017

Abstract In this paper, we present a novel algorithm for efficiently mining high average-utility itemsets (HAUIs) from incremental databases, in which their volumes can be expanded dynamically. The previous algorithms have inefficiencies in that they must scan a given database multiple times so as to generate candidate itemsets and determine valid itemsets level by level. The reason is that they follow the basic framework of an *Apriori-like* approach. This drawback can cause critical problems in processing incremental databases because scanning a database becomes a tougher task as the size of the database is increased. In contrast, the algorithm proposed in this paper builds a compact tree structure maintaining all necessary information in order to avoid such excessive database scanning during its mining process. The previous algorithms suffer from the huge generation of unnecessary candidate itemsets at each level accompanied by the naive combination based candidate generation manner of an *Apriori-like* approach, which generates candidate itemsets with $(k + 1)$-lengths by simply joining itemsets with k-lengths. On the other hand, our algorithm employs the pattern growth approach, which allows the algorithm to generate a set of only essential candidate itemsets. In order for our algorithm to constantly preserve the compactness of its tree structure during the entire incremental mining process, a restructuring technique is exploited. In the performance evaluation, we show that our algorithm is faster and consumes less memory space than competitors.

Keywords Association rule mining · Incremental itemset mining · High average-utility itemset mining

1 Introduction

In the beginning of association rule mining [21, 38, 39, 41], frequent itemset mining (FIM) [17, 19] was mainly used in order to find useful itemset information that can be referred to in decision making processes. In FIM, transaction databases are composed of multiple transactions with items that have binary information, which determines whether the corresponding items appear in transactions or not. If the frequency (also called a support) of a given itemset satisfies a predefined threshold, the itemset is mined as a frequent itemset. The Apriori algorithm [1] is the most well-known FIM algorithm discovering frequent itemsets based on a level-wise approach, which generates $(k + 1)$-itemsets by joining k-itemsets found in the previous level. Since these join operations are based on a naive combination manner, the itemsets generated from the join operations are potential frequent itemsets (called candidate itemsets), which may have actual supports less than a threshold. Thus, this algorithm requires a database scan at each level in order to calculate actual supports of candidate itemsets. A range of algorithms have been developed based on the methodology of the Apriori algorithm called an *Apriori-like* approach. However, a number of database scans cause significant performance degradation. Therefore, the FP-Growth algorithm [8] using a compact tree structure and a pattern growth approach was proposed. This algorithm has much better performance than Apriori because it generates frequent

✉ Unil Yun
yunei@sejong.ac.kr

Donggyu Kim
donggyukim@sju.ac.kr

[1] Department of Computer Engineering, Sejong University, Seoul, Republic of Korea

itemsets without the generation of candidate itemsets, and scans a given database only two times for constructing its tree structure compactly. Meanwhile, there are other various association rule mining approaches such as sequential itemset mining [7, 33, 40, 42], weighted frequent itemset mining [18, 20, 37], and high utility itemset mining [5, 14, 22, 23, 28, 29, 31, 43]

In a variety of decision making processes, high utility itemset mining (HUIM) can be more advantageous than traditional FIM. In this approach, if the utility of a given itemset is greater than or equal to a given threshold, the itemset becomes a high average utility itemset (HUI). Since HUIM evaluates itemset utilities by considering non-binary item information (called utility information) including the importance and quantity of each item, this approach can extract more useful itemsets from a given database compared to FIM.

However, HUIM accompanies several problems in its mining result. HUIM adds the utilities of all items included in an itemset in order to calculate the utility of the itemset. Since the summation of item utilities in an itemset is more likely to become larger as the length of the itemset is increased, HUIM generally mines a huge number of HUIs, which consist of many items with low utilities. Note that items with low utilities generally have high supports and therefore appear in most transactions. This property of HUIM can cause limitations in the usage of HUIM methods because searching specific itemsets from a huge number of result itemsets is a difficult task even to domain experts.

To supplement such drawbacks of HUIM, a novel approach called high average-utility itemset mining (HAUIM) [9, 13, 36] was suggested. HAUIM employs an average-utility measure, which calculates the utility of an itemset as the summation of item utilities divided by the length of the itemset. In HAUIM, even though the length of an itemset is long, the evaluated utility of the itemset can be low if most items contained in the itemset have low item utilities. Therefore, by using HAUIM methods, it is possible to provide a lesser number of meaningful result itemsets to users.

Since new data are continually generated from various sources and accumulated in databases, techniques for efficiently yielding new mining results reflecting new data have been required for data analysis. The algorithms for handling static databases generally scan a given database two times at the first steps of their mining processes so as to pre-prune invalid items. In incremental itemset mining, however, such pre-pruning strategies based on two database scans cannot be employed because items that are not important in a given database can have high importance after new data are reflected into the database. For example, even though certain items have supports less than a threshold in an original database, their supports can be greater than the threshold after a number of new transactions are added into the database. Therefore, the algorithms for handling static databases have to process all data from scratch in order to mine itemsets from incremental databases whenever new data are generated. Since such mining processes require excessive execution time and huge memory, incremental itemset mining methods [32] for efficiently mining itemsets from dynamic databases have been developed in recent years. Even though the researches on methods for mining high average-utility itemsets (HAUIs) from incremental databases have been conducted in previous studies, the existing algorithms still have inefficiencies in their runtime and memory usage performances because they have their roots in an *Apriori-like* approach.

Motivated by the problems of previous studies, we propose a novel algorithm named IMHAUI (*Incremental mining of high average-utility itemsets*) that can mine HAUIs from incremental databases more efficiently than the existing algorithms by employing a compact tree structure and pattern growth approach. This paper has the contributions as follows. 1) We suggest a new tree structure named IHAUI-Tree (*Incremental high average utility itemset tree*) that maintains the information of incremental databases so that the proposed algorithm can mine HAUIs without a number of database scans. 2) We adopt the path adjusting method that is one of restructuring techniques in order to preserve the compactness of IHAUI-Tree. The path adjusting method is modified to restructure IHAUI-Tree based on an AUUB descending order. 3) We perform a time complexity analysis in order to prove that the proposed algorithm has better performance than competitors theoretically. In addition, based on the results of experiments conducted on four real datasets and two groups of synthetic datasets, we provide empirical proofs supporting that the proposed algorithm is much more efficient than previous algorithms.

The rest of this paper is organized with the following contents. In Section 2, we explain the background knowledge related to the topic of this paper in order to gain understanding. After that, in Section 3, the description of the proposed algorithm is provided together with a multiple number of running examples. In Section 4, we evaluate the performance of the proposed algorithm in terms of runtime, memory usage, and scalability through the investigation of experimental results obtained from various experiments. Finally, we conclude the paper in Section 5, summarizing contents in the paper and suggesting the directions of our future studies.

2 Background

2.1 High utility itemset mining

The Two-phase algorithm [24] firstly proposed the concept of transaction weighted utilization (TWU) to maintain

the anti-monotone property in HUIM. With TWU values, Two-phase can determine whether itemsets can be extended to HUIs or not. Thereafter, tree-based HUIM algorithms such as IHUP [2], UP-Growth [30], MU-Growth [35], and HUPID [34] were introduced in order to achieve more efficient HUIM processes because Two-phase has many drawbacks such as the requirement of multiple database scans. Even though they have better performances compared to Two-phase, they still require huge runtime and memory space in order to validate a large number of generated candidate itemsets. In recent years, several list-based algorithms such as HUI-Miner [25], FHM [6], and HUP-Miner [12] that can mine HUIs without candidate generation have been devised for facilitating more efficient HUI mining processes, which directly generate actual HUIs without excessive candidate itemset validation processes.

2.2 High average-utility itemset mining

Since the concept of high average-utility itemset mining (HAUIM) was proposed to solve the problems of HUIM, a number of HAUIM algorithms have been developed in order to efficiently mine HAUIs from static databases. The first HAUIM algorithm, TPAU (*Two-phase average-utility mining*), [9] mines HAUIs by employing an *Apriori-like* approach similar to the Two-phase algorithm. This algorithm uses the concept of an average-utility upper-bound so as to maintain the anti-monotone property in HAUIM because anti-monotone properties used in other approaches cannot be applied to HAUIM. However, even though this algorithm can reduce its search space through the anti-monotone property, it consumes excessive computational resources for scanning databases in multiple times because it follows the basic framework of an *Apriori-like* approach. Therefore, several HAUIM algorithms have been proposed to enhance the performance of TAPU by using novel approaches such as the mining process based on a projected database [15], improved average-utility upper-bound strategy [16], and an enumeration tree [26]. Since these algorithms store the information of a given database in main memory, they do not require multiple database scans in their mining process. Thus, they have much better performances compared to TPAU. However, these are not suitable for mining HAUIs from incremental databases because they have to perform their mining processes from the scratch whenever new data should be reflected to mining results.

2.3 Mining high average-utility itemsets from incremental databases

The ITPAU (*Incremental two-phase average-utility mining*) [10] algorithm is a state-of-the-art algorithm for mining HAUIs from incremental databases. This method also employs the basic framework of an *Apriori-like* approach similarly to TPAU, but additionally adopts an itemset information maintenance technique [4] so as to increase efficiency in each incremental mining process. The comprehensive HAUI mining procedure of ITPAU can be described as follows. First, the mining process for an original database is conducted in the same manner as that of TPAU. However, the information of promising itemsets with average-utility upper-bounds no less than a given threshold is stored in main memory. The reason is that promising itemsets are more likely to have high average-utilities after new transactions are additionally inserted into the original database. Therefore, the validation of the itemsets that are promising itemsets in the previous mining step can be completed by scanning only the new transactions because their information in the original database can be obtained by directly accessing main memory. Therefore, ITPAU can attenuate the difficultness of constant database scanning and make each incremental mining process more efficient. Based on such advantages of the itemset information maintenance technique, ITPAU obviously has better runtime performance than TPAU in incremental mining environments. However, compared to the traditional TPAU algorithm, this algorithm may require a little more memory space because itemset information should be maintained in the main memory. In addition, even though this algorithm can speed up the runtime of incremental mining processes, it has an explicit limitation because it follows the framework of Apriori.

2.4 Preliminaries

Table 1 presents a simple transaction database with utility information. Table 1a shows the five transactions. The transactions have the corresponding transaction identifiers (called TIDs), which allow us to distinguish them. For

Table 1 Transaction database with utility information

TID	Transaction
100	c(2) d(1), e(1)
200	a(2), b(1) c(3) e(1)
300	a(1), c(3) d(4)
400	a(3), b(1)
500	b(1), c(1)

(a)Transaction database

Item	External utility
a	5
b	7
c	3
d	4
e	6

(b) External utility table

example, the first transaction in Table 1a is denoted as T_{100}. In addition, each transaction consists of multiple items ('a', 'b', 'c', 'd', 'e') with internal utilities (presented by the numbers in brackets), which indicate their own quantities in the corresponding transactions. In this example database, the internal and external utility of an item are positive integers. For example, the internal utility of 'c' in T_{100} is 2. On the other hand, Table 1b shows the external utility of each item, which signifies the importance of the corresponding item. For example, the external utility of 'a' is 5. The following definitions have been used in the previous studies on HAUIM.

Definition 1 (Item utility) The item utility of item i in transaction T can be calculated by multiplying the external utility of i and the internal utility of i in T. When $eu(i)$ and $iu(T, i)$ respectively indicate the external utility of i and the internal utility of i in T, the item utility of i in T is denoted as $u(T, i)$ and can be calculated by the following formula.

$$u(T, i) = iu(T, i) \times eu(i)$$

For example, in Table 1, the item utility of 'c' in T_{100} is $iu(T_{100}, \text{'c'}) \times eu(\text{'c'}) = 2 \times 3 = 6$.

Definition 2 (Average-utility) The average-utility of itemset X in T can be obtained by dividing the summation of item utilities contained in X by the length of X. When the length of X is denoted as $l(X)$, the average-utility of X in T is denoted as $au(T, X)$ and defined as the following equation.

$$au(T, X) = \frac{\sum_{i \in X \subseteq T} u(T, i)}{l(X)}$$

For example, the average-utility of the itemset, {'c', 'd'}, in T_{100} is $\frac{u(T_{100}, \text{'c'}) + u(T_{100}, \text{'d'})}{l(\{\text{'c'}, \text{'d'}\})} = \frac{6+4}{2} = 5$.

The average-utility of X in a given database can be computed by summating average-utilities of X in all transactions having X as their own subset. Therefore, the average-utility of X in D is denoted as $au(X)$ and defined as follows.

$$au(X) = \sum_{X \subseteq T \in D} au(T, X)$$

For example, the average-utility of {'c', 'd'} in Table 1 is $au(T_{100}, \{\text{'c'}, \text{'d'}\}) + au(T_{300}, \{\text{'c'}, \text{'d'}\}) = 5 + 12.5 = 17.5$.

Definition 3 (Minimum utility threshold) A minimum utility threshold is a criterion for determining whether itemsets are valid or not in HAUIM. A percentage value denoted as δ is initially given from a user. Then, a minimum utility threshold can be gained through the product of δ and the total utility denoted as U, which is the summation of all utilities values in a database. Therefore, a minimum utility threshold can be defined as the following equation.

$$minutil = \delta \times U$$

For example, when δ is given as 10 %, *minutil* in Table 1 is 11 because the total utility of the database is $(6 + 4 + 6) + (10 + 7 + 9 + 6) + (5 + 9 + 16) + (15 + 7) + (7 + 3) = 110$.

Definition 4 (High average-utility itemset) If X has an average-utility greater than or equal to a minimum utility threshold, X is a high average-utility itemset.

For example, when a minimum utility threshold is 11, {'c', 'd'} is a high average-utility itemset because its average-utility is greater than 11.

In HAUIM, the anti-monotone property based on supports or TWU values cannot be directly employed. Thus, the alternative way for maintaining the anti-monotone property is necessitated in order to reduce the search space of HAUIM. The following definitions associated with the anti-monotone property holding in HAUIM are used in the previous studies.

Definition 5 (Maximal utility) The maximal utility of T is set as the item utility, which is the highest among all item utilities in T. Therefore, the maximal utility of T is notated as $mu(T)$ and can be defined as the following formula.

$$mu(T) = max(u(T, i_1), u(T, i_2), \ldots, u(T, i_n))$$

For example, the maximal utility of T_{100} in Table 1 is $max(u(T_{100}, \text{'c'}), u(T_{100}, \text{'d'}), u(T_{100}, \text{'e'})) = max(6, 4, 6) = 6$.

Definition 6 (Average-utility upper-bound) Based on the concept of a maximal utility in Definition 5, the average-utility upper-bound (AUUB) of X can be obtained by summating maximal utilities of all transactions having X as their own subset. The following equation is the definition of AUUB.

$$ub(X) = \sum_{X \subseteq T \in D} mu(T)$$

For example, the AUUB value of {'c', 'd'} is $mu(T_{100}) + mu(T_{300}) = 6 + 16 = 22$.

Definition 7 (High average-utility upper-bound itemset) If X has an average-utility upper-bound greater than or equal to a minimum utility threshold, X is a high average-utility upper-bound itemset (HAUUBI) based on Definition 6.

For example, {'c', 'd'} is HAUUBI because its AUUB is greater than 11.

The maximal utility of a transaction can indicate the highest average-utility that any itemset included in the transaction can have. Thus, the summation of all maximal utilities of transactions including an itemset can indicate the highest average-utility that the supersets of the itemset

can have. Consequently, if a certain itemset is not HAUUBI because its AUUB is less than a given minimum utility threshold, the supersets of this itemset are not HAUIs. Based on this property, an anti-monotone property can be maintained in HAUIM.

Definition 8 (Anti-monotone property in HAUIM) Let X' be any superset of X. If $ub(X) < minutil$, $au(X') < minutil$. Therefore, the search space of HAUIM can be significantly reduced.

3 Mining high average-utility itemsets from incremental databases

In this section, we describe how the proposed method mines HAUIs from incremental databases. The algorithm uses a tree structure named IHAUI-Tree that is similar to FP-Tree employed by FP-Growth. IHAUI-Tree maintains the information of an incremental database compactly through the node sharing effect. Whenever newly generated transactions are reflected into IHAUI-Tree, a restructuring technique is applied to IHAUI-Tree in order to maximize the node sharing effect so that the proposed algorithm can save memory space. Each incremental mining process is performed based on the pattern growth approach, which is widely employed in other tree-based itemset mining algorithms because its performance is more efficient than an *Apriori-like* approach. Since our algorithm adopts the concept of a high average-utility upper-bound, the result itemsets generated from each mining process are candidate itemsets. Therefore, at the last phase of the mining process, the actual average-utilities of candidate itemsets need to be calculated through the candidate validation process requiring an additional database scan. The overall mining process of the proposed algorithm is shown in Fig. 1.

3.1 IHAUI-Tree construction

When a given incremental database is empty, IHAUI-Tree is composed of an empty header table and a tree structure having the root node without child nodes. After new transactions are accumulated in the incremental database, the information of the transactions is stored in IHAUI-Tree. The header table maintains the information of distinct items such as total AUUB values in the incremental database. On the other hand, the information of each transaction can be stored in IHAUI-Tree through transaction insertions, which create unique paths referring to the corresponding transactions in the tree. Here, sorting transactions in certain ordering manners (e.g. a support descending order) has a great impact on the compactness of the tree structure. The well-known FIM algorithm, FP-Growth, using a tree structure arranges transactions in a support descending order to maximize the node sharing effect by locating items with high supports in the upper parts of the tree. In the proposed algorithm, an AUUB descending order is employed because of the following two reasons. First, items with high AUUB values also have high supports in general because the more frequently an item is contained in transactions; the greater the AUUB of the item becomes. Therefore, by sorting transactions in an AUUB descending order the node sharing effect in the construction of IHAUI-Tree can be reasonably maximized in the same principle of the construction of FP-Tree. Second, unessential items are more likely to be excluded from the construction of local trees by constructing IHAUI-Tree based on an AUUB ascending order.

Lemma 1 *With an AUUB descending order, the unessential items can be excluded from the construction of local trees.*

Proof In HAUIM, items with insufficient AUUB values are pruned according to the anti-monotone-property. Therefore, during local tree construction, if items do not have sufficient AUUB values, they can be removed from local trees. In IHAUI-Tree, however, all information should be maintained in order to facilitate incremental mining processes. Note that the construction of local trees in the pattern growth approach is performed by extracting paths from the root node to the nodes carrying the specific items. Therefore, if items with insufficient AUUB values are positioned over items with high AUUB values in the tree, they are more likely to be included in the construction of local trees for items having sufficient AUUB values even though they need to be removed anyway. Consequently, by positioning items with low AUUB values under items with high AUUB values in IHAUI-Tree, the proposed algorithm can reduce computational overhead for constructing local trees in its mining process. □

Example 1 Assume that we build the initial IHAUI-Tree based on the transaction database shown in Table 1. Figure 2 presents the process of transaction insertions for five transactions in Table 1. We use an alphabetical order (a > b > c > d > e) because there are no previous data stored in IHAUI-Tree. Note that transactions in Table 1 are already sorted in an alphabetical order. First, we insert the first transaction, T_{100}, into IHAUI-Tree. Since items 'c', 'd', and 'e' are firstly inserted into IHAUI-Tree, the new entries are generated in the header table. In addition, since the tree does not have any path, a whole new path for T_{100} is created in the tree. In the creation of nodes, the created nodes are connected with the entries having the same items by using pointers. The AUUB values of the newly created entries and nodes are initialized to 6 because the maximal utility of T_{100} is $mu(T_{100}) = max(6, 4, 6) = 6$. Figure 2a shows

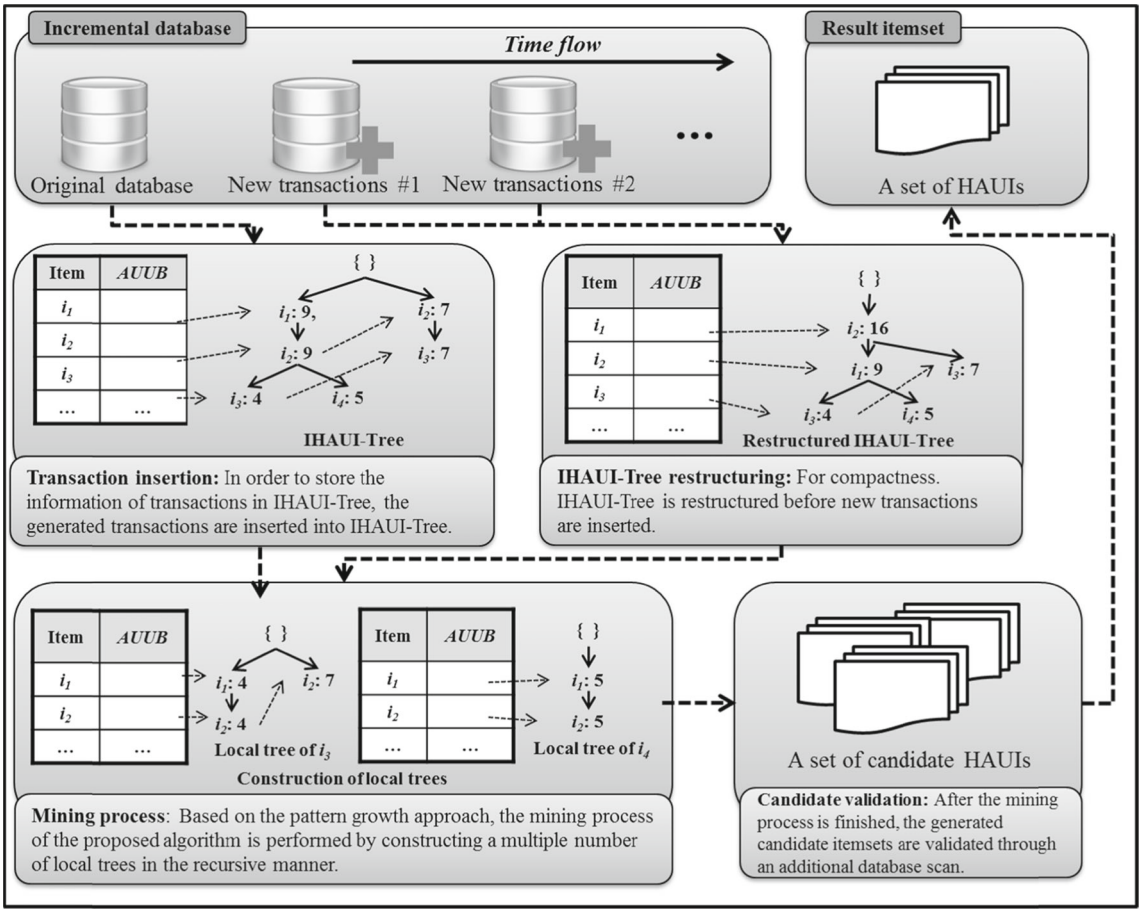


Fig. 1 Overall mining process of IMHAUI

IHAUI-Tree after T_{100} is inserted. The maximal utilities of the remaining transactions can be obtained as follows: $mu(T_{200}) = 10$, $mu(T_{300}) = 16$, $mu(T_{400}) = 15$, and $mu(T_{500}) = 7$. Based on these values, we insert the rest of transactions in IHAUI-Tree. In the insertion of T_{200} shown in Fig. 2b, the AUUB values of the existing entries carrying 'c' and 'e' are updated to 16 without the generation of new entries. On the other hand, the entries for 'a' and 'b' are newly created in the header table and their AUUB values are initialized to 10. Furthermore, a whole new path is additionally created for inserting T_{200} into the tree because there is no node carrying the first item of T_{200}, 'a', under the root node. The AUUB values of the created nodes are also initialized to 10. In the insertion of T_{300}, the node with 'a' under the root node can be shared. Therefore, without the generation of a new node for 'a', the existing node's AUUB is increased to 26. However, the rest of items, 'c' and 'd', are inserted by creating new nodes for them because there is no existing node carrying 'c' under the node with 'a'. The AUUB values of these nodes are initialized to 16. Figure 2c shows IHAUI-Tree after inserting T_{300}. In the insertion of T_{400}, the transaction can be inserted into the tree without the generation of any new node because the whole path with the nodes carrying 'a' and 'b' already exist in the tree. Figure 2d presents IHAUI-Tree after inserting T_{400} by sharing the path. When the last transaction, T_{500}, is inserted into IHAUI-Tree in the same manner as those of above insertions, we can obtain IHAUI-Tree shown in Fig. 2e.

3.2 IHAUI-Tree restructuring

IHAUI-Tree should be restructured by rearranging each path based on the optimal AUUB descending order so as to preserve its compactness. For conducting the restructuring process of IHAUI-Tree, the path adjusting method [11] is exploited. This method performs the bubble sort manner based tree restructuring process through three basic steps called "*node insert*", "*node exchange*", and "*node merge*". In the restructuring process, two adjacent nodes (parent and child nodes) in each path of IHAUI-Tree are recursively compared in order to determine whether they are sorted correctly or not. Here, there are two cases when two adjacent

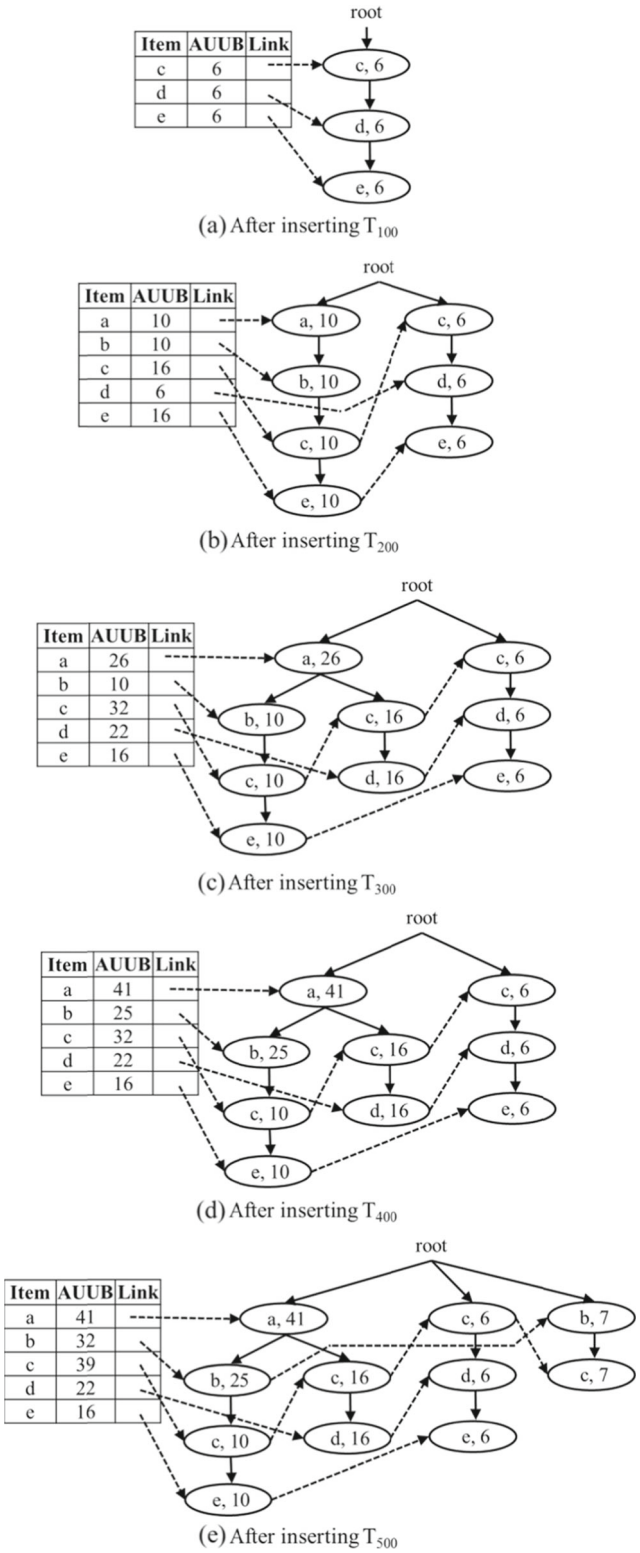


Fig. 2 Transaction insertion process for initial IHAUI-Tree

nodes need to be sorted. First, parent and child nodes can have different AUUB values. In this case, the parent node always has AUUB greater than the child node. Another case is that parent and child nodes have the same AUUB value.

Lemma 2 *In a given IHAUI-Tree, any node always has AUUB greater than or equal to those of its child nodes.*

Proof As mentioned in the description of IHAUI-Tree construction, the AUUB values of all nodes participating in a transaction insertion are increased by the maximal utility of the corresponding transaction. In the transaction insertion, each item in the transaction is inserted one by one from the root node. Therefore, if a certain node participates in a transaction insertion, all of its ancestors positioned between itself and the root node also participate in the corresponding transaction insertion. Let a set of nodes $< n_1, n_2, \ldots, n_k >$ be one of paths in a given IHAUI-Tree. n_k does not participate in a transaction insertion unless all of its ancestors $n_1, n_2, \ldots, n_{k-1}$ participate in the transaction insertion. Consequently, when the AUUB value of node n is notated as $AUUB(n)$, it is always true that $AUUB(n_k) \leq AUUB(n_{k-1}) \leq \ldots \leq AUUB(n_2) \leq AUUB(n_1)$.

If parent and child nodes have the same AUUB value, we can simply exchange the positions of the two nodes through "*node exchange*" without "*node insert*". Otherwise, an additional node carrying the item of the parent node needs to be inserted in the same level of the parent node through "*node insert*". The purpose of this process is to redistribute the AUUB value of the parent node so that the parent and child nodes can have the same AUUB value. Therefore, the AUUB value of the newly created node is set to the subtraction of AUUB values of the parent and child nodes. In this process, if the parent node has more than one child node, all child nodes except for the one to be sorted are disconnected from the current parent node and reconnected to the newly inserted node. After two adjacent nodes are rearranged, duplicated nodes carrying the same item can appear in a set of child nodes. In order to preserve the compactness of the tree, these nodes are merged through "*node merge*". A restructuring process is conducted whenever newly generated transaction data need to be reflected in IHAUI-Tree. Therefore, even though the size of an incremental database continuously becomes huge, IHAUI-Tree can minimize the memory consumption for storing data by maintaining an optimal AUUB descending order all the time. □

Example 2 Since the initial IHAUI-Tree in Example 1 was constructed on the basis of an alphabetical order, it needs to be restructured based on an optimal AUUB descending order before new transactions are inserted. The AUUB descending order of items can be obtained from the header table of IHAUI-Tree shown in Fig. 2e. Based on the AUUB values of items in the header table, the following AUUB descending order can be obtained: a > c > b > d > e (41 > 39 > 32 > 22 > 16). To sort all paths in the tree according to the obtained AUUB descending order, each path is checked one by one. We can know that the two paths

< ‘a’, ‘b’, ‘c’, ‘e’ > and < ‘b’, ‘c’ > need to be rearranged because ‘c’ should appear in advance of ‘b’ according to the obtained AUUB descending order. On the other hand, the rest of paths do not need to be rearranged because they do not include both ‘b’ and ‘c’. Note that, in real implementation, all the pairs of adjacent nodes in paths are compared to each other on the basis of the bubble sort manner in order to determine whether they are correctly sorted or not. When each node in a path can be denoted as (*item*, *AUUB*), we can sort < ‘a’, ‘b’, ‘c’, ‘e’ > by exchanging the positions of two nodes, (‘b’, 25) and (‘c’, 10). Since the parent node, (‘b’, 25), has AUUB greater than that of the child node, (‘c’, 10), we cannot simply exchange the items of two nodes. Therefore, we insert a new node that is the sibling of the parent node, and set its AUUB as the subtractions of 25 and 10, which is 15. On the other hand, the path, < ‘b’, ‘c’>, does not require “*node insert*” because the parent and child nodes have the same AUUB value. Figure 3a shows IHAUI-Tree after “*node insert*” is performed. After that, the items of two nodes in each path can be exchanged without any problem because the nodes have the same AUUB value. Figure 3b presents IHAUI-Tree after “*node exchange*” is performed by exchanging the items of nodes. We can see that the root node and (‘a’, 41) have more than one child node carrying ‘c’. Therefore, “*node merge*” is conducted for these nodes in order to preserve the compactness of IHAUI-Tree. After duplicated nodes are merged into one node, we can obtain IHAUI-Tree shown in Fig. 3c, which has a smaller number of nodes compared to IHAUI-Tree in Fig. 2e.

Example 3 Consider inserting newly generated transactions into the restructured IHAUI-Tree shown in Fig. 3c. Table 2 shows the transactions that are newly generated in the incremental database. Before inserting them into the tree, we sort

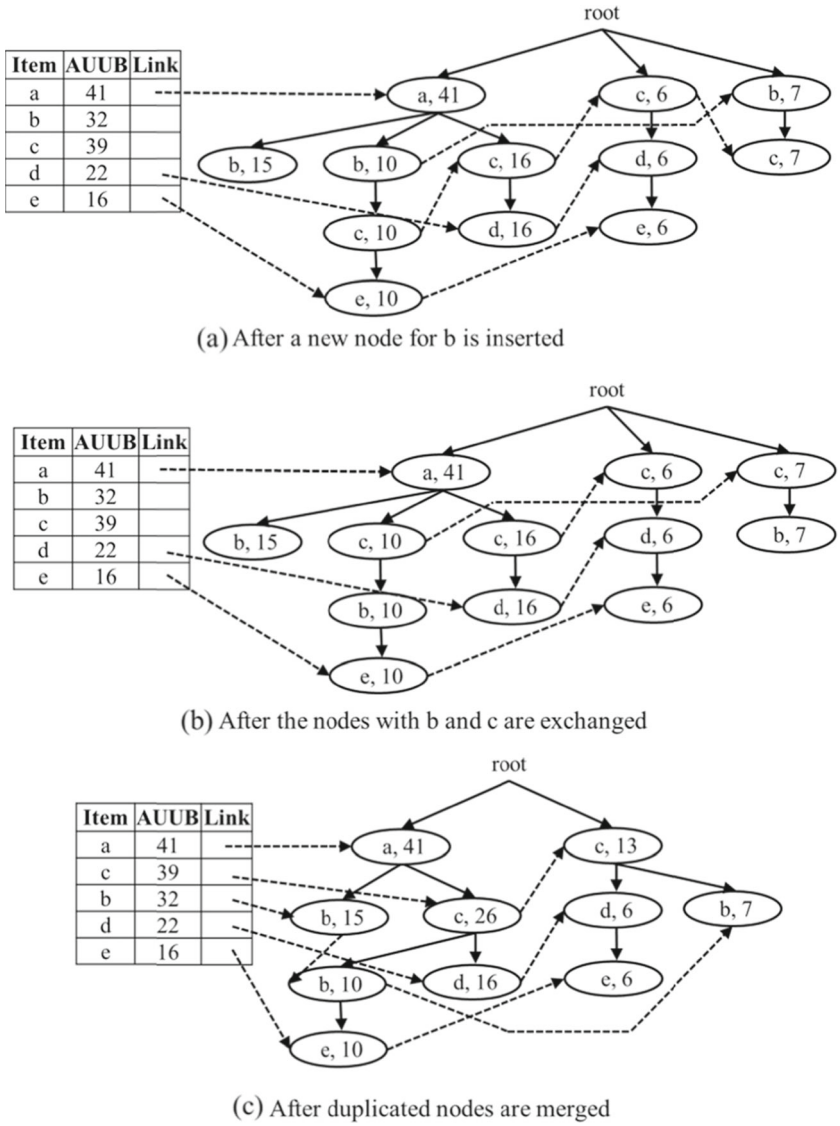


Fig. 3 IHAUI-Tree restructuring process

Table 2 New transactions

TID	Transaction
600	a(1), b(1), c(1), d(2)
700	b(1), c(3), d(2)
800	d(1), e(1)

them in the AUUB descending order (a > c > b> d > e), which is used in the restructuring process of Example 2. Therefore, we can obtain the reordered transactions as follows: T_{600} < a, c, b, d>, T_{700} < c, b, d>, and T_{800} < d, e>. Thereafter, the ordered transactions are inserted into IHAUI-Tree in the same manner as that used in the construction of the initial IHAUI-Tree. After all of three transactions in Table 2 are completely inserted into IHAUI-Tree, we can obtain IHAUI-Tree shown in Fig. 4, which maintains the information of the original transactions in Table 1 and the newly generated transactions in Table 2.

3.3 Mining HAUIs from IHAUI-Tree

The HAUI mining process of the proposed algorithm is conducted based on the pattern growth approach by exploiting only the information stored in IHAUI-Tree. Therefore, it does not require constant database scans unlike the previous algorithms. This approach builds a number of local trees for specific itemsets called prefixes. In our algorithm, local trees for specific prefixes can be constructed via the following manner. A prefix item is initially selected from the header table of IHAUI-Tree and an itemset including the selected prefix item becomes a new prefix. However, if the selected item has a total AUUB value less than the threshold in the header table, we do not construct the local tree for the corresponding prefix item according to the anti-monotone property. For constructing the local tree for the prefix, we extract a set of paths related to the selected prefix item (commonly called conditional pattern base). Each path from the node with the selected prefix item to the root node is extracted. Each extracted path has its path AUUB value, which is set to the AUUB value of the node with the prefix item. During this process, all nodes carrying a specific item can be efficiently visited through the link information embedded in IHAUI-Tree. Thereafter, the local tree is constructed based on the extracted conditional pattern base. In the construction, any item that has a total AUUB value less than the threshold is removed. During the above mining process, the generated prefixes are added to a set of candidate itemsets. After the mining process is completed, a candidate itemset validation process is performed by calculating actual average-utilities of candidate itemsets through an additional database scan.

Example 4 Consider mining HAUIs from IHAUI-Tree shown in Fig. 4. A minimum utility threshold is given as 25. To perform pattern growth operations, we first select a prefix item from the header table of IHAUI-Tree one by one. In general, the selection of prefix items follows a bottom-up sequence of the header table. Based on the bottom-up sequence, we first select 'e' as a prefix item. However, we can know that all itemsets extended from 'e' are invalid itemsets because its AUUB in the header table is 22, which is less than a minimum utility threshold. Therefore, any extension from 'e' is not considered. In addition, the prefix {'e'} is not added to a set of candidate itemsets. On the other hand, since the next prefix item, 'd', has AUUB greater than 25, we set {'d'} as the current prefix and add it to a set of candidate itemsets. Furthermore, the pattern growth operations for {'d'} are performed. To construct the local tree for {'d'} , we extract the conditional pattern base of {'d'} from IHAUI-Tree. When each extracted path is presented as $< item_1, item_2, \ldots, item_n,: path\ AUUB >$, the conditional pattern base of {'d'} is extracted as follows: <'a', 'c', 'b': 8>, < 'a', 'c' : 16 >, < 'c' : 6 >, and < 'c', 'b' : 9 >. In the conditional pattern base, items have total AUUB values as follows: 'a' = 24, 'b' = 17, 'c' = 39. Since the items 'a' and 'b' have AUUB values less than 25, these items are removed. Therefore, only 'c' remains in the conditional pattern base as follows: < 'c': 8 >, < 'c': 16 >, < 'c' : 6 >, and < 'c' : 9 >. By using this conditional pattern base, we can a build local tree for {'d'} as shown in

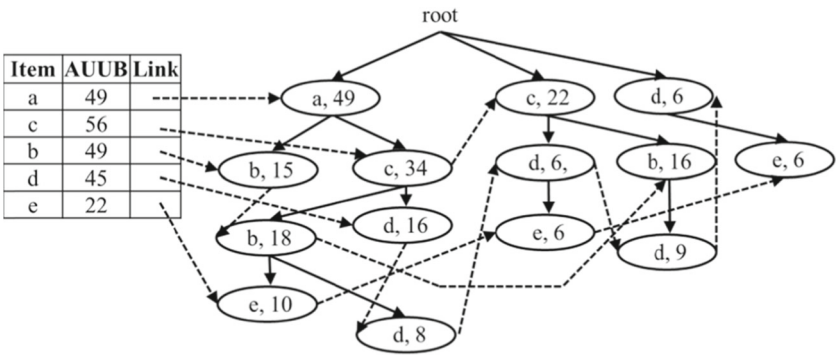


Fig. 4 IHAUI-Tree after new transactions are inserted

Fig. 5a. Thereafter, ‘c’ is recursively selected as the next prefix item. Then {‘d’, ‘c’} is added to a set of candidate itemsets and becomes a new prefix. Since there is no item in the conditional pattern base of {‘d’, ‘c’}, local trees are not constructed additionally. Therefore, the pattern growth operation for {‘d’} is terminated. Subsequently, ‘b’ is selected as the next prefix item in the header table of IHAUI-Tree. The conditional pattern base of {‘b’} can be extracted as follows: < ‘a’: 15 > , < ‘a’, ‘c’: 18>, < ‘c’: 16 >. The AUUB values of ‘a’ and ‘c’ have are 33 and 34, respectively. Therefore, any item is not removed from the conditional pattern base. The local tree for {‘b’} can be built as shown in Fig. 5b. ‘a’ and ‘c’ exist in the constructed local tree. Based on the bottom-up sequence, ‘a’ is selected as a prefix item in advance of ‘c’. Then, the prefix becomes {‘b’, ‘a’}. Therefore, we extract the conditional pattern base of {‘b’, ‘a’} from Fig. 5b as follows: < ‘c’, 33>. By inserting < ‘c’, 33> into an empty local tree, the local tree for {‘b’, ‘a’} can be constructed as shown in Fig. 5c. Since there is no more available item after ‘c’ is selected and the prefix becomes {‘b’, ‘a’, ‘c’} the current recursive process is terminated and we return to the recursive mining step of the local tree for {‘b’} . Then, ‘c’ is selected as the next prefix item. Therefore, the prefix becomes {‘b’, ‘c’}. {‘b’, ‘c’} is added to a set of candidate itemsets and the pattern growth operations for ‘b’ are finished. The pattern growth operations for ‘a’ and ‘c’ are also conducted in the same manner as those of the above processes. After the mining process is completed, we can gain the following candidate itemsets: {‘a’} , {‘c’} , {‘c’, ‘a’} , {‘b’} , {‘b’, ‘a’}, {‘b’, ‘c’}, {‘b’, ‘a’, ‘c’}, {‘d’}, and {‘d’, ‘c’}. The actual average-utilities of these candidate itemsets are calculated by rescanning the transactions in Tables 1 and 2. The actual average-utilities of the generated candidate itemsets are computed as follows: {‘a’} $= 10 + 15 + 5 * 2 = 35$, {‘c’} $= 6 + 9 + 9 + 3 + 3 + 9 = 39$, {‘c’, ‘a’} $= (19 + 14 + 8)/2 = 20.5$, {‘b’} $= 7 * 5 = 35$, {‘b’, ‘a’} $= (17 + 22 + 12)/2 = 25.5$, {‘b’, ‘c’} $= (16 * 2 + 10 * 2)/2 = 26$, {‘b’, ‘a’, ‘c’} $= (26+15)/3 = 13.7$, {‘d’} $= 4+16+8+8+4 = 40$, and {‘d’, ‘c’} $= (10+25 + 11 + 17)/2 = 31.5$. Consequently, itemsets {‘a’}, {‘c’}, {‘b’}, {‘b’, ‘a’}, {‘b’, ‘c’}, and {‘d’} having average-utilities greater than or equal to 25 are mined as actual HAUIs by the proposed algorithm.

3.4 IMHAUI algorithm description

In this part, we describe the overall procedure of the proposed algorithms through the analysis of pseudo codes. Figure 6 shows the pseudo codes presenting the overall mining process of the proposed algorithm. As shown in Fig. 6, the overall mining process is composed of three major functions named *IMHAUI*, *Restructure*, and *Mining*.

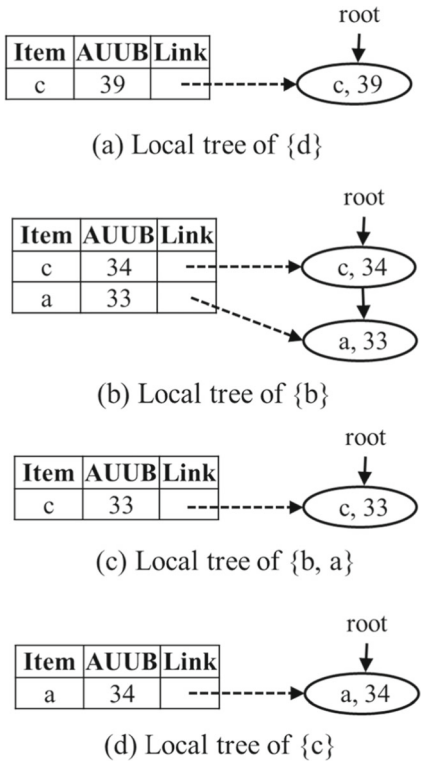


Fig. 5 Construction of local trees

The followings are the brief descriptions of these functions. 1) *IMHAUI* function: this is the main function, which continuously processes inputted transaction data and calls *Restructure* and *Mining* functions for restructuring *IHAUI-Tree* and performing the mining process. This function receives two parameters, which are an incremental database and a minimum utility threshold denoted as D and M, respectively. 2) *Restructure* function: this function is used to restructure *IHAUI-Tree* according to the optimal AUUB descending order. It receives *IHAUI-Tree* that needs to be restructured as a parameter. 3) *Mining* function: this function is used to mine candidate HAUIs based on the pattern growth approach. It receives three parameters, which are *IHAUI-Tree* (this tree is either IHAUI-Tree constructed in *IMHAUI* function or a local tree constructed in the previous recursive mining step), a minimum utility threshold, and a prefix itemset denoted as P.

First, we analyze the *IMHAUI* function shown in Fig. 6a. In the beginning, the function initializes *IHAUI-Tree* in order to accumulate the information of the incremental database throughout the procedure. At the start of each incremental mining step, if *IHAUI-Tree* is not empty, the *Restructure* function is called in order to restructure *IHAUI-Tree* based on the optimal AUUB descending order. If *IHAUI-Tree* is empty, the *Restructure* function is not called. After *IHAUI-Tree* is restructured, the newly generated transactions in the incremental database are inserted

Fig. 6 IMHAUI algorithm

```
Function 1 IMHAUI (incremental database D, minimum utility threshold M)
Initialize IHAUI-Tree
For each increment step in D //Whenever new transactions are generated
   If IHAUI-Tree is not empty
      Call Restructure(IHAUI-Tree)
   End if
   For each transaction T
      Reorder T in the AUUB descending order
      Insert T into IHAUI-Tree
   End for
   Call Mining(IHAUI-Tree, M, empty itemset)
   Validate candidate itemsets and provide mining results
End for
```

(a) *IMHAUI*

```
Function 2 Restructure(IHAUI-Tree)
Decide the optimal AUUB descending order based on the header table of IHAUI-Tree
For each path P in IHAUI-Tree
   For each pair of nodes parent and child in P //Sort nodes in P by comparing two
   adjacent nodes based on the bubble sort manner
      If parent and child have been incorrectly ordered
         If parent's AUUB is greater than child's AUUB
            Perform "node Insert"
         End if
         Perform "node exchange"
         If duplicated nodes exist
            Perform "node merge"
         End if
      End if
   End for
End for
```

(b) *Restructure*

```
Function 3 Mining(IHAUI-Tree, minimum utility threshold M, prefix itemset P)
For each item i in the header table of IHAUI-Tree
   If i has AUUB no less than M in the header table
      Record i ∪P as a candidate itemset
      //Construction of the local tree for i ∪P
      Extract conditional pattern base of i from IHAUI-Tree
      Remove items with AUUB values less than M from conditional pattern base
      Construct local_IHAUI-Tree based on conditional pattern base
      Call Mining(local_IHAUI-Tree, M, i ∪P)
   End if
End for
```

(c) *Mining*

into *IHAUI-Tree* one by one through the loop operation. Each transaction is sorted in the AUUB descending order determined by the above restructuring process. After the information of the newly generated transactions is reflected into *IHAUI-Tree*, the *Mining* function is called in order to perform the mining process. Finally, candidate itemsets generated during the mining process are validated through an additional database scan. Then, itemsets satisfying a minimum utility threshold condition are provided as mining results. This routine (from a restructuring process to the mining process) is repeated in every incremental mining step.

The *Restructure* function presented in Fig. 6b sorts each path of *IHAUI-Tree* in an optimal AUUB descending order decided based on the total AUUB values of items in the header table. For each path, all pairs of two adjacent nodes are compared with each other in order to determine whether they are correctly ordered or not. For each pair of adjacent nodes, the following restructuring steps are applied. If a parent node has AUUB greater than that of a child node, the "*node insert*" process is performed. After that, "*node exchange*" is performed in order to change the order of two nodes. Finally, when the identifiers of two nodes are exchanged, the existence of duplicated nodes is checked and "*node merge*" is performed if there are duplicated nodes.

The *Mining* function is a core function for performing the pattern growth operations. This function builds local trees for prefixes and recursively calls itself so as to perform the pattern growth operations extending prefixes progressively. At the first part of the algorithm each item in the header table of *IHAUI-Tree* is selected one by one and checked whether it has AUUB greater than or equal to a minimum utility threshold. If the item has AUUB no less

than a minimum utility threshold, the next prefix is generated by combining the selected prefix item and the previous prefix itemset. The next prefix is added to a set of candidate itemsets and the local tree for the next prefix is constructed based on the conditional pattern base extracted from *IHAUI-Tree*. Here, all unessential items which have AUUB values less than a minimum utility threshold are discarded from the conditional pattern base. Thereafter, *Mining* function is recursively called by using the next prefix itemset and the constructed local tree as parameters. The above recursive processes are iterated until no valid item exists in the newly constructed local tree.

3.5 Time complexity analysis

Let us calculate the time complexity of the proposed algorithm when an incremental mining process is performed. Let n be the number of distinct items contained in an incremental database after new transactions with $\frac{n}{2}$ distinct items being inserted into the database and l be the maximum length of itemsets generated from the database. Therefore, scanning the whole database requires the time complexity of $O(n)$ for processing all items. The proposed algorithm scans the database two times in order to construct IHAUI-Tree and validate candidate itemsets. Assume that initial IHAUI-Tree is built before new transactions are added into the database. Therefore, the proposed algorithm scans only new transactions for inserting them into IHAUI-Tree. However, for validating candidate itemsets, the whole database needs to be checked. Therefore, the proposed algorithm requires the time complexity of $O(\frac{n}{2} + n)$ for scanning the database during its entire incremental mining process. It can be simplified as $O(\frac{3n}{2})$. On the other hand, ITPAU generates candidate itemsets at each level and scans the database for identifying AUUB values of candidate itemsets. In addition, one database scan is required for calculating actual average-utilities of candidates. Even though it utilizes the information of itemsets generated in the previous mining process, the whole database needs to be scanned for calculating AUUB of newly generated itemsets. Therefore, ITPAU requires the time complexity of $O((l + 1) \times n)$ for scanning the database. Since $l + 1$ is greater than $\frac{3}{2}$ in general, we can determine that the proposed algorithm requires less time for scanning the database less than ITPAU theoretically.

In each incremental mining process, the proposed algorithm restructures IHAUI-Tree based on an AUUB descending order before conducting its mining process. Since the path adjusting method sorts each path based on the bubble sort manner, the time complexity for restructuring IHAUI-Tree with n nodes is $O(n^2)$ in the worst case. In addition, the proposed algorithm constructs multiple local trees recursively on the basis of the pattern growth approach for mining candidate itemsets. For each item in IHAUI-Tree, a new prefix for the item is generated and its local tree is constructed if AUUB of the item in the header table is greater than or equal to *minutil*. The worst case assumes that all items satisfy the *minutil* condition so that the number of constructed local trees is maximized. In order to construct a local tree, the conditional pattern base is extracted from IHAUI-Tree by visiting node associated with i. Let m ($m < n$) be the number of nodes participating in the construction of a local tree of i and R_i be the time for the recursive pattern growth operation of i's local tree. Based on the information of m nodes extracted from IHAUI-Tree, a local tree with m nodes is constructed. Therefore, in order to construct local trees for n items in IHAUI-Tree, the algorithm requires the time complexity of $O(2 \times m \times n)$. Since the recursive pattern growth operation is performed for each local tree constructed from IHAUI-Tree, the time complexity of the entire mining process is $O\left(n^2 + 2 \times m \times n + \sum_{i=1}^{n} R_i\right)$. Each R_i can be expressed as follows. Since the local tree of i is composed of m items, m local trees can be constructed from the local tree of i. In the construction of these local trees, m' ($m > m'$) nodes participate. Therefore, $R_i = O\left(2 \times (m') \times m + \sum_{j=1}^{m} R_{ij}\right)$. As shown in the above case, since time for the recursive pattern growth operations of local trees gradually becomes smaller as the depths of the recursive pattern growth operations are increased, the time complexity of the entire mining process can be simplified as $O(n^2)$ by considering only the highest order term.

On the other hand, ITPAU generates $(k + 1)$-itemsets by joining k-itemsets. Therefore, ITPAU has the worst time complexity when it has to generate all possible combinations that can be generated from items in the database. Let L_k be the number of generated itemsets with k-lengths and ${}_kC_i$ be the number of combinations of k items in groups of i items. That is, $L_1 = {}_kC_1$, $L_2 = {}_kC_2, \ldots, L_k = {}_kC_k$. Since the number of items in the database is n, the number of all possible combinations that can be generated from the database is ${}_nC_1 + {}_nC_2 + \ldots + {}_nC_n = 2^n - 1$ according to the combination formula. Therefore, for generating all possible itemsets, ITPAU requires the time complexity of $O(2^n)$. Consequently, since $O(2^n) > O(n^2)$, the proposed algorithm can finish its mining process much faster than ITPAU theoretically.

In the above time complexity analysis, we determine that the proposed algorithm outperforms the state-of-the-art incremental HAUIM algorithm, ITPAU, theoretically. However, it is not enough to show that the proposed algorithm substantially has better performance than ITPAU in the mining processes using real data. Therefore, in the next section, we evaluate the performance of the proposed algorithm based on various experiments conducted on real datasets.

4 Performance evaluation

4.1 Experimental settings

We implemented the two algorithms, ITPAU [10] and HUPID [34], in order to compare their performances with that of the proposed algorithm, HUPID, which is an HUIM algorithm for processing incremental databases. This is one of the most recent incremental HUIM algorithms using tree structures and pattern growth approaches similarly to ours. Therefore, we can show the differences between HUIM and HAUIM by comparing this algorithm with the proposed algorithm. On the other hand, ITPAU is a state-of-the-art incremental HAUIM algorithm employing an *Apriori-like* approach and the itemset information maintenance technique. We can thus prove that the proposed algorithm is the most efficient incremental HAUIM algorithm through the comparison of ITPAU and the proposed algorithm. Our algorithm and HUPID perform tree restructuring processes before beginning each incremental mining process. In the construction of tree structures and restructuring processes, the proposed algorithm and HUPID use an AUUB descending order and a TWU descending order, respectively. We made all algorithms in C++ programming language. In addition, a PC with 3.30GHz CPU, 8GB RAM, and the Windows 7 64bit operating system was employed for performing tests. We used the various datasets including both real and synthetic datasets in experiments Table 3 provides the detailed information of datasets, Chain-store, Foodmart, Mushroom, Breast-cancer Wisconsin, the group of T10I4Dx datasets, and the group of $Tx_1Nx_2Lx_3$ datasets.

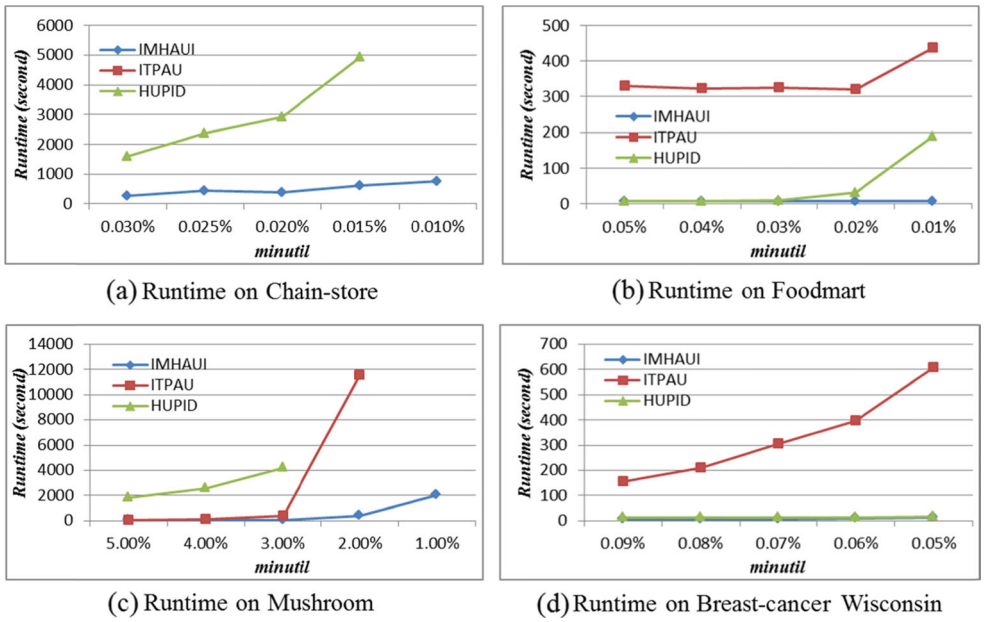
Chain-store and Foodmart are datasets collected from major grocery stores and have actual utility data. On the other hand, Mushroom and Breast-cancer Wisconsin [3] are datasets with information of mushroom species and patients related to breast cancer diseases, respectively. They have arbitrary utility values. Chain-store can be obtained from NU-MineBench [27] version 2.0. Foodmart is contained in the Microsoft Foodmart 2000 database. The rest of real datasets, Mushroom and Breast-cancer Wisconsin, can be gained from UCI Machine Learning repository (https://archive.ics.uci.edu/ml/index.html). The group of T10I4Dx datasets has characteristics in that the number of transactions is increased in proportion to the increase of x. On the other hand, the group of $Tx_1Nx_2Lx_3$ datasets has characteristics in that and the number of items, the average length of transactions, and the number of discoverable itemsets are raised according to the variables x_1, x_2, and x_3, respectively. All synthetic datasets were generated by the IBM generator [1]. We conducted experiments for evaluating the performances of algorithms in terms of their runtime, memory usage, and scalability. The tests on runtime and memory usage were conducted on real datasets while the synthetic datasets were employed for the scalability tests.

4.2 Performance evaluation under different minimum utility thresholds

We first compare the runtime performances of algorithms. Figure 7 shows the experimental results of the runtime tests conducted on the real datasets. The tests were performed by gradually decreasing the minimum utility threshold denoted

Table 3 Datasets

Dataset	Number of transactions	Number of items	Average length	Number of discoverable itemsets
Chain-store	1,112,950	46,087	7.2	–
Foodmart	4,141	1,559	4.4	–
Mushroom	8,124	119	23	–
Breast-cancer Wisconsin	699	92	10	–
T10I4D100K	100,000	1,000	10	–
T10I4D200K	200,000	1,000	10	–
T10I4D400K	400,000	1,000	10	–
T10I4D600K	600,000	1,000	10	–
T10I4D800K	800,000	1,000	10	–
T10I4D1000K	1,000,000	1,000	10	–
T10N10000L1000	100,000	10,000	10	1000
T20N20000L2000	100,000	20,000	20	2000
T30N30000L3000	100,000	30,000	30	3000
T40N40000L4000	100,000	40,000	40	4000



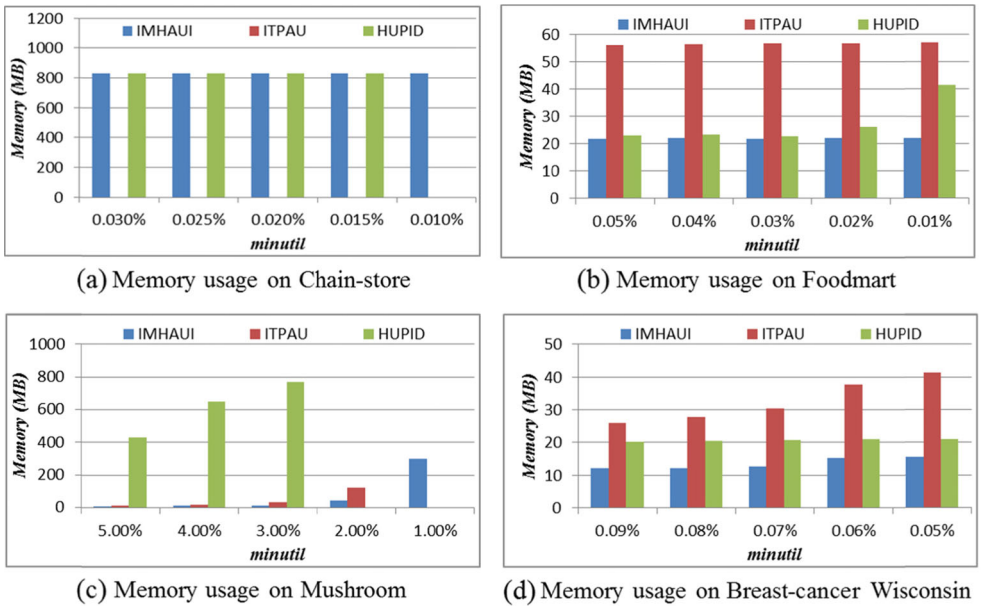
(a) Runtime on Chain-store (b) Runtime on Foodmart

(c) Runtime on Mushroom (d) Runtime on Breast-cancer Wisconsin

Fig. 7 Runtime performance under different minimum utility thresholds

as *minutil*. All algorithms perform their incremental mining processes whenever 20 % of the whole dataset are newly read. The runtime of algorithms was measured on the total execution time for performing all incremental mining processes. In figures, the y-axis shows the changes of the algorithms' execution times expressed in seconds while the x-axis shows the variation of *minutil* conditions. In the experimental results, we can observe that the runtime performances of algorithms become worse as *minutil* is decreased. The reason why the runtime performances are degraded according to the decrease of the threshold is that the number of generated itemsets is increased as the threshold becomes lower. In the experiment on Chain-store, Fig. 7a, the mining process of ITPAU was unfinished at all thresholds because it required huge execution times (over 15000 s) in order to constantly scan the entire data of Chain-store, which is a very huge dataset. Note that, therefore, Fig. 7a does not show the runtime performance of ITPAU. In addition, HUPID causes a memory overflow at threshold 0.01 % because it generates a huge number of candidate itemsets while the proposed algorithm generates much less candidate itemsets. Figure 7b shows the result of the runtime experiment on Foodmart. In Fig. 7b, we can see that the proposed algorithm has much better runtime performance than ITPAU in all cases. In particular, the runtime requirement of ITPAU significantly increases at the 0.01 % threshold. In the experimental results of the test on Mushroom, Fig. 7c, the runtime performances of HUPID and ITPAU cannot be measured at thresholds less than 3 % and 2 %, respectively. The reason is that, at low thresholds, HUPID causes memory overflows and ITPAU consumes excessive runtime to finish its mining process. Even though HUPID generates more itemsets than ITPAU because it is an HUIM algorithm, it generally has better runtime performance than ITPAU because it uses a tree structure in order to avoid constant database scanning. However, if the number of generated HUIs overwhelms the number of generated HAUIs, the performance of HUPID can be worse than that of ITPAU. For example, in Fig. 7c, HUPID has runtime performance worse than ITPAU. On the other hand, the proposed algorithm always has the best runtime performance in Fig. 7c. The experimental result of the test on Breast-cancer Wisconsin, Fig. 7d, also shows that the runtime performance of the proposed algorithm is much better than that of ITPAU.

Next, we compare the performances of algorithms in terms of memory space consumed by the algorithms. Figure 8 shows the experimental results of the memory performance tests conducted on four real datasets. These tests were conducted under the same experimental conditions as those of the above runtime tests. The memory performances of algorithms were measured on algorithms' peak memory usage during their entire incremental mining processes. The y-axis in figures indicates the memory usage of algorithms expressed in Megabytes. We can observe that the memory performances of algorithms become worse as *minutil* is decreased similarly to the aspect of the runtime performances of algorithms. The reason is that the number of generated candidates is raised according to the decrease of the threshold and the generated candidate itemsets should be stored in the main memory in order to be handled in candidate validation processes. In the result of the memory test on Chain-store, Fig. 8a we can see that IMHAUI and HUPID require somewhat similar memory usage. The reason is that both of them have to use a huge amount



(a) Memory usage on Chain-store (b) Memory usage on Foodmart

(c) Memory usage on Mushroom (d) Memory usage on Breast-cancer Wisconsin

Fig. 8 Memory performance under different minimum utility thresholds

of memory in order to maintain all information of Chain-store in their tree structures. However, HUPID causes a memory overflow at the threshold, 0.01 %, while IMHAUI finishes its mining process normally at the same threshold. The reason is that HUPID generates much more candidate itemsets than IMHAUI because it is an HUIM algorithm. Figure 8b shows the memory usage of algorithms in the mining processes on Foodmart. From the figure, we can learn that the proposed algorithm always has the highest memory performance regardless of the *minutil* condition. On the other hand, ITPAU has the worst memory performance because a huge number of candidate itemsets are generated in its mining process. Unlike the result in Fig. 8b, HUPID requires the largest memory space at all cases in Fig. 8c because the number of candidate HUIs generated from Mushroom is much greater than the number of candidate HAUIs. Therefore, HUPID causes memory overflows when *minutil* becomes lower than 3 %. In Fig. 8c and d, we can observe that the proposed algorithm has the best memory performance at all *minutil* settings.

4.3 Performance evaluation under different transaction insertion ratios

In the experiments for the performance evaluation in Section 4.2, we fixed the transaction insertion ratios of algorithms' incremental mining processes to 20 %. However, the change of a transaction insertion ratio is an important factor that determines the performance of algorithms in incremental mining environments. Therefore, we conducted experiments under different transaction insertion ratios in order to evaluate the runtime and memory performances of the proposed algorithm. Figure 9 shows the results of the runtime tests conducted under different transaction insertion ratios. These tests were performed by decreasing the transaction insertion ratio from 40 % to 10 %. The *minutil* settings used in the experiments on Chain-store, Foodmart, Mushroom, and Breast-cancer Wisconsin were fixed to 0.03 %, 0.05 %, 3 %, and 0.09 %, respectively. We can see that the execution times of algorithms are increased as the transaction insertion ratio is decreased. The reason is that the number of mining processes performed in an incremental mining process is increased as the transaction insertion ratio is reduced. When a transaction insertion ratio is 10 %, the algorithms have to perform mining processes 10 times throughout the duration of their entire incremental mining process. Therefore, we can learn that the algorithms require the highest execution time when the transaction insertion ratio is 10 %. In Fig. 9a, the runtime of ITPAU is not shown because it consumes too much execution time. In addition, the runtime of HUPID is not shown in Fig. 9c because it causes memory overflows at every transaction insertion ratio. On the other hand, we can observe that the proposed algorithm always has much better performance than ITPAU regardless of a transaction insertion ratio in all experimental results.

Figure 10 presents the results of the memory experiments conducted under different transaction insertion ratios. The experimental settings are same as those of the above runtime tests. In Fig. 10a, the proposed algorithm and HUPID consume similar memory space as in the case of the result in Fig. 8a. We can see that ITPAU generally requires more memory as the transaction insertion ratio is decreased

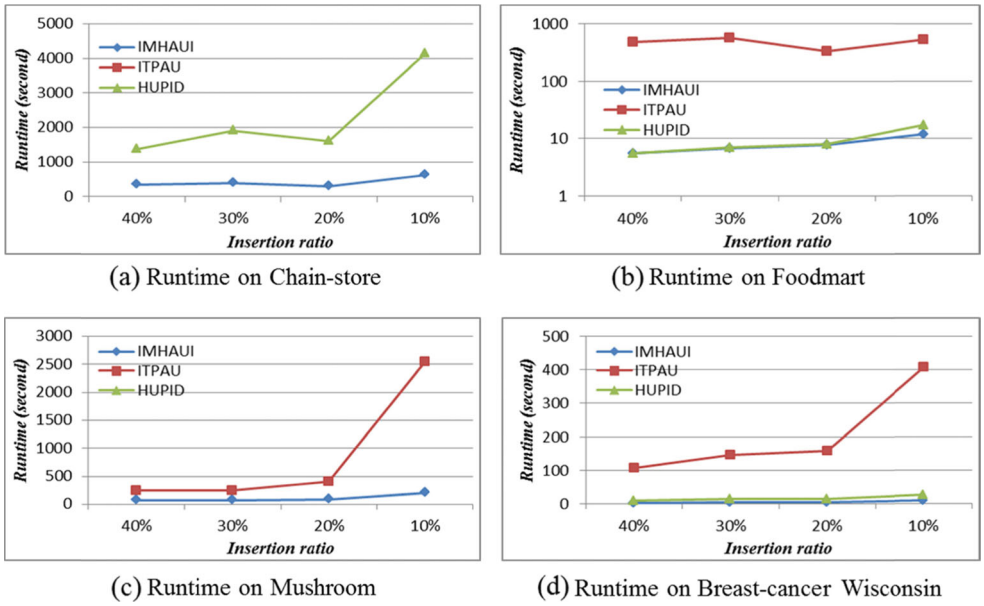


Fig. 9 Runtime performance under different transaction insertion ratios

because more memory space is used to maintain the information of itemsets generated during each mining process. On the other hand, we can determine that the proposed algorithm has the best memory performance at all transaction insertion ratios in Fig. 10b, c, and d.

4.4 Scalability performance evaluation

In this subsection, we evaluate the scalability performance of our algorithm, which indicates how well the algorithm can handle incremental databases when their scales such as the numbers of transactions and distinct items are significantly increased. Figure 11 presents the experimental results of scalability tests conducted on the two groups of synthetic datasets (T10I4Dx and $Tx_1Nx_2Lx_3$). In the test on T10I4Dx, the number of transactions contained in a dataset gradually increases from 100 K to 1000 K. On the other hand, in the test on $Tx_1Nx_2Lx_3$, the numbers of distinct items, the average length of transactions, and the number of discoverable itemsets are increased from 10000, 10, and 1000 to 40000, 40, and 4000, respectively. In the both of tests, *minutil* was fixed to 0.1 % for the entirety of the test. We can observe that the runtime performances of algorithms generally become worse as the numbers of transactions and

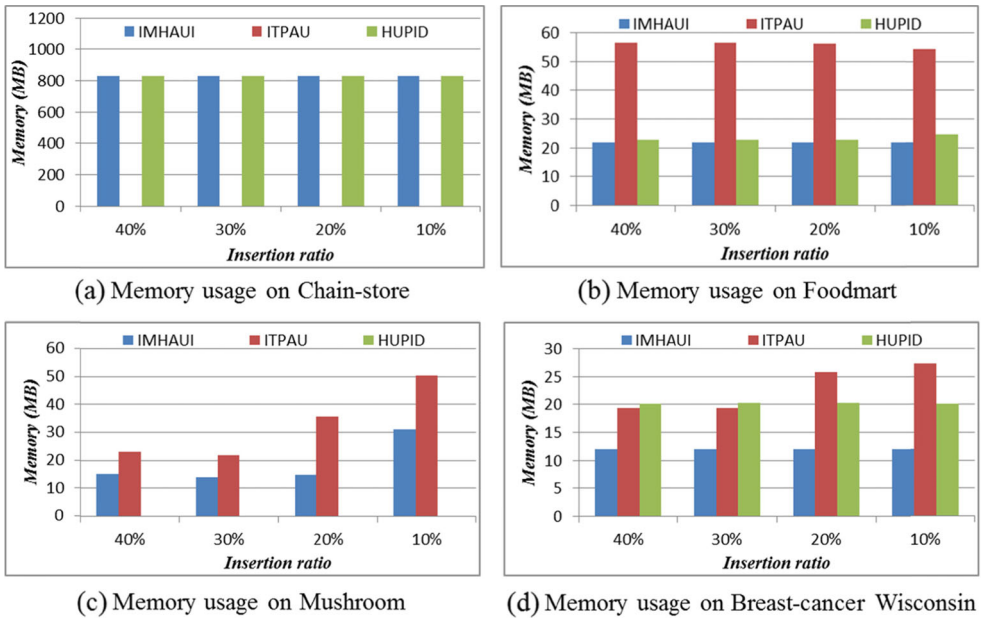
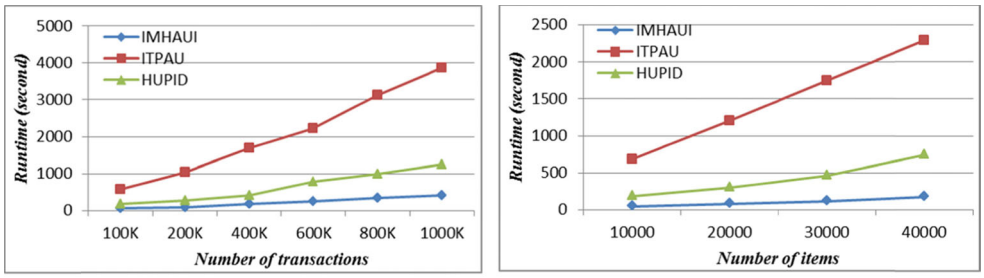


Fig. 10 Memory performance under different transaction insertion ratios

Fig. 11 Scalability performance



(a) T10I4Dx (b) $Tx_1Nx_2Lx_3$

items are raised. However, we can also see that the performance of IMHAUI slowly becomes worse while the performances of others worsen much faster than that of IMHAUI. In particular, the execution time of ITPAU is increased much faster than those of the proposed algorithm and HUPID because the large number of database scans causes dramatic performance deterioration when the numbers of transactions and distinct items are huge. The performance of HUPID also becomes worse faster than that of IMHAUI because the number of generated HUIs is increased significantly as the scales of databases are expanded while the number of generated HAUIs is raised gently. Overall, we can know that the proposed algorithm has the best scalability performance among the compared algorithms.

5 Conclusions

In this paper, we proposed a novel algorithm for mining HAUIs from incremental databases, which is much more efficient than state-of-the-art algorithms. Unlike the previous ones, the proposed algorithm can generate candidate itemsets without a number of database scans because all necessary data of a given incremental database can be stored in a compact tree data structure. In addition, our algorithm periodically performs a restructuring process based on the path adjusting method in order to preserve the compactness of its data structure. In the performance evaluation, we showed that the proposed algorithm outperformed competitors in terms of runtime, memory usage, and scalability performances. However, the validation of candidate itemsets still requires excessive runtime, and this limitation can be a crucial obstacle in mining tasks for incremental databases with the rapid generation of new transaction data. Recently, various algorithms have been proposed to mine itemsets from static databases without the generation of candidate itemsets. They employ list structures for capturing actual utility information so that the validation of candidate itemsets is not required. Therefore, based on such recent studies, we are scheduled to conduct researches on the techniques for mining HAUIs from incremental databases without the generation of candidate itemsets in our future works.

Acknowledgments This research was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF No. 20152062051 and NRF No. 20155054624), and the Business for Academic-industrial Cooperative establishments funded Korea Small and Medium Business Administration in 2015 (Grants No. C0261068).

References

1. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: 20th international conference on very large data bases, pp 487–499
2. Ahmed CF, Tanbeer SK, Jeong B, Lee Y (2009) Efficient tree structures for high utility pattern mining in incremental databases. IEEE Trans Knowl Data Eng 21(12):1708–1721
3. Bennett KP, Mangasarian OL (1992) Robust linear programming discrimination of two linearly inseparable sets. Optim Methods Software 1:23–34
4. Cheung DW, Han J, Ng VT, Wong CY (1996) Maintenance of discovered association rules in large databases: an incremental updating approach. In: The 12th IEEE international conference on data engineering, pp 106–114
5. Duong Q, Liao B, Fournier-Viger P, Dam T (2016) An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies. Knowl-Based Syst 104:106–122
6. Fournier-Viger P, Wu C, Zida S, Tseng V (2014) FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. In: ISMIS, pp 83–92
7. Fan Y, Ye Y, Chen L (2016) Malicious sequential pattern mining for automatic malware detection. Expert Syst Appl 52:16–25
8. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, pp 1–12
9. Hong T, Lee C, Wang S (2011) Effective utility mining with the measure of average utility. Expert Syst Appl 38(7):8259–8265
10. Hong T, Lee C, Wang S (2009) An incremental mining algorithm for high average-utility itemsets. In: ISPAN 2009, pp 421–425
11. Koh J, Shieh S (2003) An efficient approach for maintaining association rules based on adjusting FP-tree structures. In: DASFAA, pp 417–424
12. Krishnamoorthy S (2015) Pruning strategies for mining high utility itemsets. Expert Syst Appl 42(5):2371–2381
13. Kim D, Yun U (2016) Efficient mining of high utility pattern with considering of rarity and length. Appl Intell 45(1):152–173
14. Kim D, Yun U (2016) Mining high utility itemsets based on the time decaying model. Intell Data Anal 20(5):1157–1180
15. Lan G, Hong T, Tseng V (2012) A projection-based approach for discovering high average-utility itemsets. J Inf Sci Eng 28:193–209

16. Lan G, Hong T, Tseng V (2012) Efficiently mining high average-utility itemsets with an improved upper-bound strategy. Int J Inf Technol Decis Making 11(5):1009–1030
17. Le T, Vo B (2015) An N-list-based algorithm for mining frequent closed patterns. Expert Syst Appl 42(19):6648–6657
18. Lee G, Yun U, Ryu K (2014) Sliding window based weighted maximal frequent pattern mining over data streamss. Expert Syst Appl 41(2):694–708
19. Lee G, Yun U, Ryang H (2015) An uncertainty-based approach: frequent itemset mining from uncertain data with different item importance. Knowl-Based Syst 90:239–256
20. Lee G, Yun U, Ryang H, Kim D (2016) Approximate maximal frequent pattern mining with weight conditions and error tolerance. Int J Pattern Recognit Artif Intell 30(6):1–42
21. Lee G, Yun U, Ryang H, Kim D (2016) Erasable itemset mining over incremental databases with weight conditions. Eng Appl Artif Intell 52:213–234
22. Lin J, Gan W, Hong T, Tseng V (2015) Efficient algorithms for mining up-to-date high utility patterns. Adv Eng Inform 29(3):648–661
23. Lin J, Gan W, Fournier-Viger P, Hong T, Tseng V (2016) Efficient algorithms for mining high-utility itemsets in uncertain databases. Knowl-Based Syst 96:171–187
24. Liu Y, Liao W, Choudhary A (2005) A two-phase algorithm for fast discovery of high utility itemsets. In: Advances in knowledge discovery and data mining, pp 689–695
25. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM international conference on Information and knowledge management, pp 55–64
26. Lu T, Vo B, Nguyen HT, Hong T (2014) A new method for mining high average utility itemsets. In: Computer Information Systems and Industrial Management, pp 33–42
27. Pisharath J, Liu Y, Ozisikyilmaz B, Narayanan R, Liao WK, Choudhary A Memik G NU-MineBench version 2.0 dataset and technical report, http://cucis.ece.northwestern.edu/projects/DMS/
28. Ryang H, Yun U (2015) Top-K high utility pattern mining with effective threshold raising strategies. Knowl-Based Syst 76:109–126
29. Ryang H, Yun U, Ryu K (2016) Fast algorithm for high utility pattern mining with sum of item quantities. Intell Data Anal 20(2):395–415
30. Tseng V, Shie BE, Wu CW, Yu PS (2013) Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans Knowl Data Eng 25(8):1772–1786
31. Tseng V, Wu C, Fournier-Viger P, Yu PS (2016) Efficient algorithms for mining top-K high utility itemsets. IEEE Trans Knowl Data Eng 28(1):54–67
32. Tanbeer SK, Ahmed CF, Jeong B, Lee Y (2009) Efficient single-pass frequent pattern mining using a prefix-tree. Inf Sci 179(5):559–583
33. Tsai C, Lai B (2015) A location-item-time sequential pattern mining algorithm for route recommendation. Knowl-Based Syst 73:97–110
34. Yun U, Ryang H (2015) Incremental high utility pattern mining with static and dynamic databases. Appl Intell 42(2):323–352
35. Yun U, Ryang H, Ryu K (2014) High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates. Expert Syst Appl 41(8):3861–3878
36. Yun U, Kim D, Ryang H, Lee G, Lee K (2016) Mining recent high average utility patterns based on sliding window from stream data. J Intell Fuzzy Syst 30(6):3605–3617
37. Yun U, Lee G (2016) Incremental mining of weighted maximal frequent itemsets from dynamic databases. Expert Syst Appl 54:304–327
38. Yun U, Lee G (2016) Sliding window based weighted erasable stream pattern mining for stream data applications. Futur Gener Comput Syst 59:1–20
39. Yun U, Lee G, Kim C (2016) The smallest valid extension-based efficient, rare graph pattern mining, considering length-decreasing support constraints and symmetry characteristics of graphs. Symmetry 8(5):1–26
40. Yun U, Pyun G, Yoon E (2015) Efficient mining of robust closed weighted sequential patterns without information loss. Int J Artif Intell Tools 24(1):1–28
41. Yun U, Lee G, Lee K (2016) Efficient representative pattern mining based on weight and maximality conditions. Expert Syst 33(5):439–462
42. Zhang J, Wang Y, Yang D (2015) CCSpan: mining closed contiguous sequential patterns. Knowl-Based Syst 89:1–13
43. Zhang X, Deng Z (2015) Mining summarization of high utility itemsets. Knowl-Based Syst 84:67–77