applied sciences

MDPI

Article

Maintenance of Discovered High Average-Utility Itemsets in Dynamic Databases

Binbin Zhang [1,2], Jerry Chun-Wei Lin [3,4,*], Yinan Shao [3], Philippe Fournier-Viger [5] and Youcef Djenouri [6]

1 Department of Biochemistry and Molecular Biology, Shenzhen University Health Science Center, Shenzhen 518060, China; zhangbb@szu.edu.cn
2 Center for Anti-Aging and Regenerative Medicine, Shenzhen University Health Science Center, Shenzhen 518060, China
3 School of Computer Science and Technology, Harbin Institute of Technology Shenzhen Graduate School, Shenzhen 518055, China; shaoyn0817@gmail.com
4 Department of Computing, Mathematics, and Physics, Western Norway University of Applied Sciences (HVL), Bergen 5063, Norway
5 School of Natural Sciences and Humanities, Harbin Institute of Technology Shenzhen Graduate School, Shenzhen 518055, China; philfv@hitsz.edu.cn
6 IMADA Lab, Southern Denmark University, Odense 5230, Denmark; y.djenouri@gmail.com
* Correspondence: jerrylin@ieee.org; Tel.: +86-755-8654-0396

Received: 13 April 2018; Accepted: 8 May 2018; Published: 11 May 2018

Abstract: High-utility itemset mining (HUIM) is an extension of traditional frequent itemset mining, which considers both quantities and unit profits of items in a database to reveal highly profitable itemsets regardless of their size. High average-utility itemset mining (HAUIM) is designed to find average-utility itemsets by considering both their utility and the number of items that they contain. Thus, average-utility itemsets are obtained based on a fair utility measurement since the average utility typically does not increase much with the size of itemsets. However, most algorithms for discovering high average utility itemsets are designed to extract patterns from a static database. If the size of a database decreases or increases over time (e.g., as a result of transaction insertions), the database must be scanned again in batch mode to update the results. Thus, previously discovered knowledge is ignored and the time previously spent for pattern extraction is wasted. We thus present an incremental HAUIM algorithm for transaction insertion (FUP-HAUIMI) to maintain information about patterns when a database is updated, based on the FUP concept. An average-utility-list (AUL)-structure is first built by scanning the original database. Then, FUP-HAUIMI selects high average-utility upper-bound itemsets and categorizes them according to four cases. For each case, itemsets are maintained and updated using a specific updating procedure. While traversing the enumeration tree representing the search space in a depth-first way, a join operation is performed to quickly and incrementally update the AUL-structures. Several experiments were carried to evaluate the runtime, memory usage, number of potential patterns (candidates), and the scalability of the designed approach. Results show that the performance of FUP-HAUIMI is excellent compared to the state-of-the-art HAUI-Miner algorithm running in batch mode and the state-of-the-art incremental high-utility pattern mining (IHAUPM) algorithm for incremental average-utility pattern mining.

Keywords: high average-utility mining; dynamic database; transaction insertion; FUP

1. Introduction

Mining useful or meaningful information is a major KDD (Knowledge Discovery in Database) task, which has been widely considered as interesting and useful for more than two decades. Association rule mining (ARM) is a basic KDD problem, designed to reveal correlations between purchased items in customer transactional records. A fundamental algorithm named Apriori [1] was first designed to mine association rules (ARs). It discovers patterns in a level-wise way in a static database. Initially, it finds the set of frequent itemsets based on a parameter called the minimum support threshold. Then, Apriori derives the set of ARs from these itemsets according to a second parameter called the minimum confidence threshold. Although this approach can find all the desired patterns, it can have very long runtimes and huge memory requirements. Based on this observation, the Frequent Pattern (FP)-growth [2] algorithm was proposed. It speeds up ARM by keeping information required for mining ARs in a compressed FP-tree structure. The FP-tree structure stores frequent items, which are later used for mining patterns in a recursive way, growing patterns from small patterns to longer patterns, without generating candidate patterns and using a depth-first search. Deng et al. [3] then introduced a new vertical data structure called N-list, which keeps the required information for mining patterns, and an efficient PrePost algorithm to mine the set of frequent patterns. Several algorithms used in different domains and applications [4–7] were developed to efficiently find different types of itemsets, respectively.

Most approaches for ARM can only be applied on binary databases. Thus, it is considered that all items are equally interesting or important. The internal values (such as the purchase quantities of items) or the external values (such as the unit profits of items) are not taken into account in ARM. For this reason, ARM is unable to specifically reveal the profitable itemsets. Consequently, retailers or managers cannot take efficient decisions or choose appropriate sales strategies on the basis of the most profitable sets of products. For example, caviar may yield much more profit than bottles of milks in a supermarket. To solve this problem, HUIM (High-utility itemset mining) [8,9] was introduced to discover the set of high-utility itemsets (HUIs). To reduce the search space and maintain the downward closure property, a transaction-weighted utility (TWU) model [10] was developed to find a set of patterns called HTWUIs (High Transaction-Weighted Utilization Itemsets). Discovering HTWUIs can be used to avoid the "combinatorial explosion" problem by maintaining the downward closure property for mining HUIs (High-utility Itemsets). Lin et al. then presented a high-utility pattern (HUP)-tree [11] that keeps only 1-HTWUIs into a condense tree structure based on the TWU model. Liu et al. [12] then developed the HUI-Miner algorithm and presented a utility-list (UL) structure for mining HUIs, which is the most efficient structure to mine HUIs. Lan et al. then presented several works to address the problem of HUIM focusing on the on-shelf problem [13,14]. Zihayat then presented a top-*k* algorithm to efficiently find the top-*k* HUIs from the set of the discovered HUIs in the stream dataset [15]. Several algorithms [16–19] were presented for specific applications and designed to mine various types of HUIs.

Although HUIM can identify the set of profitable itemsets for decision-making, the utility of an itemset often increases when its size is increased. For example, any combination of caviar with another item may be considered as a high-utility itemset (HUI) for market basket analytics, which may mislead a manager/retailer and result in taking wrong or risky decisions. An alternative designed by Hong et al. [20] is HAUIM (High Average-Utility Itemset Mining) to discover the set of HAUIs (High Average-Utility Itemsets) in databases. An Apriori-like approach was first designed that considers the length (size) of each itemset. It calculates the average-utility of each itemset instead of its utility as in HUIM, which provides a flexible way of measuring the importance of itemsets for decision-making. The *auub* (Average-Utility Upper Bound) model [20] was presented to obtain a downward closure property by maintaining the HAUUBIs (High Average-Utility Upper-Bound Itemsets), thus reducing the search space to discover the set of HAUIs. Lin et al. [21] developed an efficient HAUP-tree (High Average-Utility Pattern tree) to store information related to 1-HAUUBIs in a compact tree structure, and avoid candidate generation. An efficient average average-utility (AU)-list structure [22] was then developed to efficiently discover the set of HAUIs. Several algorithms [23,24] were designed and the design of several extensions are in progress.

Most approaches for HAUI mining were developed to find the set of HAUIs in a static database. When the size of a database is increased or decreased (e.g., after inserting new transactions) [25–27], existing algorithms needs to re-mine the updated database in batch mode to update the results. Cheung et al. [25] presented the FUP (Fast UPdated) concept to maintain the discovered frequent itemsets using incremental updates. It categorizes itemsets into four cases and the itemsets of each case are then, respectively, maintained using a designed procedure. The FUP concept was utilized in HUIM [27]. In this paper, we utilize the FUP concept to update the results when transactions are instead based on the *auub* model and AUL-structures. An efficient FUP-HAUIMI algorithm is presented to incrementally update the discovered information. The major contributions are the following.

- A FUP-HAUIMI algorithm is presented to incrementally update the discovered knowledge without candidate generation for transaction insertion. The AUL-structure is utilized in the designed algorithm to efficiently keep information for mining patterns and incrementally updating results.
- The FUP concept is also utilized and modified to provide updating procedures for the FUP-HAUIMI algorithm. Based on the FUP concept, the correctness and completeness of the algorithm to update the discovered knowledge is ensured.
- Experiments show that the designed FUP-HAUIMI algorithm has better performance to maintain and update the discovered HAUIs than that of the state-of-the-art HAUI-Miner algorithm running in batch mode and the state-of-the-art incremental IHAUPM algorithm.

2. Related Work

In this section, work related to incremental pattern discovery and high average-utility itemset mining (HAUIM) is discussed.

2.1. Incremental Pattern Discovery

Data mining is used to find implicit but meaningful knowledge appearing in databases, and it is commonly used in several domains such as market basket analytics, recommendation systems, behavior analysis and prediction systems. The fundamental way to mine relationships between items is called association-rule mining [1,2]. The first algorithm for this task is Apriori [1]. It finds the set of frequent itemsets (FIs) based on a minimum support threshold, and then generates a set of association rules (ARs) in a second stage according to a minimum confidence threshold. Apriori performs a level-wise search. Thus, it can produce numerous candidates at each level to find the actual FIs. To speed up the mining process, Han et al. [2] designed the frequent-pattern (FP)-growth algorithm and developed the FP-tree structure. FP-growth recursively mines the set of FIs without generating candidates. Several algorithms were developed to discover ARs or FIs and it is an active research area [4–7].

In real-life situations, the size of the database may change over time. For example, some transactions may be inserted into a database and some transactions may be removed. To handle this case, the discovered information should be updated. However, traditional algorithms operate in batch mode. Thus, they must be applied again to mine the updated database without reusing results from its previous execution, even if only very few minor updates are performed. To handle this situation for dynamic databases, Cheung et al. designed the FUP (Fast UPdated) concept [25] to maintain information about the discovered frequent itemsets for incremental data mining. It categorizes itemsets based on four cases. Then, the itemsets of each case are updated and maintained by a specific updating procedure. The four FUP cases are illustrated in Figure 1.

In Case 1, an itemset remains a large (frequent) itemset after a database update. Thus, this itemset does not influence the final rules. For itemsets in Case 2, rules are updated since information about the inserted transactions is kept for maintenance. In this case, some rules may become invalid since, when the database is updated, the frequency of each itemset may become smaller than the updated minimum support threshold (count).

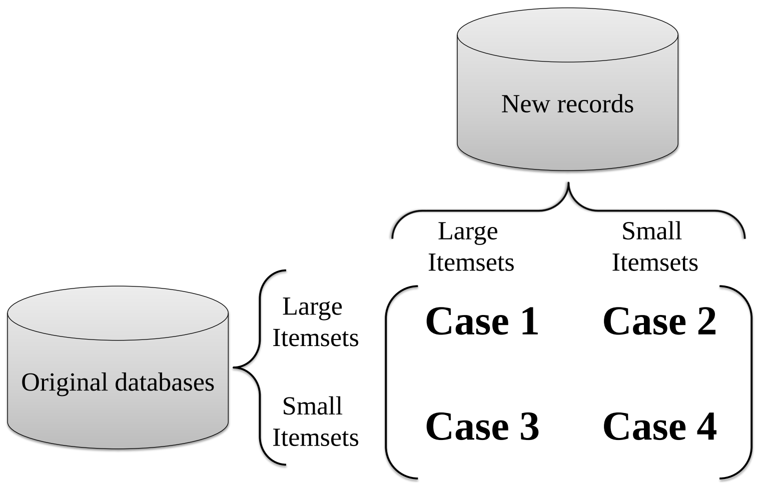


Figure 1. The four cases of the FUP concept.

In Case 4, an itemset remains small (infrequent) after the database update. Thus, it does not affect the final result and itemsets in this case can be ignored. In Case 3, an itemset was not kept when mining the original database. Hence, the original database needs to be re-scanned to obtain the itemset support value for maintenance. In this case, some rules may arise after the database update.

Since the FUP concept applies an Apriori-like approach, which may be inefficient, Hong et al. developed the FUFP-tree (Fast Updated Frequent Pattern tree) algorithm to update frequent itemsets for transaction insertion [26]. It uses an FP-tree-like structure to store information about frequent itemsets in a compressed tree. An FP-growth like approach is applied to discover the updated information without candidate generation. Many extensions of the FUP concept were designed to handle dynamic databases with transaction insertion [27].

2.2. High Average-Utility Itemset Mining

Traditional algorithms for FIM/ARM only consider the frequency of items in binary databases to mine patterns. Other factors such as the importance, interest and weight of items are not considered. Thus, the provided information may be insufficient for a manager/retailer and wrong decisions may be taken based on these patterns. HUIM (High-Utility Itemset Mining) [8,9,13–15] was designed to take into account purchase quantities (internal utility) and unit profits (external utility) of items to reveal profitable itemsets. An itemset is said to be a HUI (High-Utility Itemset) if its utility value is larger than the minimum high-utility threshold (count). To avoid the “combinational exploration” problem and ensure that there is a downward closure property, the HTWUI (High Transaction-Weighted Utilization Itemset) concept was presented, and a property called the TWDC (Transaction-Weighted Downward Closure) property [10] for mining HUIs. Based on the concept of HTWUI, correct and complete algorithms were designed to discover HUIs. Lin et al. designed the HUP-tree (High Utility Pattern tree) algorithm [11] to efficiently discover the set of HUIs based on the developed HUP-tree structure. It keeps only the 1-HTWUIs to explore the search space of itemsets and do not generate candidates. The HUP-tree algorithm is much more efficient than traditional Apriori-like approaches. The UP-Growth+ algorithm and the UP-tree structure were designed by Tseng et al. [28]. They adopt several pruning strategies to reduce the number of unpromising candidates for mining the actual HUIs. Liu et al. [12], respectively, developed the UL-structure (Utility-List) and the HUI-Miner algorithm to efficiently find the set of HUIs. HUI-Miner generate candidates but can still reveal the complete set of HUIs. In addition, it uses a simple join operation to generate *k*-itemsets based on the UL structure and the enumeration tree without candidate generation. Several approaches for HUIM have been developed and it is an active research area [16–19].

In HUIM, the utility of an itemset is the sum of the utilities of its items. This ignores the length of the itemset and that the utility of an itemset often increases with its size. It also sometimes provides redundant information. For example, any combination of items with caviar may be considered as a HUI. This is an unfair way of assessing whether an itemset is a HUI using a single minimum high-utility threshold (count).

High average-utility itemset mining (HAUIM) [20] provides a measure to assess the average-utility of an itemset based on its length. The average-utility of an itemset is its total utility divided by the number of items that it contains. The first Apriori-like algorithm for HAUIM is called TPAU [20]. It generates candidates at each level that it then evaluates. The TPAU algorithm overestimates the average-utility of itemsets using the (*auub*) upper-bound model to obtain a downward closure property, which is used to ensure the completeness and correctness for mining HAUIs (High Average-Utility Itemsets). Lin et al. designed the HAUP-tree (High Average-Utility Pattern tree) [21] to keep information for mining patterns, thus speeding up their discovery. The quantities of prefix items of the current processed item are then kept as an array and attached to the processed item (node). An efficient HAUI-Miner algorithm [22] was also presented, which uses a link-list structure to store information for mining patterns. The HAUI-Miner algorithm is the state-of-the-art algorithm based on the *auub* model to mine HAUIs. An IHAUPM algorithm [29] was presented to incrementally update the discovered HAUIs. It is the state-of-the-art algorithm for incremental maintenance of high average-utility itemsets for transaction insertion. However, the compressed tree structure is inefficient since each node in the tree has to keep an array, which is used to store the quantities of prefix items for the processed node. Several algorithms for HAUIM have been developed and it is an active research area [23,24].

3. Preliminaries and Problem Statement

In this section, an example is given to explain the definitions of the designed approach, which is shown in Table 1. Each transaction consists of several distinct items, and purchase quantities are attached to items. The unit profit table of the original database is shown in Table 2. The user's minimum high average-utility threshold δ is set to 14.4%, which is set by the user based on its preferences and the application domain.

Table 1. An example database.

TID	Items with Their Quantities
T_1	a:4, b:1, c:2, d:1
T_2	b:4, d:3, e:1, g:11
T_3	a:1, c:4, f:3
T_4	c:1, d:2, e:2
T_5	a:2, d:1, f:5

Table 2. Unit profits of items.

Item	a	b	c	d	e	f	g
Profit	3	5	4	8	11	7	2

Definition 1. *The average-utility of an item i_j in the transaction T_q can be denoted as $au(i_j, T_q)$, which is defined as:*

$$au(i_j, T_q) = \frac{q(i_j, T_q) \times p(i_j)}{1}, \tag{1}$$

where $q(i_j, T_q)$ is the quantity of i_j in T_q, and $p(i_j)$ is the unit profit value of i_j.

For instance, the average-utilities of Items (b–d) in T_1 are calculated as $au(b, T_1) = \frac{1 \times 5}{1}(= 5)$, $au(c, T_1) = \frac{2 \times 4}{1}(= 8)$, and $au(d, T_1) = \frac{1 \times 8}{1}(= 8)$, respectively.

Definition 2. *The average-utility of a k-itemset X in a transaction T_q can be denoted as $au(X, T_q)$, which is defined as:*

$$au(X, T_q) = \frac{\sum_{i_j \subseteq X \wedge X \subseteq T_q} q(i_j, T_q) \times p(i_j)}{|X|} = \frac{\sum_{i_j \subseteq X \wedge X \subseteq T_q} q(i_j, T_q) \times p(i_j)}{k}, \quad (2)$$

where k is the number of items in X.

For instance, the average-utility of Itemsets (*ac*) and (*acd*) in T_1 are calculated as $au(ac, T_1) = \frac{4 \times 3 + 2 \times 4}{2}$ (= 10), and $au(acd, T_1) = \frac{4 \times 3 + 2 \times 4 + 1 \times 8}{3}$ (= 9.3), respectively.

Definition 3. *The average-utility of an itemset X in D is denoted as $au(X)$, and defined as:*

$$au(X) = \sum_{X \subseteq T_q \wedge T_q \in D} au(X, T_q). \quad (3)$$

For instance, the average-utility of Itemset (*ac*) in the database is calculated as: $au(ac, T_1) + au(ac, T_3)(= \frac{20}{2} + \frac{19}{2})(= 19.5)$.

Definition 4. *The transaction utility of a transaction T_q is denoted as $tu(T_q)$, and defined as:*

$$tu(T_q) = \sum_{i_j \subseteq T_q} u(i_j, T_q). \quad (4)$$

For instance, $tu(T_1)$(= 12 + 4 + 8 + 8)(= 32), $tu(T_2)$(= 77), $tu(T_3)$(= 44), $tu(T_4)$(= 42), and $tu(T_5)$(= 49).

Definition 5. *The total utility of a database D is denoted as TU^D, which is the sum of all transaction utilities as:*

$$TU^D = \sum_{T_q \in D} tu(T_q). \quad (5)$$

For instance, the total utility of Table 1 is calculated as TU^D (= 32 + 77 + 44 + 42 + 49) (= 244).

To obtain a downward closure property for pruning unpromising candidates early, the average-utility upper bound (*auub*) model [20] was designed. It overestimates the utility of itemsets. This process ensures the correctness and completeness of the process for mining itemsets. The definitions are as follows.

Definition 6 (Transaction-maximum utility, tmu)**.** *The transaction-maximum utility of a transaction T_q is denoted as $tmu(T_q)$ and defined as:*

$$tmu(T_q) = max\{u(i_j) | i_j \subseteq X \wedge X \subseteq T_q\}. \quad (6)$$

For instance, the *tmu* of transactions T_1 to T_5 are calculated as $tmu(T_1)$(= 12), $tmu(T_2)$(= 24), $tmu(T_3)$(= 21), $tmu(T_4)$(= 22), and $tmu(T_5)$(= 35), respectively.

Definition 7 (Average-utility upper-bound, auub)**.** *The average-utility upper-bound (auub) of an itemset X in the original database is denoted as $auub(X)^D$ and defined as:*

$$auub(X)^D = \sum_{X \subseteq T_q \wedge T_q \in D} tmu(T_q), \quad (7)$$

where $tmu(T_q)$ is the maximum utility of transaction T_q such that $i_j \subseteq X \wedge X \subseteq T_q$.

For instance, the *auub* values of the itemsets (a) and (ac) are calculated as: $auub(a)^D$(= 12 + 21 + 35)(= 68) and $auub(ac)^D$(= 12 + 21)(= 33), respectively.

Property 1 (Downward closure of *auub*)**.** *Let an itemset Y be the superset of an itemset X such that $X \subseteq Y$. According to the downward closure property of auub, we obtain that:*

$$auub(Y)^D \leq auub(X)^D. \tag{8}$$

Thus, if $auub(X)^D \leq TU^D \times \delta$, then $auub(Y)^D \leq auub(X)^D \leq TU^D \times \delta$ holds for any superset of an itemset X.

Definition 8 (High average-utility upper bound itemset, HAUUBI)**.** *An itemset X is a high average-utility upper bound itemset (HAUUBI) in the original database if it satisfies that condition:*

$$HAUUBI^D \leftarrow \{X | auub(X)^D \geq TU^D \times \delta\} \tag{9}$$

For instance, assume that the minimum average-utility threshold is set to 14.4%. Thus, the itemset (a) is a high average utility upper bound itemset (HAUUBI) since $auub(a)^D$ (= $68 > 244 \times$ 14.4%)(= 35.13). However, the itemset (ab) is not a HAUUBI since $auub(ab)$ (= $12 < 35.13$).

For the incremental mining problem, we assume that several transactions are inserted into the original database. In this example, transactions for insertion are shown in Table 3. Thus, the problem definition of this paper is defined as follows.

Table 3. A new database of inserted transactions.

TID	Items with Their Quantities
T_6	a:3, b:5, c:3, d:1
T_7	a:4, c:2, e:1, f:1, g:5

Problem Statement: For incremental HAUIM, it is necessary to design an efficient algorithm for transaction insertion to maintain and update the discovered information, and multiple scans of the updated database should be avoided. Thus, an itemset is considered as a HAUI if it satisfies the following condition:

$$HAUI \leftarrow \{X | au(X)^U \geq (TU^D + TU^d) \times \delta\}, \tag{10}$$

where $au(X)^U$ is the updated average-utility of X, TU^D is the total utility in the original database D, and TU^d is the total utility in the modified database d. Moreover, δ is a user specified minimum average-utility threshold, which can be adjusted by experts for different domains.

4. Proposed FUP-HAUIMI Algorithm with Transaction Insertion

In the past, the HUI-Miner algorithm was used to efficiently mine HUIs using an efficient UL-structure (Utility-List structure). Lin et al. then utilized the UL-structure and developed an efficient AUL-structure (Average-Utility-List structure) and designed the HAUI-Miner algorithm [22] for mining HAUIs. The HAUI-Miner is based on the well-known *auub* model to ensure the correctness and completeness for discovering patterns, and is the state-of-the-art batch algorithm. However, in real-life, databases are often updated by inserting or deleting transactions. Previous HAUIM algorithms must re-scan an updated database in batch mode to update the set of HAUIs. This process wastes computer resources. In this paper, we thus develop an incremental algorithm called FUP-HAUIMI for maintaining a database for the case of transaction insertion. The AUL-structure and the FUP concept are utilized to speed up the mining performance, and ensure the correctness and completeness for discovering HAUIs. A simple join operation is used in this paper to generate patterns for exploring the enumeration tree. Details of the developed FUP-HAUIMI algorithm are explained below.

4.1. The Average-Utility-List (AUL)-Structure

The AUL-structure is utilized by the HAUI-Miner algorithm [22]. Each itemset X in the AUL-structure structure keeps three fields such as the transaction ID represented as (*tid*), the utility of an item X in a transaction T_q denoted as (*iu*), and the transaction-maximum-utility of an item X in transaction T_q denoted as (*tmu*). For generating k-itemsets ($k \geq 2$), an enumeration tree representing the search space is explored using a depth-first search to determine whether the superset (k + 1)-itemsets of k-itemsets should be generated. If it satisfies the condition, a simple join operation is recursively performed to generate the AUL-structures of k-itemsets. This process is then recursively performed until no candidates are generated. After that, a database scan is done to find the actual HAUIs. The enumeration tree for the example database is shown in Figure 2. Based on the AUL-structures, we can obtain the following properties.

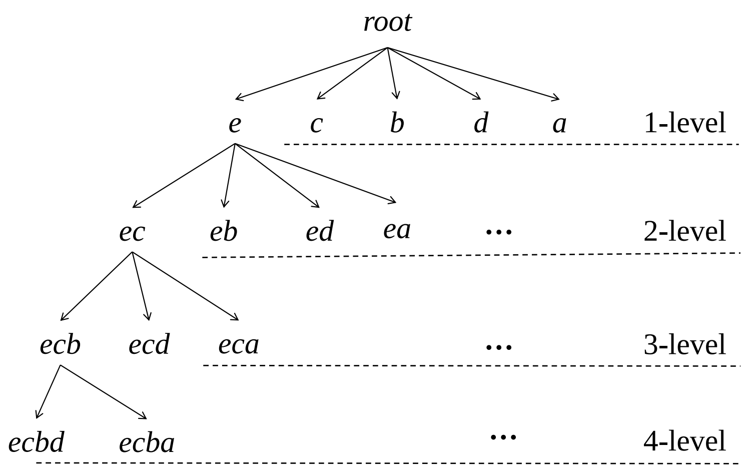


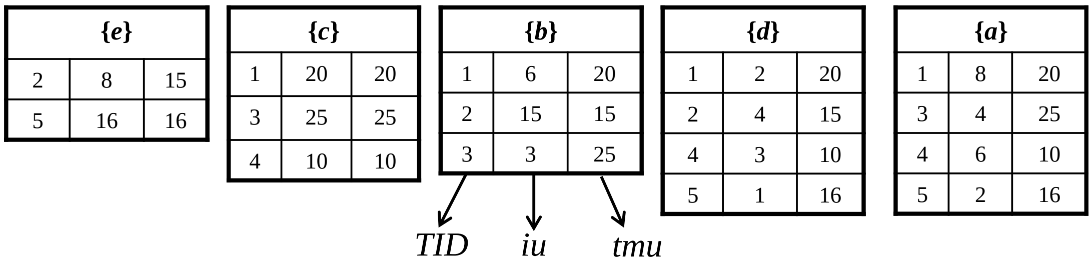
Figure 2. The enumeration tree used in the example.

Property 2. *All the 1-HAUUIs of the original database are sorted by auub-ascending order, which is also used as the construction order for the AUL-structures.*

Property 3. *To ensure the correctness and completeness for mining HAUUBIs and HAUIs, the AUL-structures of all items in the inserted transactions should be constructed for later maintenance. This is reasonable since the size of the inserted transactions is much smaller than that of the original database in real-life situations.*

Property 4. *The auub-ascending order of 1-HAUUBIs in the updated database is almost the same as the sorted order in the original database since the size of the inserted transactions is smaller than the size of the original database. Thus, this sorting order of itemsets generally does not change much when the database is updated.*

Since all 1-items of inserted transactions are kept to build the initial AUL-structures, the designed FUP-HAUIMI algorithm can efficiently find the updated HAUUBIs and HAUIs without generating candidates and is complete and correct to update HAUUBIs and HAUIs. The AUL-structures of 1-HAUUBIs obtained from the original database are shown in Figure 3.



{e}		
2	8	15
5	16	16

{c}		
1	20	20
3	25	25
4	10	10

{b}		
1	6	20
2	15	15
3	3	25

{d}		
1	2	20
2	4	15
4	3	10
5	1	16

{a}		
1	8	20
3	4	25
4	6	10
5	2	16

Figure 3. The constructed AUL-structures of the original database.

4.2. The Adapted Fast Updated Concept

The designed FUP-HAUIMI algorithm categorizes 1-HAUUBIs into four cases according to the FUP concept, which is extended from [25]. The 1-HAUUBIs of each case are then, respectively, processed by the designed procedures to update the 1-HAUUBIs. The four cases of the adapted FUP concept are shown in Figure 4. Thus, we can obtain the following information as:

- **Case 1:** An itemset is a 1-HAUUBI both in the original database and in the inserted transactions. In this case, this itemset remains a 1-HAUUBI after the database is updated.
- **Case 2:** An itemset is a 1-HAUUBI in the original database but not a 1-HAUUBI in the inserted transactions. In this case, the itemset is a 1-HAUUBI or not a 1-HAUUBI after the database is updated.
- **Case 3:** An itemset is a non 1-HAUUBI in the original database but a 1-HAUUBI in the inserted transactions. In this case, an additional database scan is needed to obtain the average-utility of the itemset in the original database, and then update its average-utility after the database is updated.
- **Case 4:** An itemset is a non 1-HAUUBI both in the original database and in the inserted transactions. In this case, an itemset remains a non 1-HAUUBI after the database is updated.

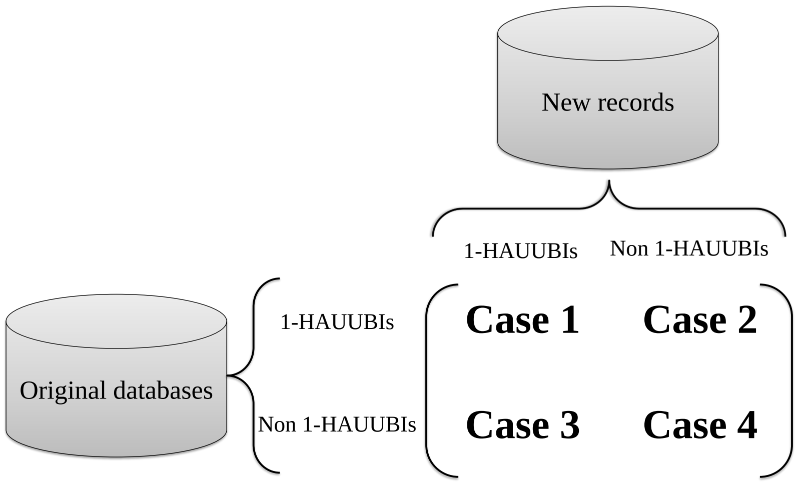


Figure 4. Four cases of the adapted FUP concept.

Based on the above cases, lemmas, proofs, and corollaries are provided as follows to ensure the correctness and completeness for discovering 1-HAUUBIs for incremental HAUIM.

Lemma 1. *An itemset X is a 1-HAUUBI in the original Database (D) and inserted transactions (d). Thus, it remains a 1-HAUUBI in the updated Database (U).*

Proof. Since X is a 1-HAUUBI both in (D) and (d), its $auub(X)^D \geq TU^D \times \delta$ and $auub(X)^d \geq TU^d \times \delta$. Thus, the updated result is: $auub(X)^U$ = $auub(X)^D$ + $auub(X)^d$ = $TU^D \times \delta + TU^d \times \delta$ = t$(TU^D + TU^d) \times \delta$. Hence, we conclude that the itemset remains a 1-HAUUBI in the updated Database (U). □

Corollary 1. *An itemset X remains a 1-HAUUBI in (U) if it belongs to Case 1.*

Lemma 2. *An itemset X is a 1-HAUUBI in (D) but a non 1-HAUUBI in (d). There are two possible situations: (1) it is a 1-HAUUBI in (U); or (2) it becomes a non 1-HAUUBI in (U).*

Proof. Since an itemset X is a 1-HAUUBI in (D), it follows that $auub(X)^D \geq TU^D \times \delta$. It is a non 1-HAUUBI in (d) since $auub(X)^d < TU^d \times \delta$. There are two possible situations: (1) $auub(X)^U = auub(X)^D + auub(X)^d \geq TU^D \times \delta + TU^d \times \delta$ = $(TU^D + TU^d) \times \delta$; or (2) $auub(X)^U < (TU^D + TU^d) \times \delta$. □

Corollary 2. *Two situations may occur for the itemset X: (1) it remains a 1-HAUUBI in (U); or (2) it is a non 1-HAUUBI in (U) if it belongs to Case 2.*

Lemma 3. *If an itemset X is a non 1-HAUUBI in (D) but is a 1-HAUUBI in (d), an additional database scan is needed to obtain the $auub(X)^D$. There are two possible situations: (1) it remains a non 1-HAUBBI in (U); or (2) it becomes a 1-HAUUBI in (U).*

Proof. An itemset X is a non 1-HAUUBI in (D) such as $auub(X)^D < TU^D \times \delta$, and is a 1-HAUUBI in (d) such as $auub(X)^d \geq TU^d \times \delta$. Thus, we can have that: (1) $auub(X)^U$ = $auub(X)^D + auub(X)^d \geq TU^D \times \delta + TU^d \times \delta$ $(= TU^D + TU^d) \times \delta$; or (2) $auub(X)^U < (= TU^D + TU^d) \times \delta$. □

Corollary 3. *For an itemset X, two situations may occur: (1) it remains a non 1-HAUUBI in (U); or (2) it becomes a 1-HAUUBI in (U) if it appears in Case 3.*

Lemma 4. *If an itemset X is a non 1-HAUUBI in both (D) and (d), it remains a non 1-HAUUBI in (U).*

Proof. An itemset X is a non 1-HAUUBI both in (D) and (d) such that $auub(X)^D < TU^D \times \delta$, and $auub(X)^d < TU^d \times \delta$. Thus, we can obtain that $auub(X)^U$ = $auub(X)^D + auub(X)^d$< $(TU^D + TU^d) \times \delta$. □

Corollary 4. *An itemset X remains a non 1-HAUUBI in (U) if it appears in Case 4, which can be directly ignored since it would not affect the final rules.*

Based on the above lemmas, proofs, and corollaries, the downward closure property of the *auub* model can still hold for incremental mining. Thus, the discovery of the updated 1-HAUUBIs is correct and complete. The FUP-HAUIMI algorithm initially constructs the AUL-structures of 1-HAUUBIs based on the original database. The AUL-structures of all items in the inserted transactions are also obtained, which can ensure the completeness and correctness of the designed maintenance approaches. When some transactions are inserted into the original database, the designed FUP-HAUIMI algorithm first divides the HAUUBIs into four cases, and the itemsets of each case are then, respectively, maintained and updated by the designed procedures. After that, the *k*-HAUUBIs are then checked to find their actual *k*-HAUIs. The pseudocode of the developed FUP-HAUIMI algorithm is provided in Algorithm 1.

Algorithm 1: Proposed FUP-HAUIMI algorithm

```
Input: $D$, the original database; $utable$, a unit profit table; $d$, inserted transactions; $\delta$, minimum
       average-utility threshold; $D.AULs$, the AUL-structures of $D$.
Output: the set of high average-utility itemsets.
1  for each $T_q \in d$ do
2      for each $X \subseteq T_q$ do
3          build $X.AUL$;
4      $d.AULs \leftarrow \cup X.AUL$;
5  Merge($D.AULs, d.AULs$);
6  for each $X \in U.AULs$ do
7      if $\frac{X.iu.sum}{|X|} \geq (TU^d + TU^D) \times \delta$ then
8          $HAUIs \leftarrow HAUIs \cup X$;
9      if $\frac{X.tmu.sum}{|X|} \geq (TU^d + TU^D) \times \delta$ then
10         $exAULs \leftarrow null$;
11         for each $Y$ after $X$ in $U.AUL$ do
12             $exAULs \leftarrow exAULs$ + Construct($X.AUL, Y$);
13         Merge($X, exAUL$);
14 for $X \in U.AULs$ do
15     if $\frac{X.iu.sum}{|X|} \geq (TU^d + TU^D) \times \delta$ then
16         $HAUIs \leftarrow \cup X$;
17 return $HAUIs$, $U.AULs$.
```

The designed Algorithm 1 first scans the database by considering items in the newly inserted transactions to construct their AUL-structures (Lines 1–4 of Algorithm 1). After that, the AUL-structures built from the original database and inserted transactions are merged (Line 5 of Algorithm 1). Based on the merged AUL-structures, if the average-utility of an itemset is no less than the updated minimum average-utility count, it is considered as a high average-utility itemset (Lines 7–8 of Algorithm 1), and then its supersets are considered using a depth-first search according to the enumeration tree. This process is then recursively repeated until no tree nodes are generated (Lines 9–13 of Algorithm 1). In the **Merge** function, if an item does not exist in the original database (Line 4 of Algorithm 2), the original database is then rescanned to construct the AUL-structure of the item. The average-utility of the selected itemsets is then calculated (Lines 14–16 of Algorithm 1). After that, the algorithm terminates. The updated patterns have been obtained.

Algorithm 2: Merge ($D.AULs, d.AULs$)

```
1  set $X.AUL \leftarrow null$;
2  set $U.AULs \leftarrow null$;
3  for each $X \in d.AULs$ do
4      if $\frac{X.iu.sum}{|X|} \geq TU^d \times \delta \wedge X \notin D.AUL$ then
5          scan $D$ to obtain $X.AUL$ from $D$;
6      if $\exists X \in D.AULs \wedge X \in d.AULs$ then
7          for each element $E_j \in X.AUL$ do
8              $X.iu.sum \leftarrow X.iu.sum + E_j.iu$;
9              update $X.AUL.tmu$;
10             $X.AUL \leftarrow E_j$;
11         $U.AULs \leftarrow \cup X.AUL$;
```

The **Construct** function shown in Algorithm 3 is used to determine whether the supersets of an itemset must be maintained using the set of *tids* to generate combinations (Lines 1–3 of Algorithm 3). The information about utilities (Lines 4–6 of Algorithm 3) are updated, and the updated AUL-structures are then returned (Lines 7–8 in Algorithm 3).

Algorithm 3: Construct $(X.AUL, Y)$

```
Input: $X.AUL$, the AUL-structures of $X$; $Y$, the itemset Y after $X$ in $X.AUL$.
Output: $XY.AUL$, the AUL-structures of $XY$.
1 $XY.AUL \leftarrow null;$
2 if $\exists E \in Y.AUL \wedge X.AUL.tid == Y.AUL.tid$ then
3     $E_{XY}.AUL.tid \leftarrow X.AUL.tid;$
4     $E_{XY}.iu \leftarrow (X.AUL + Y.AUL)/2;$
5     update $E_{XY}.tmu;$
6     $E_{XY} \leftarrow < E_{XY}.tid, E_{XY}.iu, E_{XY}.tmu >;$
7     $XY.AUL \leftarrow \cup E_{XY}.$
8 return $XY.AUL;$
```

5. Experimental Evaluation

In this section, an evaluation of the runtime, memory usage, number of patterns and scalability of the proposed algorithm is presented. The designed FUP-HAUIMI algorithm is compared with the state-of-the-art HAUI-Miner algorithm running in batch mode [22], and the state-of-the-art IHAUPM algorithm [29] running in incremental mode. All the algorithms were written in Java, and executed on an Intel(R) Core(TM) i7-6700 4.00 GHz processor with 8 GB of main memory. Windows 10 was used as operating system for running the experiments. Six datasets were used, including four real-life datasets [30] and two synthetic datasets. The simulated datasets were generated by the IBM Quest Synthetic data generator [31]. A two-phase simulation model [10] is used to generate the internal and external utilities. To describe the datasets, $\#|\mathbf{D}|$ represents the number of transactions in the dataset; $\#|\mathbf{I}|$ is the number of items in the dataset; **AveLen** indicates the average-length of the transactions in the dataset; **MaxLen** denotes the maximal length among all transactions in the dataset; and **Type** indicates whether the dataset is sparse or dense. Characteristics of the datasets are shown in Table 4.

Table 4. Characteristics of the datasets.

Dataset	#|D|	#|I|	AvgLen	MaxLen	Type
retail	88,162	16,407	10	76	Sparse
T10I4D100K	100,000	870	10.1	29	Sparse
kosarak	990,002	41,270	8	2498	Sparse
T40I10D100K	100,000	1000	39.6	77	Dense
mushroom	963	119	23	23	Dense
foodmart	21,556	1559	4	11	Sparse

Experiments were performed using various minimum high average-utility thresholds (TH) values and different insertion ratios (IR). The IR is the percentage of transactions in the original dataset that is inserted for incremental maintenance. Results are presented next.

5.1. Runtime

In the experiments, the runtime of three algorithms was measured for various TH values and a fixed IR (=1%), as shown in Figure 5. Since the characteristics of the datasets used in the experiments are varied, different TH values are used for different datasets.

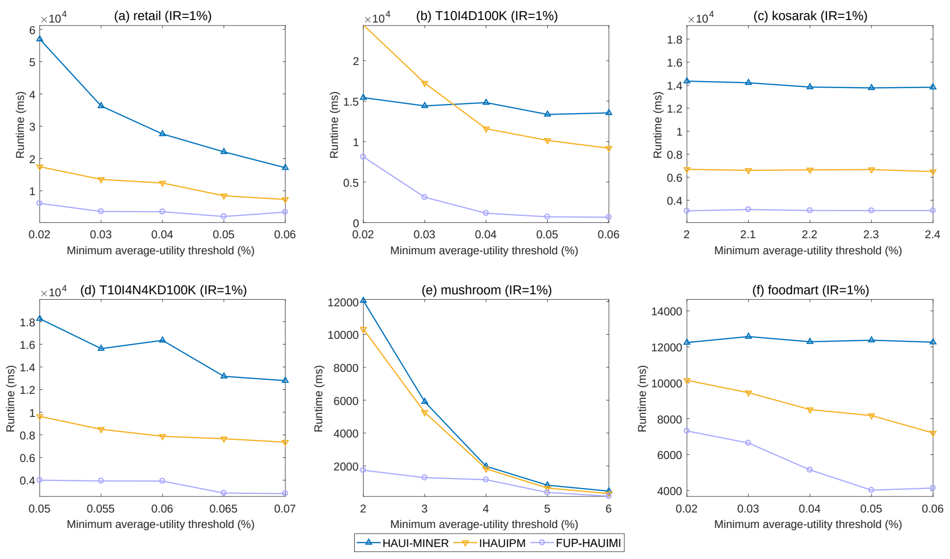


Figure 5. Runtimes for various threshold values.

In Figure 5, it is obvious that the designed FUP-HAUIMI algorithm has better performance than the other two algorithms for the six datasets. As the TH value is increased, the runtimes of the three algorithms decrease. This is reasonable since when the TH is increased, fewer HAUIs are discovered. Hence, less runtime is required for the three algorithms. Besides, it can be seen that the designed FUP-HAUIMI algorithm remains stable for various TH values for some datasets such as Figure 5a,c,d. The reason is that, although the TH is increased, most of the knowledge was discovered for lower minimum average-utility threshold values. It is also worth to mention that HAUI-Miner is the state-of-the-art algorithm for mining HAUIs based on the *auub* model, and that IHAUIM is the state-of-the-art algorithm for incremental HAUIM using a tree structure. Thus, it can be concluded that the designed FUP-HAUIMI algorithm performs well for handling a dynamic database with transaction insertion. This is because the designed FUP-HAUIMI algorithm only handles small parts of the inserted transactions, and some itemsets in Case 3 can be ignored. Hence, the runtime is reduced. Thanks to the efficiency of the AUL-strcuture, it is easy to calculate and obtain the required HAUIs. Experiments are also conducted for various IR values with a fixed TH value for the six datasets. Results are shown in Figure 6.

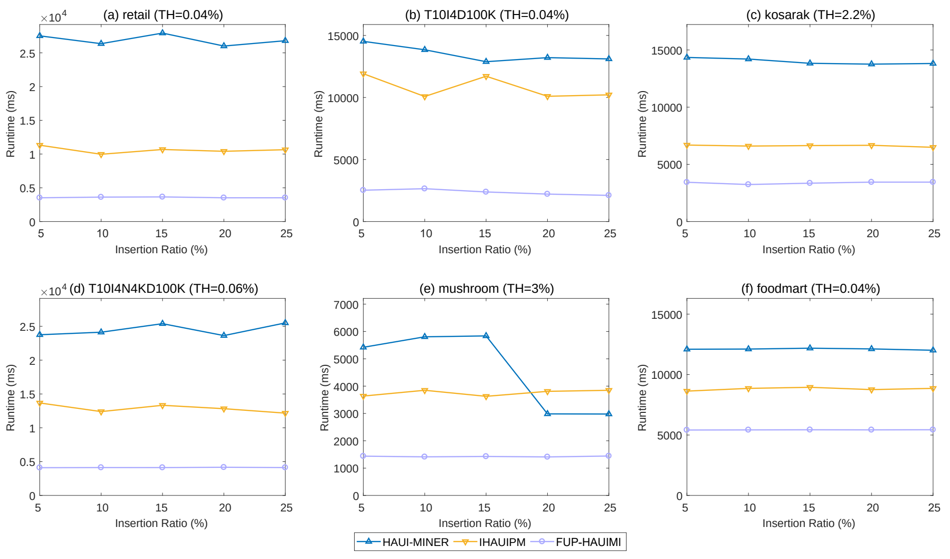


Figure 6. Runtimes for various insertion ratios.

As shown in Figure 6, it can be seen that the proposed FUP-HAUIMI algorithm still outperforms the HAUI-Miner and the IHAUPM algorithms. As the IR is increased, it is obvious that all the algorithms remains stable, and especially the designed FUP-HAUIMI algorithm. The reason is that inserted transactions are extracted from the original database according to the percentage of IR, the items in the inserted transactions are sometimes similar to the original one, most knowledge was already discovered, and only fewer itemsets are updated. However, the proposed FUP-HAUIMI still reduces the runtime and outperforms the other algorithms for all datasets. Generally, the designed FUP-HAUIMI helps to reduce the mining cost since the discovered knowledge can be efficiently handled and maintained. In addition, the multiple database scans can be successfully voided in most cases.

5.2. Memory Usage

In this section, experiments in terms of memory usage are conducted for various TH values and a fixed IR value. Results are shown in Figure 7.

In Figure 7, it can be seen that the HAUI-Miner algorithm outperforms others in terms of memory usage. The reason is that HAUI-Miner uses the utility-list structure to maintain the discovered information, which is a compressed and efficient data structure. Thus, it generally needs less memory than the IHAUPM algorithm since IHAUPM employs a tree structure for incremental maintenance. Besides, the HAUI-Miner algorithm does not need to keep extra information for maintenance. When the size of the database is changed, the HAUI-Miner algorithm re-scans the database to obtain the updated information, which is of course costly but requires less memory. For incremental maintenance, the IHAUPM algorithm generally needs more memory than the developed FUP-IHAUIMI algorithm except for Figure 7c. This is reasonable since IHAUPM uses the HAUP-tree structure to maintain information, which requires more memory than the AUL-structures of the FUP-HAUIMI algorithm. Moreover, the kosarak dataset is a very sparse dataset with a very long transactions. In this case, the developed FUP-HAUIMI may need to generate more AUL-structures to keep the necessary information for the later mining progress. The results for various IR values and a fixed TH are shown in Figure 8.

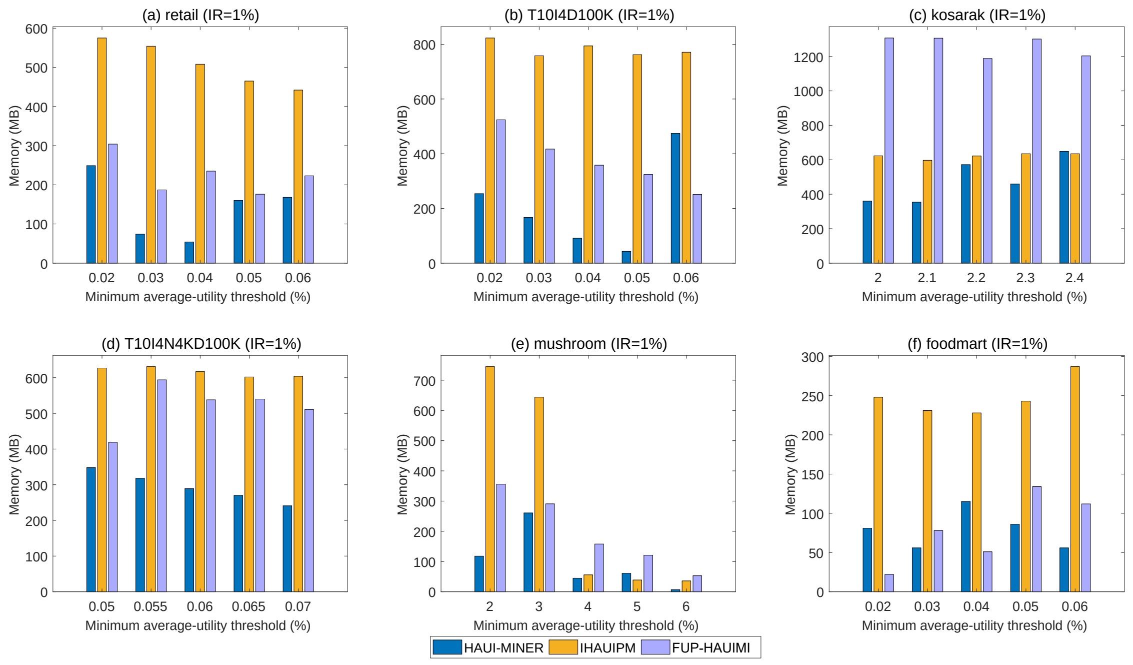


Figure 7. The results of memory usage w.r.t varied thresholds.

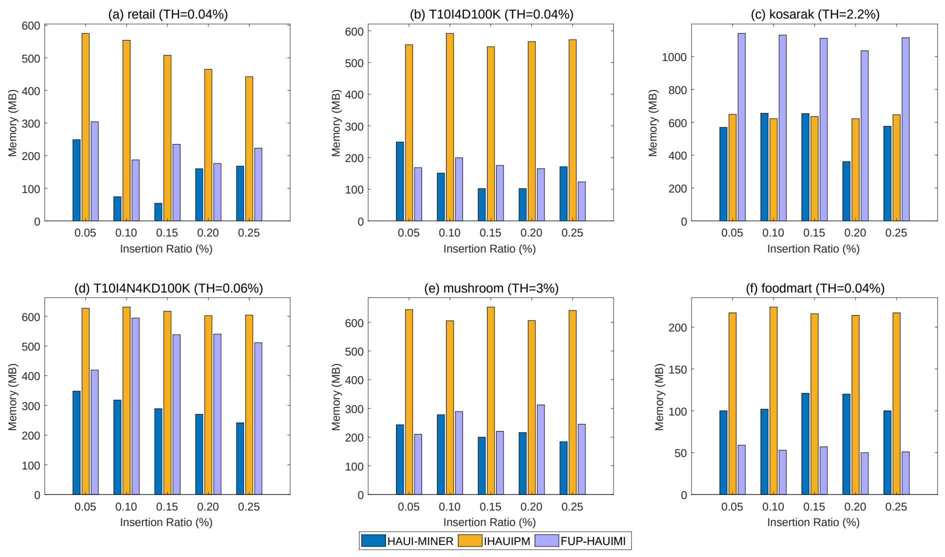


Figure 8. Memory usage for various insertion ratios.

In Figure 8, similar results to those in Figure 7 are observed. HAUI-Miner requires less memory usage, and IHAUPM requires more memory than the developed FUP-HAUIMI algorithm except for the kosarak dataset. Thus, we can conclude that the AUL-structure may be not useful to reduce the memory usage while performing the incremental data mining for very sparse datasets with very long transactions. Generally, the designed FUP-HAUIMI outperforms IHAUPM for incremental mining of high average-utility itemsets, especially the AUL-structures can successfully keep the necessary information than that of the tree-based structures.

5.3. Number of Patterns

In this section, experiments are presented to assess the number of candidate patterns produced for discovering the actual HAUIs. The results for various TH values and a fixed IR are shown in Figure 9.

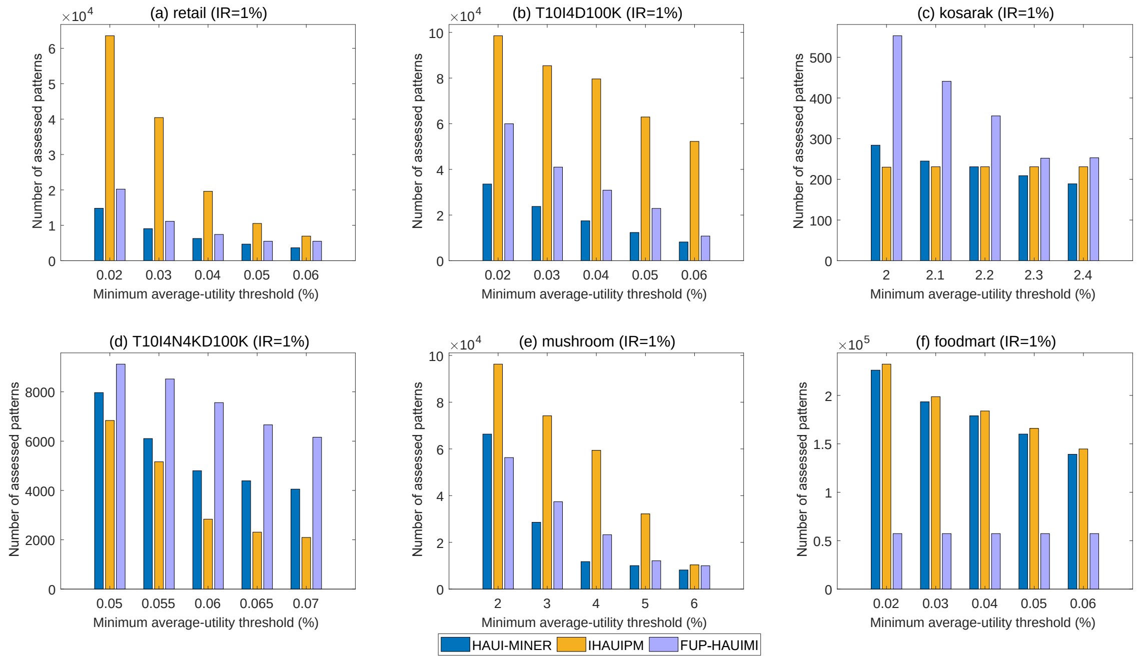


Figure 9. Number of candidate patterns for various threshold values.

In Figure 9, we can see that the candidate patterns considered by the proposed FUP-HAUIMI algorithm are much fewer than that of the HAUI-Miner and IHAUPM algorithms except in Figure 9d. The reason is that the T40I10D100K dataset is dense. Thus, many transactions have the same items for maintenance. Thus, the proposed FUP-HAUIMI algorithm may need to check more patterns in the enumeration tree to check if the supersets must be generated. However, in general cases, the FUP-HAUIMI algorithm still checks less patterns than other algorithms. This shows that the AUL-structures and the adapted FUP concept can successfully reduce the cost of incremental mining of average-utility itemsets. Results for various DR values and a fixed TH are shown in Figure 10.

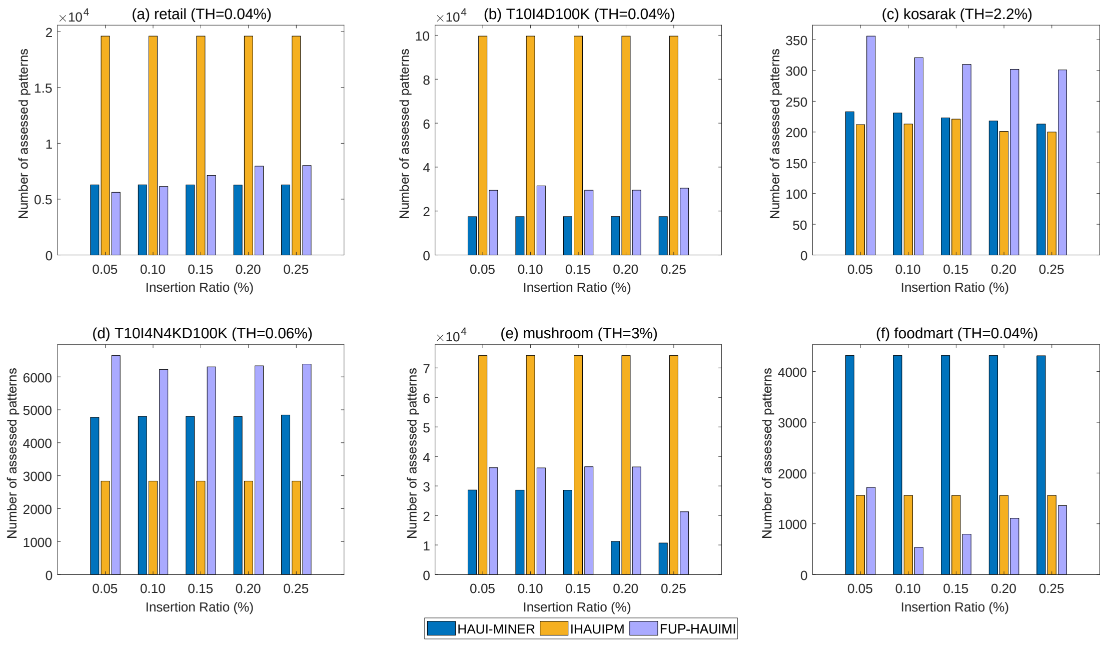


Figure 10. Number of candidate patterns for various insertion ratios.

Again, we can observe that the FUP-HAUIMI algorithm needs to examine more candidates in very sparse and dense datasets such as the datasets of Figure 10c,d. However, for the other datasets such as the datasets of Figure 10a,b,e, the proposed FUP-HAUIMI algorithm outperforms the IHAUPM algorithm, and even has the best results in Figure 10f. However, in terms of runtime performance, the proposed FUP-HAUIMI algorithm outperforms the other algorithms. Thanks to the efficient FUP concept and the AUL-structures, the runtime of the FUP-HAUIMI can be greatly reduced. From the above results, we can thus conclude that although extra memory usage is required and more candidate patterns are checked by the designed FUP-HAUIMI algorithm, this latter can still achieve better efficiency and effectiveness in most cases except for a very sparse dataset with long transactions or for a very dense dataset.

5.4. Scalability

In this section, the scalability of three algorithms were compared on a synthetic dataset, which is generated by the IBM Generator [31]. The size of the database (scalability) is varied from 100 K to 500 K transactions, with a 100 K increment. The results are shown in Figure 11.

It can be observed in Figure 11 that the proposed FUP-HAUIMI algorithm has better scalability than HAUI-Miner and IHAUPM. As $|X|$ is increased, the runtime of the HAUI-Miner increases since the database needs to be scanned multiple times. However, the runtime of FUP-HAUIMI only slightly increases. In addition, we can see that for these datasets, the designed FUP-HAUIMI algorithm performs better than the other two algorithms in terms of runtime, memory usage, and the number of assessed patterns. Thanks to the AUL-structures, the memory usage can be greatly reduced, which can be seen in Figure 11b. In addition, the FUP concept achieves good efficiency since it can reduce the number of assessed patterns, as it can be seen in Figure 11c. Based on the adapted FUP concept, the proposed FUP-HAUIMI algorithm can evaluate fewer patterns and the correctness and completeness of the algorithm is ensured for mining the updated HAUIs. Generally, the proposed FUP-HAUIMI algorithm has better performance than that of the state-of-the-art batch mode HAUI-Miner algorithm and the incremental IHAUPM algorithm.

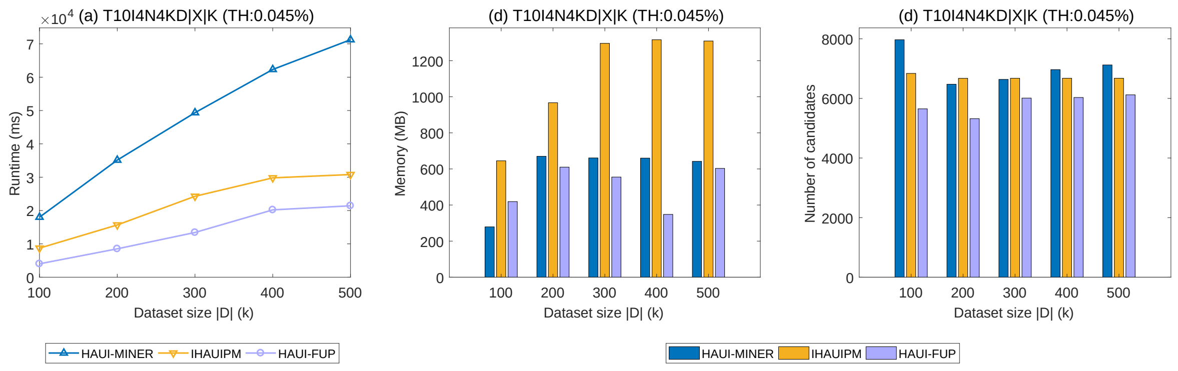


Figure 11. Scalability for various database size.

6. Conclusions and Future Work

High average-utility itemset mining (HAUIM) is an emerging topic, which is an extension of traditional high-utility itemset mining (HUIM) but takes the size of itemsets into account to evaluate the average-utility of each itemset. Most HAUIM algorithms only perform in batch mode. When the size of the database is changed, the updated database must be scanned again to obtain up-to-date information. This is a costly approach since previously discovered information and the runtime to find that information is wasted. In this paper, we utilize an average-utility-list (AUL)-structure to keep the necessary information for the updating process. In addition, the adapted FUP concept is useful to reduce the number of candidate patterns for incremental mining. Thus, the developed FUP-HAUIMI algorithm can achieve better performance than that of the state-of-the-art batch-mode HAUI-Miner and the incremental IHAUPM algorithms in terms of runtime, memory usage, number of candidate patterns and scalability.

In addition to transaction insertion, transaction deletion or transaction modification for HAUIM are also the emerging topics in dynamic data mining. How to efficiently maintain the discovered knowledge in the dynamic databases is also a critical issue and will be considered as our future works.

Author Contributions: B.Z. and J.C.-W.L. wrote the main concepts of the manuscript; Y.S. and Y.D. designed and implemented the experiments; and P.F.-V. checked the English writing and organization of the manuscript.

Acknowledgments: This research was partially supported by the Shenzhen Technical Project under JCYJ20170307151733005 and KQJSCX20170726103424709

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

References

1. Agrawal, R.; Srikant, R. Fast algorithms for mining association rules in large databases. In Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile, 12–15 September 1994; pp. 487–499.
2. 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]
3. Deng, Z.H.; Lv, S.L. Fast mining frequent itemsets using nodesets. *Expert Syst. Appl.* **2014**, *41*, 4505–4512. [CrossRef]
4. Chen, M.S.; Park, J.S.; Yu, P.S. Efficient data mining for path traversal patterns. *IEEE Trans. Knowl. Data Eng.* **1998**, *10*, 209–221. [CrossRef]

5. Creighton, C.; Hanash, S. Mining gene expression databases for association rules. *Bioinformatics* **2003**, *19*, 79–86. [CrossRef] [PubMed]
6. Lucchese, C.; Orlando, S.; Perego, R. Fast and memory efficient mining of frequent closed itemsets. *IEEE Trans. Knowl. Data Eng.* **2006**, *18*, 21–36. [CrossRef]
7. Yen, S.J.; Lee, Y.S. Mining high utility quantitative association rules. In Proceedings of the International Conference on Data Warehousing and Knowledge Discovery, Regensburg, Germany, 3–7 September 2007; pp. 283–292.
8. Liu, Y.; Liao, W.K.; Choudhary, A. A fast high utility itemsets mining algorithm. In Proceedings of the International Workshop on Utility-Based Data Mining, Chicago, IL, USA, 21–21 August 2005; pp. 90–99.
9. Yao, H.; Hamilton, H.J.; Butz, C.J. A foundational approach to mining itemset utilities from databases. In Proceedings of the SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, 22–24 April 2004; pp. 215–221.
10. Liu, Y.; Liao, W.K.; Choudhary, A. A two-phase algorithm for fast discovery of high utility itemsets. In Proceedings of the Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, Hanoi, Vietnam, 18–20 May 2005; pp. 689–695.
11. Lin, C.W.; Hong, T.P.; Lu, W.H. An effective tree structure for mining high utility itemsets. *Expert Syst. Appl.* **2011**, *38*, 7419–7424. [CrossRef]
12. Liu, M.; Qu, J. Mining high utility itemsets without candidate generation. In Proceedings of the ACM International Conference on Information and Knowledge Management, Maui, Hawaii, USA, 29 October–2 Novermber 2012; pp. 55–64.
13. Lan, G.C.; Hong, T.P.; Tseng, V.S. Discovery of high utility itemsets from on-shelf time periods of products. *Expert Syst. Appl.* **2011**, *38*, 5851–5857. [CrossRef]
14. Lan, G.C.; Hong, T.P.; Huang, J.P.; Tseng, V.S. On-shelf utility mining with negative item values. *Expert Syst. Appl.* **2014**, *41*, 3450–3459. [CrossRef]
15. Zihayat, M. Mining top-*k* high utility patterns over data streams. *Inf. Sci.* **2014**, *285*, 138–161. [CrossRef]
16. Ahmed, C.F.; Tanbeer, S.K.; Jeong, B.S.; Lee, Y.K. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Trans. Knowl. Data Eng.* **2009**, *21*, 1708–1721. [CrossRef]
17. Erwin, A.; Gopalan, R.P.; Achuthan, N.R. Efficient mining of high utility itemsets from large datasets. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, Osaka, Japan, 20–23 May 2008; pp. 554–561.
18. Liu, J.; Wang, K.; Fung, B.C.M. Direct discovery of high utility itemsets without candidate generation. In Proceedings of the IEEE International Conference on Data Mining, Brussels, Belgium, 10–13 December 2012; pp. 984–989.
19. Liu, J.; Wang, K.; Fung, B.C.M. Mining high utility patterns in one phase without generating candidates. *IEEE Trans. Knowl. Data Eng.* **2016**, *28*, 1245–1257. [CrossRef]
20. Hong, T.P.; Lee, C.H.; Wang, S.L. Effective utility mining with the measure of average utility. *Expert Syst. Appl.* **2011**, *38*, 8259–8265. [CrossRef]
21. Lin, C.W.; Hong, T.P.; Lu, W.H. Efficiently mining high average utility itemsets with a tree structure. In Proceedings of the Asian Conference on Intelligent Information and Database Systems, Hue, Vietnam, 24–26 March 2010; pp. 131–139.
22. Lin, C.W.; Li, T.; Fournier-Viger, P.; Hong, T.P.; Zhan, J.; Voznak, M. An efficient algorithm to mine high average-utility itemsets. *Adv. Eng. Inf.* **2016**, *30*, 233–243. [CrossRef]
23. Lan, G.C.; Hong, T.P.; Tseng, V.S. Efficient mining high average-utility itemsets with an improved upper-bound strategy. *Int. J. Inf. Technol. Decis. Mak.* **2012**, *11*, 1009–1030. [CrossRef]
24. Lu, T.; Vo, B.; Nguyen, H.T.; Hong, T.P. A new method for mining high average utility itemsets. In Proceedings of the IFIP International Conference on Computer Information Systems and Industrial Management, Ho Chi Minh, Vietnam, 5–7 November 2014; pp. 33–42.
25. Cheung, D.W.; Wong, C.Y.; Han, J.; Ng, V.T. Maintenance of discovered association rules in large databases: An incremental updating techniques. In Proceedings of the IEEE International Conference on Data Engineering, New Orleans, LA, USA, 26 February–1 March 1996; pp. 106–114.
26. Hong, T.P.; Lin, C.W.; Wu, Y.L. Incrementally fast updated frequent pattern trees. *Expert Syst. Appl.* **2008**, *34*, 2424–2435. [CrossRef]

27. Lin, C.W.; Lan, G.C.; Hong, T.P. An incremental mining algorithm for high utility itemsets. *Expert Syst. Appl.* **2009**, *39*, 7173–7180. [CrossRef]
28. Tseng, V.S.; Shie, B.E.; Wu, C.W.; Yu, P.S. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Trans. Knowl. Data Eng.* **2013**, *25*, 1772–1786. [CrossRef]
29. Lin, J.C.W.; Ren, S.; Fournier-Viger, P.; Pan, J.S.; Hong, T.P. Efficiently updating the discovered high average-utility itemsets with transaction insertion. *Eng. Appl. Artif. Intell.* **2018**, *72*, 136–149. [CrossRef]
30. Fournier-Viger, P.; Lin, J.C.W.; Gomariz, A.; Gueniche, T.; Soltani, A.; Deng, Z.; Lam, H.T. The SPMF open-source data mining library version 2. In Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Riva del Garda, Italy, 19–23 September 2016; pp. 36–40.
31. Agrawal, R.; Srikant, R. Quest Synthetic Data Generator, 1994. Available online: http://www.Almaden.ibm.com/cs/quest/syndata.html (accessed on 10 March 2018).