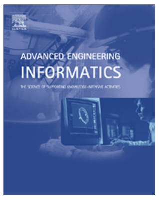
Advanced Engineering Informatics 30 (2016) 233–243

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei



Full length article

An efficient algorithm to mine high average-utility itemsets

Jerry Chun-Wei Lin [a,*], Ting Li [a], Philippe Fournier-Viger [b], Tzung-Pei Hong [c,d], Justin Zhan [e], Miroslav Voznak [f]

[a] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen Graduate School, Shenzhen, China
[b] School of Natural Sciences and Humanities, Harbin Institute of Technology, Shenzhen Graduate School, Shenzhen, China
[c] Department of Computer Science and Engineering, National University of Kaohsiung, Kaohsiung, Taiwan
[d] Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
[e] Department of Computer Science, University of Nevada, Las Vegas, USA
[f] Department of Telecommunications, VSB-Technical University of Ostrava, Czech Republic

ARTICLE INFO

Article history:
Received 24 December 2015
Received in revised form 26 March 2016
Accepted 3 April 2016
Available online 15 April 2016

Keywords:
High average-utility itemsets
List structure
Data mining
HAUIM

ABSTRACT

With the ever increasing number of applications of data mining, high-utility itemset mining (HUIM) has become a critical issue in recent decades. In traditional HUIM, the utility of an itemset is defined as the sum of the utilities of its items, in transactions where it appears. An important problem with this definition is that it does not take itemset length into account. Because the utility of larger itemset is generally greater than the utility of smaller itemset, traditional HUIM algorithms tend to be biased toward finding a set of large itemsets. Thus, this definition is not a fair measurement of utility. To provide a better assessment of each itemset's utility, the task of high average-utility itemset mining (HAUIM) was proposed. It introduces the average utility measure, which considers both the length of itemsets and their utilities, and is thus more appropriate in real-world situations. Several algorithms have been designed for this task. They can be generally categorized as either level-wise or pattern-growth approaches. Both of them require, however, the amount of computation to find the actual high average-utility itemsets (HAUIs). In this paper, we present an efficient average-utility (AU)-list structure to discover the HAUIs more efficiently. A depth-first search algorithm named HAUI-Miner is proposed to explore the search space without candidate generation, and an efficient pruning strategy is developed to reduce the search space and speed up the mining process. Extensive experiments are conducted to compare the performance of HAUI-Miner with the state-of-the-art HAUIM algorithms in terms of runtime, number of determining nodes, memory usage and scalability.

1. Introduction

Mining frequent itemsets (FIs) or association rules (ARs) in transactional databases is a fundamental task in knowledge discovery in databases (KDD) [2,3,6]. Many algorithms have been designed to mine FIs or ARs. The most common ways of deriving FIs or ARs from a database are to use a level-wise [3] or a pattern-growth approach [8,14]. Apriori [3] is the first algorithm to mine FIs in a level-wise manner. It relies on a minimum support threshold in the first phase to mine FIs, and then use the discovered FIs in the second phase to derive ARs satisfying a minimum confidence threshold. The pattern-growth approach was introduced by Han et al. [8] for mining FIs without candidate generation. FP-growth initially builds an FP-tree structure using frequent 1-itemsets. Then, during the mining process, conditional FP-trees are recursively generated, and each tree contains a designed index table (Header_Table) for mining the FIs.

Traditional frequent itemset mining (FIM) and association rule mining (ARM) algorithms only consider occurrence frequencies of items in binary databases. Other important factors such as quantities, profits, and weights of items are not taken into account by traditional FIM and ARM algorithms. Another problem is that FIs and ARs found a transaction database may only contribute a small portion of the overall profit generated by the sale of items, and infrequent itemsets may contribute a large amount of the profit. For example, the sale of diamonds may be less frequent than that of clothing or shoes in a shopping mall, but diamonds generally contribute a much higher profit per unit sold. It is thus obvious that only considering the occurrence frequency is insufficient to

* Corresponding author.
E-mail addresses: jerrylin@ieee.org (J.C.-W. Lin), tingli@ikelab.net (T. Li), philfv@hitsz.edu.cn (P. Fournier-Viger), tphong@nuk.edu.tw (T.-P. Hong), justin.zhan@unlv.edu (J. Zhan), miroslav.voznak@vsb.cz (M. Voznak).

http://dx.doi.org/10.1016/j.aei.2016.04.002
1474-0346/© 2016 Elsevier Ltd. All rights reserved.

identify highly profitable itemsets or itemsets that are generally more important to the user. Thus, high-utility itemset mining (HUIM) [11–13,22] has emerged as a critical issue in recent decades, as it can reveal the profitable itemsets in real-world situations. HUIM can be considered as an extension of FIM that considers additional information such as quantities and unit profits of items, to better assess how "useful" an itemset is to the user. An item/set is considered as a high-utility itemset (HUI) if its utility is no less than a user-defined minimum utility threshold. Since the downward closure (DC) property used in traditional FIM and ARM does not hold in traditional HUIM, Liu et al. [12] designed a two-phase approach and developed a transaction-weighted downward closure (TWDC) property to reduce the search space by pruning unpromising itemsets early. Several level-wise and pattern-growth algorithms have been proposed to efficiently mine HUIs, using the two-phase approach [4,5,7].

In traditional HUIM, the utility of an item/set is defined as the sum of its utilities in the database. An important problem with this definition is that it does not take itemset length into account. Thus, this definition is not a fair measurement of utility. To provide a better assessment of each itemset's utility, the task of high average-utility itemset mining (HAUIM) was proposed by Hong et al. [9]. The proposed average utility measure estimates the utility of an itemset by considering its length. It is defined as the sum of the utilities of the itemset in transactions where it appears, divided by the number of items that it contains. This measure addresses the bias of traditional HUIM toward larger itemsets, by considering the length of itemsets, and can thus more objectively assess the utility of itemsets. As for traditional HUIM, level-wise and pattern-growth algorithms have been designed for HAUIM. Level-wise algorithms [9] require to generate numerous candidates for mining the actual high average-utility itemsets (HAUIs). Pattern-growth algorithms [15] require to recursively build conditional trees for mining HAUIs, which is quite time-consuming. In this paper, we first design an efficient average-utility (AU)-list structure and develop an algorithm named HAUI-Miner for mining HAUIs using a single phase. The key contributions of this paper are threefold.

1. We first design an efficient HAUI-Miner algorithm to mine high average-utility itemsets (HAUIs). It relies on a novel condensed average-utility (AU)-list structure. This structure only keeps information required by the mining process, thus compressing very large databases into a condensed structure.
2. An efficient pruning strategy is developed to reduce the search space, represented as an enumeration tree, by pruning unpromising candidates early. Using this strategy, building the AU-lists of extensions of a processed node in the enumeration tree can be avoided to reduce the amount of computation.
3. Substantial experiments are conducted to compare the performance of the designed HAUI-Miner algorithm with the state-of-the-art algorithms, in terms of runtime, number of determining nodes, memory consumption, and scalability.

2. Related work

High-utility itemset mining (HUIM) [12,13,22], an extension of frequent itemset mining, is based on the measurement of internal utility and external utility. The internal utility of an item is its purchase quantity in a transaction, and the external utility of an item can be viewed as its unit profit, importance or weight. The utility of an item/set in a database is calculated as the total purchase quantity of the itemset in the database, multiplied by its unit profit (external utility). The purpose of HUIM is to discover the complete set of high-utility itemsets (HUIs), that are itemsets having a utility no less than a minimum utility threshold. Yao et al. [22] proposed a framework for mining HUIs based on mathematical properties of the utility measure. Two pruning strategies were designed to reduce the search space for discovering HUIs respectively based on utility upper bounds and expected utility upper bounds. Since the downward closure (DC) property of ARM does not hold in traditional HUIM, Liu et al. [12] then designed a transaction-weighted downward closure (TWDC) property and developed the transaction-weighted utilization (TWU) model. This latter provides upper bounds on the utilities of potential HUIs, which can be used to reduce the combinatorial explosion of the search space in traditional HUIM. However, the TWU model still requires to generate numerous candidates to obtain the actual HUIs. Pattern-growth algorithms have been proposed to compress the database into a condense tree structure using the TWU model. Lin et al. [16] designed a high-utility pattern (HUP)-tree algorithm to recursively mine high-utility itemsets using the proposed tree structure. Tseng et al. developed the UP-Growth [20] and UP-Growth+ [21] algorithms to efficiently discover HUIs based on different pruning strategies. The aforementioned approaches all rely on the TWU model and its TWDC property for discovering HUIs. The search space is, however, very large when using the TWU model, and it is thus very time-consuming to discover the actual HUIs. As an alternative to the pattern-growth mechanism, Liu et al. [13] developed the list-based HUI-Miner algorithm to discover HUIs without candidate generation. The developed utility-list structure is an efficient structure for maintaining the information required for mining HUIs using a limited amount of memory. Fournier-Viger et al. [7] extended HUI-Miner with a structure named EUCS to store information about the relationships between 2-itemsets, thus speeding up the discovery of HUIs. Several extensions of the task of HUIM have been proposed such as discovering up-to-date HUIs [17] and top-k HUIs [23].

Similarly to traditional HUIM, several HAUIM algorithms have been designed using the TWU model. Lin et al. [15] first developed the HAUP-tree structure and the HAUP-growth algorithm for mining HAUIs. In the HAUP-tree, each node at the end of a path stores the average-utility upper bound of the corresponding item as well as the quantities of the preceding items in the same path. This approach can thus be used to speed up the discovery of HAUIs. Lan et al. [10] proposed a projection-based average-utility itemset mining (PAI) algorithm to reveal HAUIs using a level-wise approach. Based on the proposed upper-bound model, the number of unpromising candidates can be greatly reduced compared to previous work based on the TWU model. Lu et al. [18] proposed the HAUI-tree algorithm to further reduce the number of unpromising candidates for mining the actual HAUIs using a designed enumeration tree structure. However, mining HAUIs using the designed algorithm is still very time-consuming since the upper-bounds used by these algorithms are loose, and thus numerous unpromising candidates need to be generated, and the recursive process for building the complete enumeration tree remains costly.

3. Preliminaries and problem statement

3.1. Preliminaries

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a finite set of m distinct items. A quantitative database is a set of transactions $D = \{T_1, T_2, \ldots, T_n\}$, where each transaction $T_q \in D$ $(1 \leqslant q \leqslant m)$ is a subset of I and has a unique identifier q, called its *TID*. Besides, each item i_j in a transaction T_q has a purchase quantity denoted as $q(i_j, T_q)$. A profit table PT indicates the unit profit value of each item in the database as $PT = \{pr(i_1), pr(i_2), \ldots, pr(i_m)\}$, where profit values are positive integers. A set of k distinct items $X = \{i_1, i_2, \ldots, i_k\}$ such that $X \subseteq I$ is said to be a k-itemset, where k is the length of the itemset. An itemset X is said to be contained in a transaction T_q if $X \subseteq T_q$. A minimum average-utility threshold δ is set according to the user's preference (a positive integer). An example quantitative database is shown in Table 1, which will be used as running example for the rest of this paper. This database contains six transactions and six distinct items, denoted with letters from (A) to (F). The profit table indicates the unit profit of each item appearing in the database, and is shown in Table 2. In the running example, the minimum average-utility threshold is set to (δ = 16%).

Definition 1. The average-utility of an item i_j in a transaction T_q is denoted as $au(i_j, T_q)$, and defined as:

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

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

For example, the average-utility of items (A), (B), (C), (D), and (F) in T_1 are respectively calculated as $au(A, T_1)(= \frac{1\times5}{1})(= 5)$, $au(B, T_1)(= \frac{6\times1}{1})(= 6)$, $au(C, T_1)(= \frac{3\times2}{1})(= 6)$, $au(D, T_1)(= \frac{3\times3}{1})(= 9)$, and $au(F, T_1)(= \frac{6\times1}{1})(= 6)$.

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

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

where k is the number of items in X.

For example, the average-utility of itemsets (AB) and (ABC) in T_1 are respectively calculated as $au(AB) = \frac{1\times5+6\times1}{2} = (5.5)$ and $au(ABC) = \frac{1\times5+6\times1+3\times2}{3}$ (=5.66).

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

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

For example, the average-utilities of itemsets (AB) and (ABC) in the database depicted in Table 1 are respectively calculated as $au(AB) = au(AB, T_1) + au(AB, T_4) + au(AB, T_5) = 5.5 + 7 + 12$ (=24.5), and $au(ABC) = au(ABC, T_1) + au(ABC, T_4) + au(ABC, T_5) = 5.66 + 6.66 + 10$ (=22.32).

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

$$tu(T_q) = \sum_{i_j \in T_q} u(i_j, T_q). \tag{4}$$

For example, the utilities of transactions in Table 1 are respectively calculated as $tu(T_1) = 5 + 6 + 6 + 9 + 6$ (=32), $tu(T_2)$(=16), $tu(T_3)$(=22), $tu(T_4)$(=28), $tu(T_5)$(=37), and $tu(T_6)$(=15).

Definition 5. The total utility of a database D is denoted as TU, and defined as the sum of all transaction utilities, that is:

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

For example, the total utility in the running example of Table 1 is calculated as $TU = 32 + 16 + 22 + 28 + 37 + 15$ (=150).

3.2. Problem statement

The problem of mining high average-utility itemsets is to discover the complete set of high average-utility itemsets (HAUIs). An itemset X is an HAUI in a database D if its utility is no less than the minimum average-utility count, specified by the user. The set of HAUIs is thus formally defined as:

$$HAUIs \leftarrow \{X | au(X) \geqslant TU \times \delta\}. \tag{6}$$

4. The proposed HAUI-Miner algorithm

In this paper, we design an average-utility (AU)-list structure to store the information needed by the mining process. Moreover, an algorithm named HAUI-Miner is also developed to mine HAUIs more efficiently than previous works. In traditional association rule mining (ARM), the downward closure (DC) property is used to reduce the search space and avoid the problem of the combinatorial explosion for mining HAUIs. In HAUIM, this property does not hold for the average utility measure. To restore this property and effectively reduce the search space, this paper introduces a transaction-maximum utility downward closure (TMUDC) property. It allows to prune unpromising candidates early, and thus to reduce the search space to efficiently discover the actual HAUIs.

Definition 6. The transaction-maximum utility of a transaction T_q is denoted as $tmu(T_q)$, and defined as the maximum utility of items in a transaction T_q, that is:

$$tmu(T_q) = max(\{u(i_j) | i_j \in T_q\}). \tag{7}$$

For example, the transaction-maximum utility of T_1 is calculated as $mu(T_1) = max\{5, 6, 6, 9, 6\}$(=9). The transaction-maximum utilities of the other transactions are calculated in the same say, and are shown in Table 3.

Definition 7. The average-utility upper-bound of an itemset X is denoted as $auub(X)$, and defined as the sum of the transaction-maximum utilities of transactions containing X, that is:

$$auub(X) = \sum_{X \subseteq T_q \wedge T_q \in D} tmu(T_q). \tag{8}$$

Table 1
A quantitative database.

TID	Transaction (item, quantity)
1	A:1, B:6, C:3, D:3, F:6
2	B:2, C:3, E:2
3	A:2, C:1, D:2, E:1
4	A:1, B:9, C:3, D:2, F:2
5	A:3, B:9, C:3, D:1, E:1
6	C:4, D:1, E:1

Table 2
A profit table.

Item	Profit
A	5
B	1
C	2
D	3
E	4
F	1

Table 3
The transaction-maximum utilities.

TID	Transaction (item, quantity)	*tmu*
1	*A*:1, *B*:6, *C*:3, *D*:3, *F*:6	9
2	*B*:2, *C*:3, *E*:2	8
3	*A*:2, *C*:1, *D*:2, *E*:1	10
4	*A*:1, *B*:9, *C*:3, *D*:2, *F*:2	9
5	*A*:3, *B*:9, *C*:3, *D*:1, *E*:1	15
6	*C*:4, *D*:1, *E*:1	8

For example, consider the database of Table 1. The value $auub(A)$ is calculated as $auub(A) = tmu(T_1) + tmu(T_3) + tmu(T_4) + tmu(T_5) = 9 + 10 + 9 + 15$ (=43). The average-utility-upper-bounds of other items are calculated in the same way, and are given in Table 4.

Definition 8. An itemset X is called a high average-utility upper-bound itemset (*HAAUUBI*) if its average-utility upper-bound is no less than the minimum average-utility count, which is defined as:

$$HAUUBI \leftarrow \{X | auub(X) \geqslant TU \times \delta\}. \tag{9}$$

Theorem 1 (Transaction-maximum-utility downward closure (*TMUDC*) property of *HAUUBI*). *The average-utility-upper-bound measure is downward-closed. The TMUDC property holds for any HAUUBI itemsets.*

Proof. Let X^k be a k-itemset, and X^{k-1} be any of its subsets. Since X^{k-1} is a subset of X^k, the set of *TIDs* of transactions containing X^k is a subset of the set of *TIDs* of X^{k-1}. Assume that X^k is a *HAUUBI*. Thus:

$$auub(X^k) = \sum_{X^k \subseteq T_q \wedge T_q \in D} tmu(T_q) \leqslant \sum_{X^{k-1} \subseteq T_q \wedge T_q \in D} tmu(T_q) = auub(X^{k-1}).$$

$$\Rightarrow auub(X^k) \leqslant auub(X^{k-1}). \square$$

By Theorem 1, $auub(X^{k-1}) \geqslant auub(X^k)$. Therefore, if X^k is a *HAUUBI*, any subset X^{k-1} of X^k is also a *HAUUBI*.

Corollary 1. *If an itemset X^k is a HAUUBI, all subsets of X^k are also HAUUBIs.*

Corollary 2. *If an itemset X^k is not a HAUUBI, all supersets of X^k are not HAUUBIs.*

Theorem 2 (*HAUUBI* ⊆ *HAUIs*). *The TMUDC property ensures that HAUUBI ⊆ HAUIs. Thus, if an itemset is not a HAUUBI, none of its supersets are HAUIs.*

Table 4
The average-utility upper-bounds of items.

Item	*auub*
A	43
B	41
C	59
D	51
E	41
F	18

Proof. $\forall X$ in D,

$$au(X) = \textstyle\sum_{X \subseteq T_q \wedge T_q \in D} au(X, T_q) \leqslant \displaystyle\sum_{X \subseteq T_q \wedge T_q \in D} \frac{tmu(X, T_q) \times |X|}{|X|} = \sum_{X \subseteq T_q \wedge T_q \in D} tmu(X, T_q) = auub(X). \square$$

$$\Rightarrow au(X^k) \leqslant auub(X^k).$$

Thus, if an itemset X is not a *HAUUBI*, it is also not a *HAUI*. This property can be used to reduce the search space by pruning numerous unpromising candidates, which speeds up the mining process.

4.1. The revised and projected databases

The proposed HAUI-Miner algorithm scans the database twice to calculate tight upper bounds on the average-utilities of candidate itemsets. During the first database scan, the set of high average-utility-upper-bound 1-itemsets (1-*HAUUBIs*) is discovered. This latter is needed to construct the AU-lists of 1-itemsets. During the second database scan, 1-itemsets that are deemed non-*HAUUBI* (according to the minimum average-utility count) are removed. In other words, for an itemset X, if $auub(X)$ is less than the minimum average-utility count ($\delta \times TU$), X is not a *HAUUBI*, and X can thus be removed from the database. The database obtained after removing all such items from a database D is called the *revised database* of D, and is denoted as D'. The pseudocode of the algorithm for obtaining 1-*HAUUBIs* and the revised database, is presented in Algorithm 1.

Algorithm 1. Algorithm to obtain 1-*HAUUBIs* and the revised database

```
Input: $D$, a quantitative database; ptable, a profit table, and
 the minimum average-utility threshold $\delta$.
Output: the set of high average-utility upper-bound
 itemsets 1-(HAUUBIs).
1 for each transaction $T_q$ in $D$ do
2     scan $T_q$ to calculate $tmu(T_q)$;
3 for each item $i$ appearing in $D$ do
4     calculate $auub(i)$;
5 1-HAUUBIs $\leftarrow \{i | auub(i) \geqslant \delta \times TU\}$;
6 scan $D$ to remove each item $i$ such that $i \notin$ 1-HAUUBIs, and
  obtain the revised database $D'$;
7 for each transaction $T_q \in D'$ do
8     sort $T_q$ in ascending order $\prec$ of $auub(i)$;
9 return the revised database $D'$ and 1-HAUUBIs;
```

In Algorithm 1, the database is initially scanned to find the transaction-maximum utility of each transaction. This is done by comparing the utility values of items for each transaction (Lines 1–2). After that, the average-utility upper-bound value of each item is calculated (Lines 3–4) and unpromising 1-*HAUUBIs* are removed from the database to obtain the revised database (Lines 5–6). The remaining items in each transaction in the revised database are then sorted in ascending order of *auub* values (Lines 7–8). After that, the revised database and the discovered *HAUUBIs* are returned by the method, and will be used by the mining process (Line 9). The revised database for the running example is shown in Table 5, where the discovered 1-*HAUUBIs* and their sorted order is $\{B \prec E \prec A \prec D \prec C\}$.

After the original database has been revised, the sub-database corresponding to each item in 1-*HAUUBIs* is then projected, forming a smaller database that is used for constructing the corresponding AU-list. If an item is not an *HAUUBI* in the sub-database, the item is removed from the sub-database. In this way, a projected database is smaller than the revised database, and can thus accelerate the construction of AU-lists. For example, consider item (B) in Table 5. Transactions are projected, forming the sub-database of (B). The result is shown in Table 6. Then, the *auub* value of each item in the sub-database is compared with the minimum average-utility count to determine if it satisfies the condition to be an *HAUUBI*. In the example of Table 6, the transaction-maximum utilities of the four transactions are respectively: $tmu(T_1) = 9$, $tmu(T_2) = 8$, $tmu(T_3) = 9$, and $tmu(T_4) = 15$; the *auub* value of each item is calculated as: $auub(B) = tmu(T_1) + tmu(T_2) + tmu(T_3) + tmu(T_4) = 41$, $auub(E) = 23$, $auub(A) = 33$, $auub(D) = 33$, and $auub(C) = 41$. Since $auub(E)(=23 < 24)$, the item (E) is not a 1-*HAUUBI* in the sub-database of (B). Thus, (E) is removed from Table 6, and the result of this process is shown in Table 7, which is called the *projected sub-database* of (B).

Table 5
The revised database.

TID	Items
1	B:6, A:1, D:3, C:3
2	B:2, E:2, C:3
3	E:1, A:2, D:2, C:1
4	B:9, A:1, D:2, C:3
5	B:9, E:1, A:3, D:1, C:3
6	E:1, D:1, C:4

Table 6
The projected sub-database of (B).

TID	Items
1	B:6, A:1, D:3, C:3
2	B:2, E:2, C:3
3	B:9, A:1, D:2, C:3
4	B:9, E:1, A:3, D:1, C:3

Table 7
The revised projected sub-database of (B).

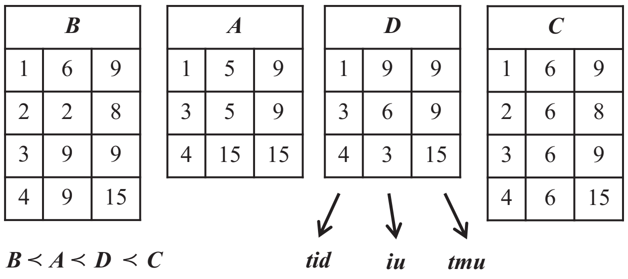
TID	Items
1	B:6, A:1, D:3, C:3
2	B:2, C:3
3	B:9, A:1, D:2, C:3
4	B:9, A:3, D:1, C:3

4.2. The average-utility (AU)-list structure

A projected database that has been revised twice can then be used to efficiently construct the average-utility-list (AU-list) structure of each item/set. The AU-list of an item/set X is a list of elements such that there is an element representing each transaction T_q where $X \subseteq T_q$. An element consists of three fields, defined as follows:

- The *tid* field indicates the transaction of T_q.
- The *iu* field indicates the utility of X in T_q, i.e., $u(X, T_q)$.
- The *tmu* field indicates the transaction-maximum utility of X in T_q, i.e., $tmu(X, T_q)$.

AU-lists constructed using the projected sub-database of (B), depicted in Table 7, are shown in Fig. 1.



B		
1	6	9
2	2	8
3	9	9
4	9	15

A		
1	5	9
3	5	9
4	15	15

D		
1	9	9
3	6	9
4	3	15

C		
1	6	9
2	6	8
3	6	9
4	6	15

Fig. 1. The AU-lists constructed using the projected sub-database of (B).

In Fig. 1, the first element (1,6,9) in the constructed AU-list of (B) indicates that (B) appears in transaction T_1, has a utility of 6 in that transaction, and that the transaction-maximum utility of (B) in that transaction is 9. If the sum of the $u(X)$ values of all elements in an AU-list is no less than the minimum average-utility count, it is directly output as a high average-utility itemset (HAUI). To construct AU-lists of k-itemset $(k \geqslant 2)$, it is unnecessary to rescan the original database. They can be constructed by performing an intersection operation using AU-lists of smaller itemsets (by comparing TIDs in AU-lists). Suppose that the lengths of two itemsets are respectively m and n. Performing the intersection of two AU-lists only requires at most $(m + n)$ comparisons for deriving the AU-list of a k-itemset. The construction algorithm of AU-lists for k-itemsets $(k \geqslant 2)$ is shown in Algorithm 2.

Algorithm 2. AU-list construction

```
Input: $P.AUL$, the AU-list of itemset $P$; $P_x.AUL$, the AU-list of itemset $P_x$; $P_y.AUL$, the AU-list of itemset $P_y$.
Output $P_{xy}.AUL$, the AU-list of itemset $P_{xy}$.
1  $P_{xy}.AUL \leftarrow null$;
2  $P_{xy}.AUL \leftarrow null$;
3  for each element $E_x \in P_x.AUL$ do
4      if $\exists E_y \in P_y.AUL \wedge E_x.tid = E_y.tid$ then
5          if $P.AUL = null$ then
6              $E_{xy} \leftarrow \langle E_x.tid, E_x.u + E_y.u, E_y.tmu \rangle$;
7          else
8              find $E \in P.AUL$ such that $E.tid = E_x.tid$;
9              $E_{xy} \leftarrow \langle E_x.tid, E_x.u + E_y.u, E_y.tmu \rangle$;
10         $P_{xy}.AUL \cup E_{xy}$;
11 return $P_{xy}.AUL$;
```

The AU-list construction algorithm takes the AU-lists of three itemsets as input $(P, P_x$ and $P_y)$. Here, the notation P_x denotes the union of itemset P with an item x, i.e. $P \cup \{x\}$. It is assumed that $x \prec y$. The output of the algorithm is the AU-list of itemset P_{xy}. The algorithm is applied as follows. First, the AU-list of P_{xy} is initialized as empty (Line 1). A loop is then performed over all elements E_x in $P_x.AUL$ (Lines 2–10) to fill the AU-list of P_{xy} by combining $P_x.AUL$ and $P_y.AUL$. If $P = \emptyset, P_{xy}$ is a 2-itemset, and elements in its AU-list structure are directly constructed by comparing elements in the AU-lists of P_x and P_y (Lines 4–5). Otherwise, P_{xy} is a k-itemset $(k \geqslant 3)$, and its AU-list is constructed by also considering the AU-list of P (Lines 7–10). For example, consider the construction of the AU-list of (BA) using the AU-lists of (B) and (A), as illustrated in Fig. 1. This is done by performing the intersection of elements having the same TIDs in the AU-lists of (B) and (A). The set of TIDs of (BA) is thus $\{1,2,3,4\} \cap \{1,3,4\} = \{1,3,4\}$. The values of u and tmu in the AU-lists are also updated. The constructed AU-list of (BA) is shown in Fig. 2.

4.3. Search space of AU-list

Based on the designed AU-list structure, the search space for mining high average-utility itemsets (HAUIs) can be represented as an enumeration tree, where each node represents a distinct itemset, which may be a potential HAUI. The proposed algorithm explores this tree using a depth-first search. It relies on a pruning strategy (presented in the next sub-section) to determine whether the AU-lists of the child nodes of the processed one need to be constructed or can be pruned directly. The sub-tree of itemset (B) for the running example of Table 7 is shown in Fig. 3.

4.4. Pruning strategy

Mining HAUIs is very costly in terms of runtime and memory if the search space is very large (if a huge amount of itemsets is considered). Suppose that the number of items in a database is k. A naive approach would require to consider all $2^k - 1$ possible non-empty itemsets as potential HUIs. To avoid this combinatorial explosion, this paper introduces an efficient pruning strategy that is effective at reducing the search space.

Definition 9. Let $SUM.X.iu$ denotes the sum of the utilities of an itemset X in a database D, that is:

$$SUM.X.iu = \sum_{X \subseteq T_q \wedge T_q \in D} u(X, T_q). \tag{10}$$

For example in Fig. 2, $SUM.B.iu$ (=6 + 2 + 9 + 9)(=26), and $SUM.A.iu$ (=5 + 5 + 15)(=25).

Definition 10. Let $SUM.X.tmu$ denotes the sum of the transaction-maximum utilities of transactions containing an itemset X in a database D, that is:

$$SUM.X.tmu = \sum_{X \subseteq T_q \wedge T_q \in D} tmu(X, T_q). \tag{11}$$

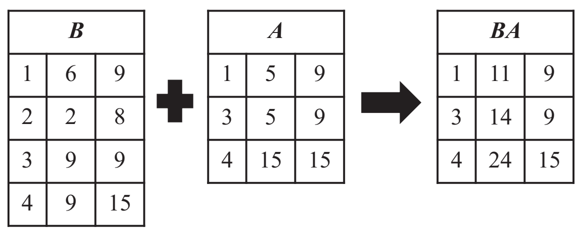
For example in Fig. 2, $SUM.B.tmu$ = 9 + 8 + 9 + 15 (=41), and $SUM.A.tmu$ = 9 + 9 + 15 (=33).

Definition 11. Given an itemset X and a transaction T such that $X \subseteq T$, the set of all items appearing after X in T is denoted as T/X and defined as $T/X = \{i | i \in T \wedge i \succ j \; \forall j \in X\}$.

For example in Table 5, $T_1/(B)$ = (ADC), and $T_1/(A)$ = (DC).

Definition 12. Let there be some itemsets X and Y. Y is said to be an extension of X if there exists an itemset $Z \neq \emptyset$ such that $Y = X \cup Z$, and $\forall j \in Z$, $\nexists i \in X$ such that $i \succ j$. Furthermore, Y is said to be a 1-extension of X if it is an extension of X and $|Z| = 1$.

To mine HAUIs efficiently, it is necessary to reduce the search space. This can be done by identifying and pruning unpromising itemsets early. In the designed AU-list structure, the sum of the *iu* and *tmu* fields provides enough information to achieve this goal.



B		
1	6	9
2	2	8
3	9	9
4	9	15

A		
1	5	9
3	5	9
4	15	15

BA		
1	11	9
3	14	9
4	24	15

Fig. 2. The construction of the AU-list of (BA).

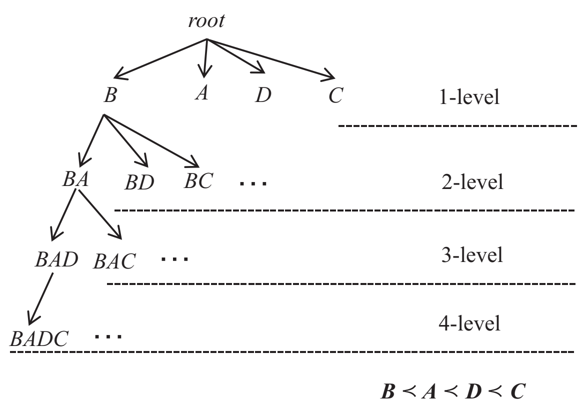


Fig. 3. The enumeration tree of (B).

Theorem 3. *Let there be an itemset X. If the value $SUM.X.tmu$ calculated using the AU-list of X is less than the minimum average-utility count, all extensions of X are not high average-utility itemsets (HAUIs).*

Proof. $\because i \in X \subseteq T$, and $u(i,T) \leqslant tmu(i,T)$,
$\therefore u(X,T) = \sum_{i \in T} u(i,T) \leqslant \sum_{i \in T} tmu(X,T) = tmu(X,T) \times |X|$.
$\forall T \supseteq X', \because X'$ is an extension of X.
$\therefore au(X',T) = \frac{u(X',T)}{|X'|} = \frac{\sum_{i \in T} u(i,T)}{|X'|} \leqslant \frac{tmu(X',T) \times |X'|}{|X'|} = tmu(X',T)$.
Let $id(T)$ denotes the tid of transaction T, and $X.tids$ denotes the set of $tids$ in the AU-list of X, and $X'.tids$ in X'. Thus:
$\because X \subset X' \Rightarrow X'.tids \subseteq X.tids$.

$$\therefore au(X') = \sum_{id(T) \in X'.tids} au(X',T) \leqslant \sum_{id(T) \in X'.tids} tmu(X',T) \leqslant \sum_{id(T) \in X.tids} tmu(X,T) = SUM.X.tmu. \qquad \square$$

Thus, if the sum of the transaction-maximum utilities of the transactions containing an itemset X is less than the minimum average-utility count, all extensions of X are not high average-utility itemsets (HAUIs) and can thus be ignored, and their AU-lists do not need to be constructed. For example, consider the 3-itemset (BAD), which is an extension of the 2-itemset (BA). The sum of the *tmu* values of (BA) is calculated as (9 + 9 + 15)(=33), which is larger than the minimum average-utility count (33 > 24). Thus, the AU-lists of extensions of (BA) in the enumeration tree need to be considered by the depth-first search and the AU-list of its 1-extensions need to be generated. The full pseudocode of the proposed HAUI-Miner algorithm is presented in Algorithm 3.

Algorithm 3. HAUI-Miner

```
Input: $P.AUL$, the AU-list of itemset $P$; $AULs$, the set of AU-list of all $P$'s 1-extensions; $\delta$, the minimum average-utility threshold; $TU$, the total utility in $D$.
Output: all the high average-utility itemsets with $P$ as the prefix.
1 for each AU-list $Y.AUL$ in $AULs$ do
2     if $\frac{SUM.Y.iu}{|Y.AUL|} \geqslant \delta \times TU$ then
3         $HAUIs \leftarrow HAUIs \cup Y$;
4     if $SUM.Y.tmu \geqslant \delta \times TU$ then
5         $extAULs \leftarrow null$;
6         for each AU-list $Z.AUL$ after $Y.AUL$ in $AULs$ do
7             $extAULs \leftarrow extAULs \cup Construct(P.AUL, Y.AUL, Z.AUL)$;
8         HAUI-Miner($Y.AUL, extAULs, \delta \times TU$);
```

Table 8
Final derived HAUIs.

Itemset	*au*	Itemset	*au*
(B)	26	(BC)	25
(A)	35	(AD)	29.5
(D)	27	(AC)	27.5
(C)	34	(DC)	27.5
(BA)	24.5	(ADC)	26.3

Table 9
Parameters used to describe datasets.

#|D|	Total number of transactions
#|I|	Number of distinct items
AvgLen	Average transaction length
MaxLen	Maximal transaction length
Type	Dataset type

Table 10
Characteristics of the datasets.

Dataset	#|D|	#|I|	AvgLen	MaxLen	Type
Chess	3196	75	37	37	Dense
Mushroom	8124	119	23	23	Dense
SIGN	730	267	52	94	Sparse
Retail	88,162	16,470	10.3	76	Sparse
T10I4D100K	100,000	870	10.1	29	Sparse
Kosarak	990,002	41,270	8.1	2498	Sparse

The HAUI-Miner algorithm takes as input (1) the AU-list of an itemset P, (2) the AU-lists $P.AUL$ of all 1-extensions of P, (3) the predefined high average-utility threshold, and (4) the total utility of the database D. The algorithm first loops to consider each AU-list $Y.AUL$ in $P.AUL$. If the sum of the transaction maximum utilities in $Y.AUL$ divided by the number of elements in $Y.AUL$ is less than the minimum average-utility count ($=\delta \times TU$), Y is an HAUI and it is output (Lines 2–3). Then, if the sum of the transaction-maximum utilities in $Y.AUL$ exceeds δ (line 4), the designed algorithm will explore the search space by intersecting Y and each AU-list Z appearing after Y in *AULs* (Lines 6–7). The construction function $(P.AUL, Y, Z)$ (Line 7) is then called to construct the AU-list of the itemset $Y \cup Z$. Then the algorithm is called recursively to explore extensions of $Y \cup Z$ (Line 8). When the algorithm terminates, all HAUIs have been successfully discovered.

5. An illustrated example

In this section, a simple example is given to illustrate how the proposed HAUI-Miner algorithm is applied step-by-step to discover HAUIs. Consider the example dataset and profit table respectively shown in Tables 1 and 2. Moreover, assume that the minimum average-utility threshold δ is set to 16% by the user. The dataset is first scanned to calculate the *auub* values of all 1-items. The results are shown in Table 4. Since the minimum average-utility count is $(150 \times 0.16)(=24)$, the set of 1-*HAUUBIs* is $\{A, B, C, D, E\}$. The database is then revised to remove unpromising itemsets and the remaining items are sorted in ascending order of their *auub* values in each transaction. The resulting revised database is shown in Table 5.

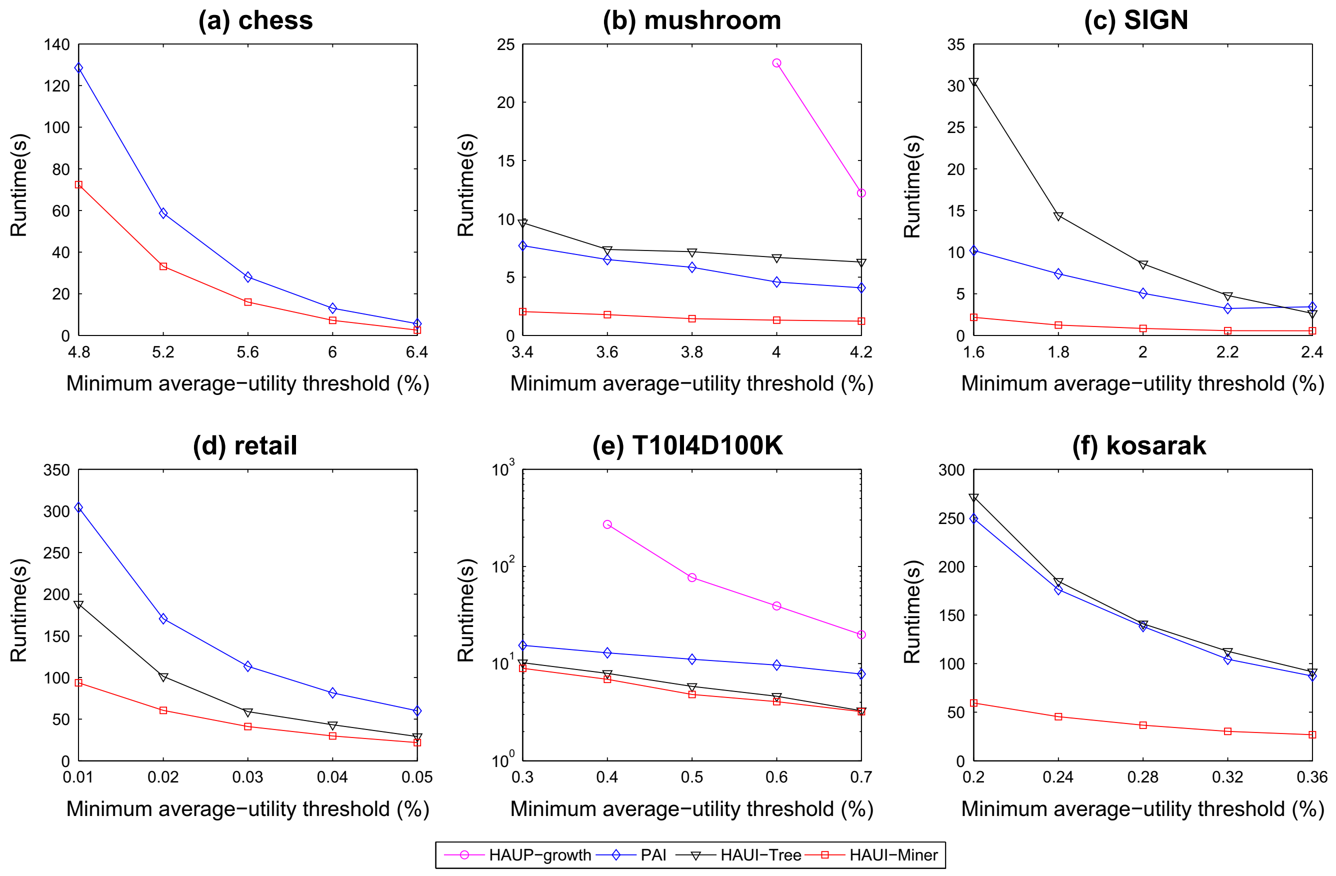


Fig. 4. Runtimes for various minimum average-utility threshold values.

After that, the projected sub-databases of items in the set of 1-*HAUUBIs* are generated. The projected sub-database of (B) is shown in Table 6. In each projected database, the local *HAUUBIs* are identified, and their utility-lists are constructed. For instance, the local AU-lists of (B), (A), (D), and (C) are built using the projected database of item (B), and are shown in Fig. 1. The AU-lists of (B)'s 1-extensions (BA, BD, BC) are then constructed by intersecting the AU-list of (B) with the AU-lists of the other 1-items. Using the constructed AU-lists of (B)'s 1-extension, it is found that $au(BA)(=24.5)$, $au(BD)(=21)$, $au(BC)(=25)$, and $auub(BA)(=33)$, $auub(BD)(=33)$, $auub(BC)(=41)$. Since $au(BA)(=24.5 \geqslant 24)$ and $au(BC)(=25 \geqslant 24)$, itemsets (BA) and (BC) are HAUIs, and are directly output. If the *auub* value of an itemset is no less than $\delta \times TU$, for example, the itemset (BD), its AU-list will then used to generate extensions, to pursue the depth-first search using that itemset. This process is recursively performed until no AU-lists can be generated. After all itemsets with the prefix item (B) have been considered, the other 1-*HAUUBIs* (E, A, D, C) are processed in the same way. The final set of HAUIs obtained for the running example is shown in Table 8.

6. Experimental results

In this section, the performance of the proposed HAUI-Miner algorithm is compared with the three state-of-the-art algorithms, named HAUP-growth [15], PAI [10] and HAUI-tree [18] algorithms on several datasets. All algorithms were implemented in Java and experiments were carried on a computer having an Intel(R) Core (TM) i7-4790 3.60 GHz processor with 8 GB of main memory, running the 64 bit Microsoft Windows 7 operating system. Experiments were conducted on six datasets, including five real-world datasets [19] and one synthetic dataset generated using the IBM Quest Synthetic Data Generator [1]. A simulation model [12] was developed to generate quantities (internal utilities) and unit profit values (external utilities) of items in transactions for all datasets. External utilities have been generated in the [0.01, 10] interval using a log-normal distribution, and internal utilities have been randomly chosen in the [1,5] interval. Parameters used to describe the six datasets are shown in Table 9, and the characteristics of these datasets are shown in Table 10.

To assess the performance of each algorithm, the execution time, number of visited nodes in the search space, maximum memory usage, and the scalability were respectively analyzed. Results are reported below. In the performed experiments, if an algorithm ran for more than 10,000 s or if it ran out of memory, the algorithm was stopped.

6.1. Runtime

In this section, runtimes of the three state-of-the-art algorithms for mining HAUIs are compared with the proposed HAUI-Miner algorithm for various minimum average-utility threshold values, on the six datasets. Results are shown in Fig. 4.

It can be observed in Fig. 4 that the proposed HAUI-Miner algorithm outperforms previous algorithms for various minimum average-utility thresholds, on all six datasets. In particular, the proposed HAUI-Miner algorithm can be one to two orders of magnitude faster than the PAI, HAUP-growth and HAUI-tree algorithms. For example in Fig. 4(b), the runtime of HAUP-growth, PAI, and HAUI-Tree are respectively 233.7, 4.5 and 6.6 s, while the proposed algorithm only took 1.3 s when the minimum average-utility threshold was set to 4%. For the HAUP-growth algorithm, no results

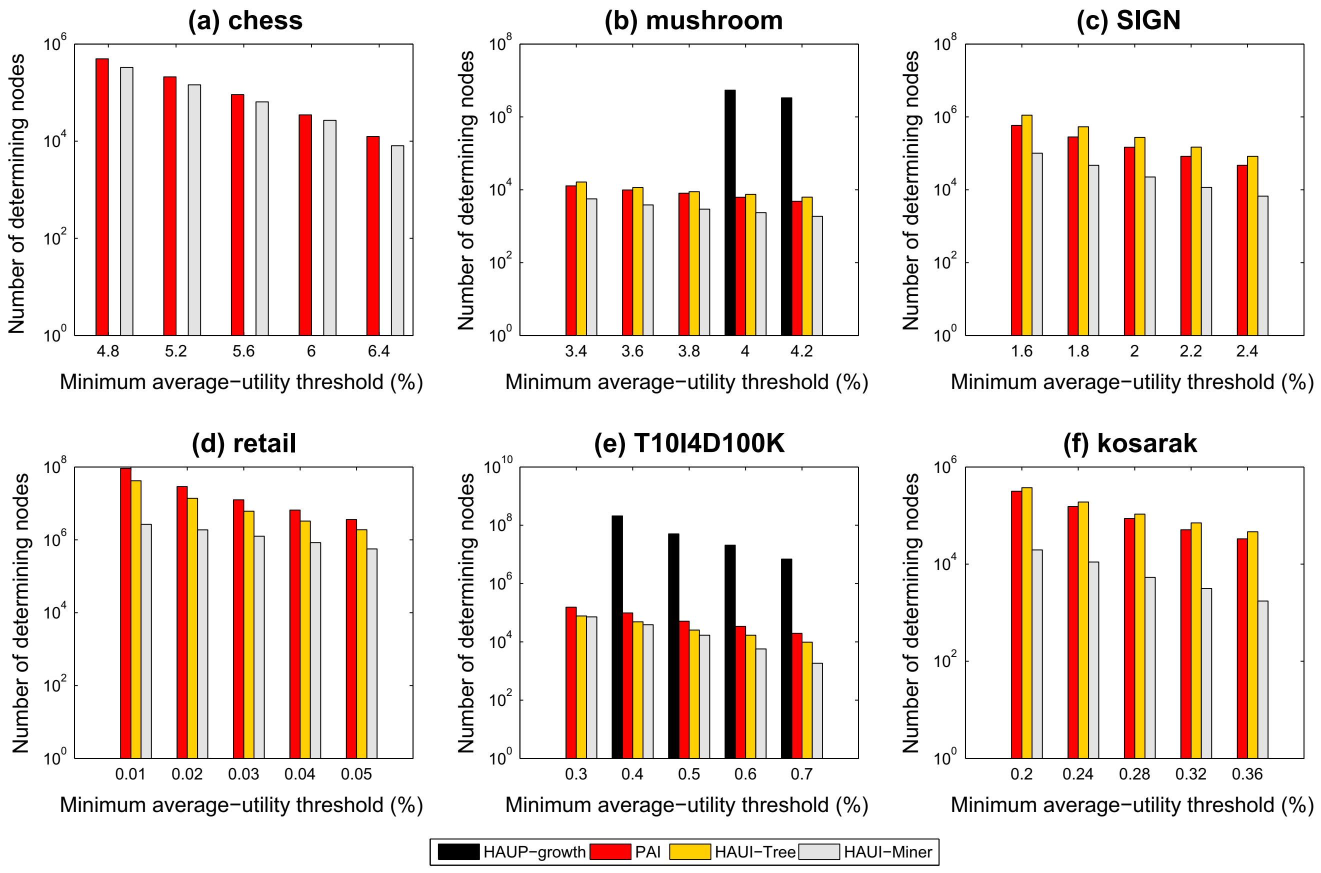


Fig. 5. Number of determining nodes for various minimum average-utility values.

are provided in Fig. 4(a), (c), (d) and (f). Moreover, the HAUP-growth algorithm has no results in Fig. 4(b) and (e) when the minimum average-utility threshold is respectively set to 3.8%, and 0.3% or below. The reason is that the HAUP-growth algorithm utilizes more memory to mine HAUIs based on its designed tree structure because it use additional arrays to maintain the information to be used by the mining process. If the array size is *m*, HAUP-growth generates up to 2^m candidates for mining HAUIs. The performance of the HAUI-tree algorithm can be explained in the same way.

The PAI algorithm uses an iterative mining method to discover HAUIs and performs an additional database scan to determine the actual HAUIs from the set of promising candidates. This approach is more efficient than that of the HAUP-growth and HAUI-tree algorithm, and thus it is faster. Another observation is that the gap between the designed HAUI-Miner algorithm and PAI is smaller when the minimum average-utility thresholds is set to large values, as well as with the other compared algorithms. This is also reasonable since when the minimum average-utility threshold is set higher, fewer candidates are generated and it is more easier to discover the actual HAUIs from a small set of candidates.

6.2. Node analysis

In this section, the number of nodes generated for discovering the actual HAUIs using each algorithm is compared. Results are shown in Fig. 5.

It can be observed in Fig. 5 that the number of nodes generated by the proposed HAUI-Miner algorithm is much less than the HAUP-growth, PAI and HAUI-Tree algorithms for various minimum average-utility threshold values on all datasets. This is because the compared algorithms are all sensitive to transaction length. This is especially the case for the HAUP-growth algorithm since an extra array is attached to each node of its tree structure to keep information to be used by the mining process. When more information is stored in these arrays, the number of nodes (candidates) generated exponentially increases. No results are provided for the HAUP-growth algorithm in Fig. 5(a), (c), (d) and (e) since it exceeds the setup maximum time limit. In the other figures such as Fig. 5(b) and (e), the HAUP-growth algorithm generates a large amount of candidates to obtain the HAUIs. The PAI and HAUI-tree algorithms both use a projection mechanism to generate smaller datasets for the later mining process. Thus, the number of itemsets considered as candidates for mining the HAUIs is less than using the HAUP-growth algorithm, as shown in Fig. 5(b)–(e). Based on the designed pruning strategy, the number of determining nodes is greatly reduced, as it can be observed in Fig. 5.

6.3. Memory usage

The memory usage of the algorithms was also compared when varying the minimum average-utility thresholds, for all datasets. Results are shown in Fig. 6.

It can be observed that the proposed HAUI-Miner algorithm performs well in terms of memory usage. The proposed HAUI-Miner algorithm needs less more memory than PAI. The reason is that PAI only uses its projection mechanism to find HAUIs, while the designed HAUI-Miner algorithm constructs the AU-lists for storing the necessary information. The HAUP-growth and the HAUI-tree algorithms both requires more memory to generate their tree structures for keeping the information needed to discover HAUIs. If the depth of the constructed tree is high, the amount of memory required by the HAUP-growth algorithm

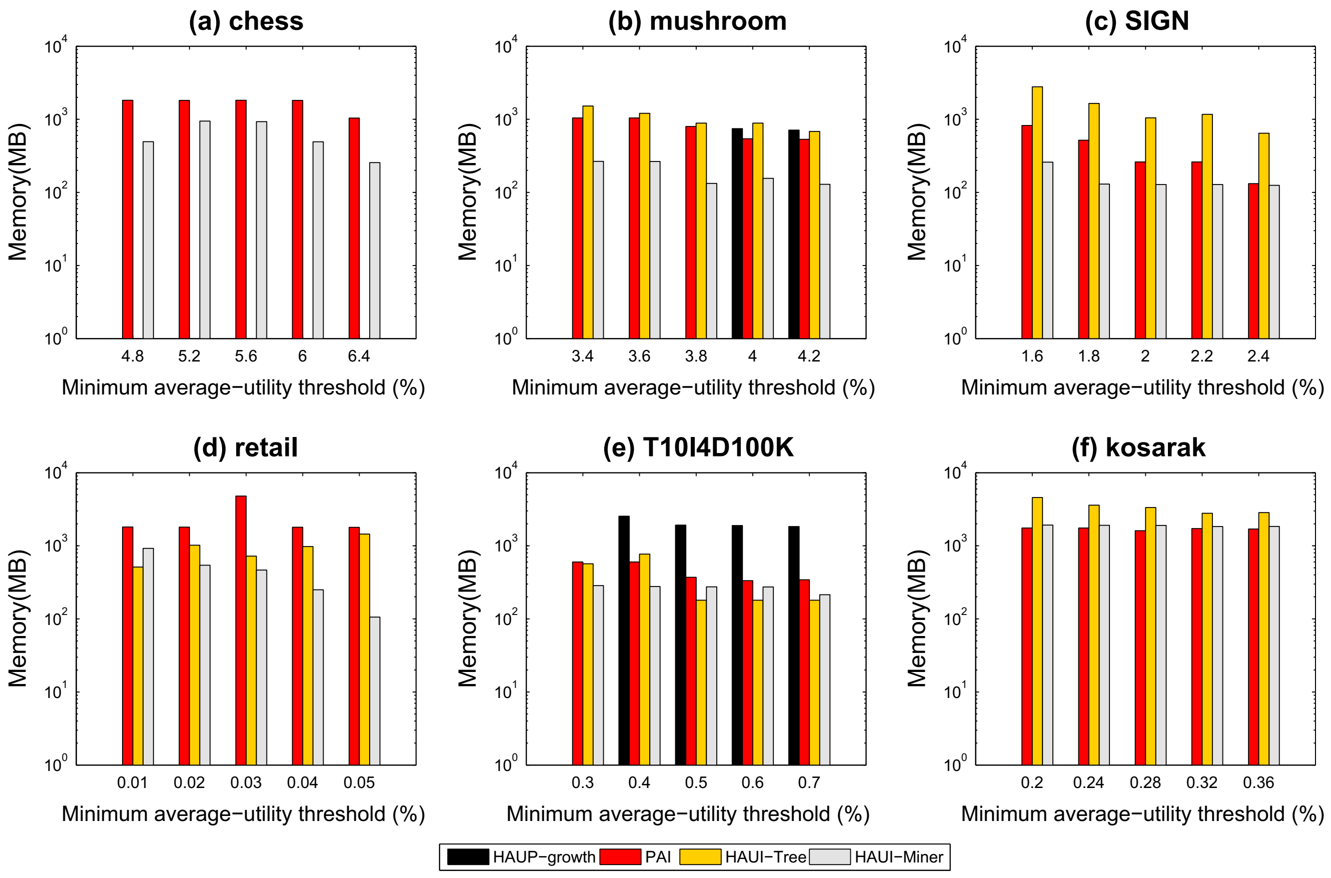


Fig. 6. Memory consumption for various minimum average-utility threshold values.

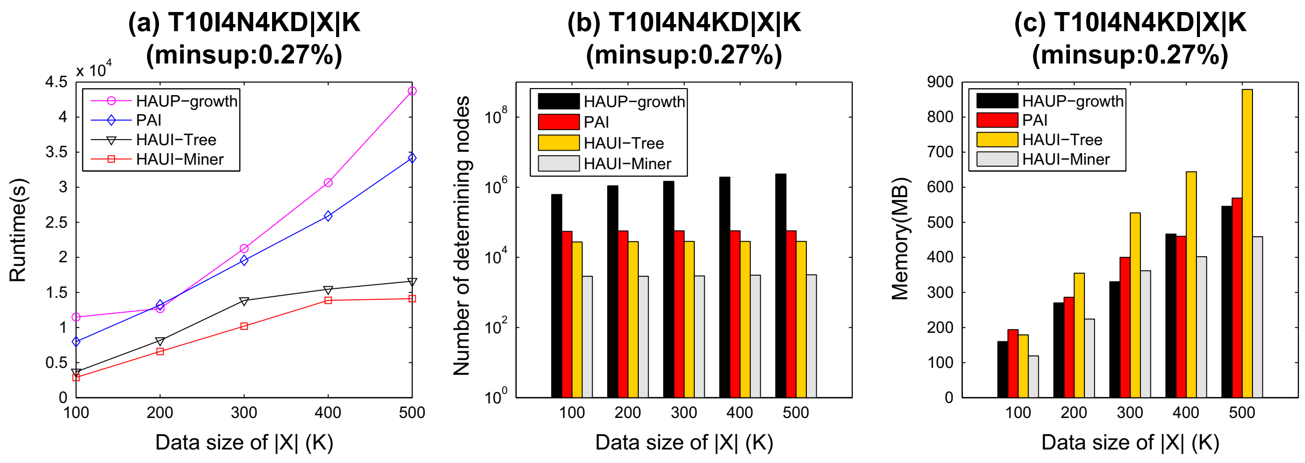


Fig. 7. Scalability for various dataset sizes.

greatly increases since an array is attached to each node, as it can be observed in Fig. 6(b) and (e). Therefore, it can be concluded that HAUI-Miner has good performance on memory usage, and requires less memory to mine the actual HAUIs in all datasets.

6.4. Scalability

The scalability of the proposed HAUI-Miner algorithm was also compared with the same three state-of-the-art HAUI mining algorithms on a series of synthetic datasets T10I4N4KD|X|K, where the number of transactions was varied from 100k to 500k transactions using a 100k increment. The minimum average-utility threshold was fixed at 0.27%. Results are shown in Fig. 7.

It can be observed in Fig. 7, that all compared algorithms have longer runtimes and require more memory to find HAUIs when the dataset size increases. Although the HAUI-tree algorithm can efficiently reduce the number of candidates for mining HAUIs using its tree structure, the information kept in the HAUI-tree is large and thus the memory usage considerably increases with the dataset size. The designed HAUI-Miner algorithm utilizes the proposed AU-list structure for storing the information needed to discover HAUIs. The pruning strategy used in the designed HAUI-Miner algorithm effectively prunes unpromising candidates, while ensuring the correctness of the designed algorithm. It can also be found that all compared algorithms and the designed HAUI-Miner have good scalability, with the exception of the HAUI-tree algorithm when the dataset is very large. Nonetheless, the proposed algorithm outperforms the other algorithms in terms of runtime, number of determining nodes, memory usage and scalability. The effectiveness and efficiency of the proposed HAUI-Miner algorithm is thus quite acceptable for real-world applications.

7. Conclusion and future work

Traditional high-utility itemset mining (HUIM) considers purchase quantities and unit profits of items to discover high-utility itemsets (HUIs). Because the utility of larger itemset is generally greater than the utility of smaller itemset, traditional HUIM algorithms tend to be biased toward finding large itemsets. Thus, the traditional utility measure is not a fair measurement in real-world applications. To address this issue, the problem of high average-utility itemset mining (HAUIM) has been proposed. HAUIM has attracted a lot of attention since it provides a useful alternative interestingness measure to evaluate the discovered patterns. In this paper, an efficient average-utility (AU)-list structure is designed to store the information needed to discover HAUIs. The HAUI-Miner algorithm discovers HAUIs by exploring a set-enumeration tree using a depth-first search. An efficient pruning strategy is also developed to prune unpromising candidates early and thus reduce the search space. Substantial experiments were conducted on both real-life and synthetic datasets to evaluate the efficiency and effectiveness of the designed algorithm in terms of runtime, number of determining nodes, memory consumption usage, and scalability. Performance was compared with the state-of-the-art HAUP-growth, PAI and HAUI-Tree algorithms.

In this paper, the HAUI-Miner algorithm was designed to discover HAUIs efficiently in a static database. However, in real-life situations, transactions are frequently updated. New transactions may be frequently added to the original database. In future work, we will thus consider developing several algorithms to mine HAUIs in incremental databases and in data streams. Besides, with the rapid growth of information technology, it is also a critical issue to mine HAUIs in big data.

Acknowledgment

This research was partially supported by National Natural Science Foundation of China (NSFC) under Grant No. 61503092.

References

[1] R. Agrawal, R. Srikant, Quest synthetic data generator <http://www.Almaden.ibm.com/cs/quest/syndata.html>1994.
[2] R. Agrawal, T. Imielinski, A. Swami, Mining association rules between sets of items in large databases, in: ACM SIGMOD Record, 1993, pp. 207–216.
[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: The International Conference on Very Large Databases, 1994, pp. 487–499.
[4] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, Y.K. Lee, Efficient tree structures for high utility pattern mining in incremental databases, IEEE Trans. Knowl. Data Eng. 21 (2009) 1708–1721.
[5] R. Chan, Q. Yang, Y.D. Shen, Minging high utility itemsets, in: IEEE International Conference on Data Mining, 2003, pp. 19–26.
[6] M.S. Chen, J. Han, P.S. Yu, Data mining: an overview from a database perspective, IEEE Trans. Knowl. Data Eng. 8 (6) (1996) 866–883.
[7] P. Fournier-Viger, C.W. Wu, S. Zida, V.S. Tseng, FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning, Foundations of Intelligent Systems, vol. 8502, 2014, pp. 83–92.
[8] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, Data Min. Knowl. Discov. 8 (2004) 53–87.
[9] T.P. Hong, C.H. Lee, S.L. Wang, Mining high average-utility itemsets, in: IEEE International Conference on Systems, Man and Cybernetics, 2009, pp. 2526–2530.

[10] G.C. Lan, T.P. Hong, V.S. Tseng, Efficiently mining high average-utility itemsets with an improved upper-bound strategy, Int. J. Inform. Technol. Decis. Making 11 (2012) 1009–1030.
[11] G.C. Lan, T.P. Hong, V.S. Tseng, An efficient projection-based indexing approach for mining high utility itemsets, Knowl. Inform. Syst. 38 (1) (2013) 85–107.
[12] Y. Liu, W.K. Liao, A. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, Lect. Notes Comput. Sci. 3518 (2005) 689–695.
[13] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: ACM International Conference on Information and Knowledge Management, 2012, pp. 55–64.
[14] C.W. Lin, T.P. Hong, W.H. Lu, The pre-fufp algorithm for incremental mining, Expert Syst. Appl. 36 (2009) 9498–9505.
[15] C.W. Lin, T.P. Hong, W.H. Lu, Efficiently mining high average utility itemsets with a tree structure, Lect. Notes Comput. Sci. 5990 (2010) 131–139.
[16] C.W. Lin, T.P. Hong, W.H. Lu, An effective tree structure for mining high utility itemsets, Expert Syst. Appl. 38 (2011) 7419–7424.
[17] Jerry C.W. Lin, W. Gan, T.P. Hong, Vincent S. Tseng, Efficient algorithms for mining up-to-date high-utility patterns, Adv. Eng. Inform. 29 (2015) 648–661.
[18] T. Lu, B. Vo, H.T. Nguyen, T.P. Hong, A new method for mining high average utility itemsets, Lect. Notes Comput. Sci. 8838 (2014) 33–42.
[19] SPMF: An Open-Source Data Mining Library. <http://www.philippe-fournier-viger.com/spmf/>.
[20] V.S. Tseng, C.W. Wu, B.E. Shie, P.S. Yu, UP-growth: an efficient algorithm for high utility itemset mining, in: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2010, pp. 253–262.
[21] V.S. Tseng, B.E. Shie, C.W. Wu, P.S. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, IEEE Trans. Knowl. Data Eng. 25 (2013) 1772–1786.
[22] H. Yao, H.J. Hamilton, C.J. Butz, A foundational approach to mining itemset utilities from databases, in: SIAM International Conference on Data Mining, 2004, pp. 215–221.
[23] M. Zihayat, A. An, Mining top-k high utility patterns over data streams, Inform. Sci. 285 (2014) 138–161.