IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 1, 2019, accepted September 3, 2019, date of publication October 7, 2019,
date of current version October 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2945840

An Efficient Tree-Based Algorithm for Mining High Average-Utility Itemset

IRFAN YILDIRIM[1,2] AND METE CELIK[1], (Member, IEEE)
[1]Department of Computer Engineering, Erciyes University, 38030 Kayseri, Turkey
[2]Department of Computer Engineering, Erzurum Technical University, 25050 Erzurum, Turkey

Corresponding author: Irfan Yildirim (irfanyildirim@erciyes.edu.tr)

ABSTRACT High-utility itemset mining (HUIM), which is an extension of well-known frequent itemset mining (FIM), has become a key topic in recent years. HUIM aims to find a complete set of itemsets having high utilities in a given dataset. High average-utility itemset mining (HAUIM) is a variation of traditional HUIM. HAUIM provides an alternative measurement named the average-utility to discover the itemsets by taking into consideration both of the utility values and lengths of itemsets. HAUIM is important for several application domains, such as, business applications, medical data analysis, mobile commerce, streaming data analysis, etc. In the literature, several algorithms have been proposed by introducing their own upper-bound models and data structures to discover high average utility itemsets (HAUIs) in a given database. However, they require long execution times and large memory consumption to handle the problem. To overcome these limitations, this paper, first, introduces four novel upper-bounds along with pruning strategies and two data structures. Then, it proposes a pattern growth approach called the HAUL-Growth algorithm for efficiently mining of HAUIs using the proposed upper-bounds and data structures. Experimental results show that the proposed HAUL-Growth algorithm significantly outperforms the state-of-the-art dHAUIM and TUB-HAUIM algorithms in terms of execution times, number of join operations, memory consumption, and scalability.

INDEX TERMS Average utility, high average utility itemset, tighter upper bounds, utility mining, pruning strategy.

I. INTRODUCTION

Frequent itemset mining (FIM), which is one of the most well-known techniques to discover relations among items in large data, was originally introduced to discover frequently purchased itemsets by customers [1]–[4]. Basically, the goal of a FIM algorithm is to find a complete set of itemsets whose frequencies (or supports) higher than or equal to a predefined threshold in a given transactional database. However, FIM assumes that databases contain only binary information and all items in the database have same importance. In other words, the non-binary attributes (i.e., weight importance) of the items are not important. Therefore, the result of FIM is not always sufficient (or meaningful) for different real-world applications when the quantities and profits of items are considered. For example, discovered itemsets by FIM may be unimportant from the business perspective since cheap products are more likely to be frequently purchased, but they are less profitable.

The problem of high-utility itemset mining (HUIM) [5], [6] was introduced as an extension of FIM to discover more meaningful itemsets by taking into account non-binary attributes of items. In the HUIM, both of purchased unit quantities (internal utilities) and unit profits (external utilities) of items are taken into account to determine how important (or profitable) an itemset is. The utility of an itemset is obtained by collecting the utility value of each item it contains from the transactions that involve itself. The result of HUIM includes only those itemsets whose utilities are not less than a minimum high-utility threshold. Such itemsets are called as high-utility itemsets (HUIs). However, as the length of the itemset increases, its utility tends to be larger since the utility of an itemset is the sum of the utiliy of each item that it contains. Therefore, HUIM mainly suffers from generating a large number of itemsets with long lengths. In addition, because of the nature of the utility measurement in HUIM,

The associate editor coordinating the review of this manuscript and approving it for publication was Senthil Kumar.

most of the discovered HUIs may contain items with low utilities.

To address these limitations, the problem of high average-utility mining (HAUIM) is then introduced with a more fair measurement named average-utility [7]. The average-utility of an itemset is derived by dividing its utility to the number of its items. An itemset is considered as a high average-utility itemset (HAUI) if its average-utility value is no less than a given minimum utility threshold (*minUtil*).

HAUIM is important for several application domains, such as, business applications [8]–[10], medical data analysis [11], [12], streaming (or incremental) data analysis [13]–[15], etc. In business applications, HAUIM can be used for cross-marketing, shelf management, and new promotions techniques to increase sales of high profitable itemsets [8]. In medical data analysis, HAUIM can be used to discover rare but critical combinations of symptoms and to find gene patterns by taking into account gene importance and its degrees of expression [11]. In streaming data analysis, HAUIM can be used to analyze streaming datasets (such as, wireless sensor, web click, etc.) by taking into account the characteristics of streaming data [13].

A typical HAUIM approach aims to find a complete set of HAUIs based on a given *minUtil* threshold. This process is computationally complex due to anti-monotonic characteristic of average-utilities of itemsets. The first proposed algorithm to mine HAUIs is the Two-phase high average-utility pattern mining (TPAU) algorithm [7]. Since then, several algorithms, such as PAI [16], HAUP-Growth [17], HAUI-Tree [18], HAUI-Miner, [19], FHAUM [20], MHAI [21], EHAUPM [9], TUB-HAUPM [10], and dHAUIM [22] were proposed to solve the HAUIM problem more efficiently. However, all the existing HAUIM algorithms need long execution times and large amounts of memory to perform their mining tasks, especially when the database size is large or the minimum utility threshold is low.

Therefore, in order to enhance the efficiency of solving the problem of mining HAUIs, efficient strategies should be developed, such as (1) introducing more effective upper-bounds and pruning strategies for early pruning unpromising itemsets from the search space, (2) proposing efficient data structures for reducing the memory consumption and the cost of database scans in addition to avoid the costly join operations, and (3) developing an effective mining method to discover the complete and correct set of HAUIs by utilizing all the strategies mentioned above together.

This study proposes an algorithm named as High Average-Utility List-Growth (HAUL-Growth) algorithm for mining HAUIs efficiently. The main contributions of this paper can be listed as follows:

- Four new upper-bounds, which are named as extension upper-bound (*eub*), tighter extension upper-bound (*teub*), bi-directional tighter extension upper-bound (*bteub*), and maximum remaining k-items extension upper-bound (*max-reub*$_k$), are proposed for pruning extensions of itemsets while ensuring that all HAUIs are discovered. The *eub* and *teub* are designed to prune the single-item extensions of itemsets. The *bteub* and *max-reub*$_k$ are designed to prune the 2-items extensions of itemsets. Among them, *eub* is calculated by utilizing the initial database while the others are calculated by utilizing the projected database obtained based on a total processing order on items.
- A compact tree data structure, which is named HAUL-Tree (High Average-Utility List-Tree), is proposed to reduce the cost of database scans and avoid the costly join operations. The HAUL-Tree is a compact representation of the given database and is designed to store the required information of mining HAUIs based on a given minimum utility threshold, *minUtil*.
- A list-based data structure, which is named Information List (IL), is also proposed. The IL of an itemset can be easily obtained from the proposed HAUL-Tree structure, which is designed to determine which single-item extensions of the itemset are HAUIs and prune its 2-items extensions.
- A pattern growth approach is proposed for efficiently mining of HAUIs based on a divide and conquer strategy by utilizing proposed upper-bounds and data structures.
- Experiments are conducted with several datasets. Experimental results show that the proposed HAUL-Growth algorithm outperforms the previous state-of-the-art HAUIM algorithms, such as dHAUIM and TUB-HAUPM, in terms of execution times, number of join operations, memory usage, and scalability.

The rest of the paper is organized as follows. Section II gives the related work. Section III presents basic concepts of the problem of HAUIM. Section IV introduces the proposed upper-bounds and pruning strategies. Section V presents the details of the proposed data structures. The mining procedure of the proposed algorithm is explained in Section VI. Experimental evaluation of the proposed algorithm is given in Section VII. Section VIII presents conclusion and future works.

II. RELATED WORK

Many studies have been proposed for HUIM since the utility model [5] is introduced by considering quantities (i.e., internal utilities) and profits (i.e., external utilities) of items. Two-Phase [6] algorithm is proposed with an upper-bound strategy, called the transaction-weighted utility (TWU) model, by adopting the downward closure (DC) to prune the unpromising itemsets early and thus reduce to search space. Although Two-Phase algorithm reduces the search space, it still generates a large number of candidates. Then, in order to reduce the number of candidates, several algorithms, such as HUP-Tree [23], UP-Growth [24], and UP-Growth+ [8], have been introduced. Among them, UP-Growth+ is reported as the fastest algorithm compared to the others [8]. All these algorithms are designed to discover HUIs in two phases. In phase 1, they generates candidate HUIs by overestimating itemsets' utilities. Then, in phase 2, they

calculate the actual utility of each candidate HUI to obtain the correct HUIs by performing a database scan.

To avoid computationally complex candidate generation procedure of the above mentioned algorithms, in the literature, algorithms which discovers HUIs in a single phase were introduced. These algorithms can be listed as HUI-Miner [25], FHM [26], d2HUP [27], HUP-Miner [28], EFIM [29], IMHUP [30], and mHUIMiner [31]. They use their additional strategies to prune the search space more efficiently and most of them are designed based on utility-list structure [25]. Although HUIM can discover more useful itemsets compared to FIM, it suffers from generating too many itemsets with long lengths. The reason is that when the size of itemsets gets increase, their utilities tend to be increased.

In order to avoid this problem, the problem of high average-utility itemset mining (HAUIM) is then introduced with a more fair utility measurement called average-utility by considering the length of itemsets. Most of the algorithms used in HUIM is modified or extended to mine HAUIs. The first proposed algorithm is TPAU [7]. It mines HAUIs in two phases. It uses the average-utility upper-bound (*auub*) property by adopting the DC to prune search space. However, it suffers from generating of numerous candidate itemsets with multiple database scans. To speed up the mining process of HAUIMs several algorithms are proposed. A projection based algorithm, PAI [16], is proposed with a pruning strategy. A tree based pattern growth approach, HAUP-Growth [17], is proposed with a tree data structure named as HAUP-Tree to avoid of multiple database scans that TPAU suffers from. HAUI-Tree [18] is proposed to solve HUIM by utilizing an index table. Then, HAUI-Miner [19] is proposed to efficiently mine HAUIs using a compact list structure called AU-List. Although, HAUI-Miner is more efficient than previous works, it is still computationally complex in terms of join operations [9].

Moreover, other HAUIM algoritms, such as FHAUM [20], MHAI [21], EHAUPM [9], TUB-HAUPM [10], and dHAUIM [22] are introduced with their upper-bounds, pruning strategies, and data structures to speed up the discovery process of HAUIs. Among them, FHAUM [20], MHAI [21], and EHAUPM [9] perform the mining process by constructing their list-based data structures. These algorithms determine promising 1-itemset based on the *auub* model. The FHAUM [20] adopts the AU-List structure [19] to store the required information for mining HAUIs. It uses a matrix which stores the *auub* of 2-itemsets and two upper-bounds called as *lubau* and *tubau* to prune the search space. The MHAI [21] uses a list-based data structure named HAI-List (High average-utility itemset list) to mine HAUIs and an upper-bound called as *mau* to prune the search space. The EHAUPM [9] uses a modified average-utility (MAU)-list structure which is designed to store the remaining maximal and revised transaction-maximum utilities of transactions. Similar to FHAUM, it has a matrix structure named as EAUCM for pruning k-itemsets ($k \geq 2$). The upper-bounds that EHAUPM is used to prune the search space are named as *lub* and *rtub*. In addition, EHAUPM uses an additional strategy named SUJ (Stop Unpromising Join Operations) to avoid unnecessary join operations to reduce the cost of mining HAUIs.

The set of promising 1-itemsets is also determined based on *auub* model by TUB-HAUPM [10]. However, for further reduction of the number of promising 1-itemsets, TUB-HAUPM performs multiple database scans. In each database scan, it re-calculates the *auub* values of items after removing unpromising items that obtained by the previous database scan. This process continues until there is no unpromising items are obtained. Afterwards, it examines the search space by utilizing two upper-bounds called as *mfuub* and *krtmuub*. Since these upper-bounds are not comparable, TUB-HAUPM accumulates both of them for each itemset from the related transactions and selects the one which has lower value to check against to the given minimum utility threshold for pruning the search space. However, TUB-HAUPM does not have a data structure. It calculates average-utility and upper-bounds for each candidate itemset by performing an additional database scan.

To the best of our knowledge, the most recent HAUIM algorithm is dHAUIM [22]. The dHAUIM, first, converts the given database into an integrated quantitative matrix Q. Afterwards, by utilizng the Q, it obtains a utility vector that stores all column utility values related to the item. In addition, the set of row indices for each item are also determined. Moreover, it uses a *diffset* technique to obtain the utility vectors of itemsets during the examination of the search space. To prune the search space, dHAUIM uses four upper-bounds which are $\overline{aub_1}$, $\overline{aub}$, $\overline{iaub}$, and $\overline{laub}$.

However, solving the problem of HAUIM by using the state-of-the-art HAUIM algorithms is still very time-consuming. The FHAUM [20], MHAI [21], and EHAUPM [9] mainly suffer from the computationally expensive join operations for generating a large number of list structures during the mining phase. The TUB-HAUPM [10] mainly suffers from numerous database scans to examine the search space since it requires an additional database scan to compute the utility and upper-bounds of each promising itemset. The dHAUIM mainly suffers from the cost of obtaining the utility vectors of itemsets since it is necessary to calculate all the required column utilities to obtain the utility vector of an itemset. More importantly, the upper-bounds used by these algorithms for pruning the search space are not tight enough, and so they suffer from generating a large number of unpromising itemsets. Because of these reasons, the state-of-the-art HAUIM algorithms do not perform well, especially when the number of transactions and/or unique items contained in a database is too large.

III. BASIC CONCEPT

In this section, definitions related to HAUIM are presented based on the previous HAUIM studies.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n unique items. A database is denoted by $DB = \{T_1, T_2, \ldots, T_m\}$. A transaction T_j, where $T_j \in DB$ and $0 < j \leq m$, is a subset of itemset I. An itemset $X = \{i_1, i_2, \ldots, i_k\}$ is a set of k unique items, where $X \subseteq I$. In the database DB, each item in a transaction has two utility values which are an internal utility (i.e., unit quantities) and an external utility (i.e., unit profits).

TABLE 1. A sample database, *DB*.

TID	Transactions with Internal Utilities
T_1	$(a:2)$ $(c:1)$ $(d:1)$ $(f:7)$ $(h:2)$
T_2	$(d:1)$ $(e:4)$ $(g:2)$
T_3	$(a:1)$ $(g:2)$
T_4	$(b:3)$ $(c:2)$ $(d:5)$ $(e:1)$ $(f:6)$
T_5	$(a:1)$ $(c:3)$ $(d:1)$ $(e:2)$
T_6	$(a:1)$ $(b:1)$ $(c:1)$ $(d:1)$ $(f:3)$

A sample database given in Table 1 is used to explain the definitions. In the database, there are six transactions with eight different items (a, b, c, d, e, f, g, and h). Each transaction contains multiple items with their internal utilities. For example, the internal utility of item a in transaction T_1 is denoted as $iu(a, T_1)$ and equals to 2. The external utilities of the items are given in Table 2. For example, the external utility of item a is denoted as $eu(a)$ and equals to 5.

TABLE 2. External utilities of items of the sample database.

Item	External Utility
a	5
b	10
c	3
d	8
e	4
f	1
g	7
h	15

Definition 1 (Utility and Average-Utility of an Itemset in a Transaction): The utility and average-utility of an itemset X in a transaction T_j are denoted as $u(X, T_j)$ and $au(X, T_j)$, respectively. They are defined as:

$$u(X, T_j) = \sum_{i \in X \wedge X \subseteq T_j} iu(i, T_j) \times eu(i), \tag{1}$$

$$au(X, T_j) = \frac{u(X, T_j)}{|X|}. \tag{2}$$

For example, let $X = \{a, c\}$. Therefore, $u(\{a, c\}, T_1) = (iu(a, T_1) \times eu(a)) + (iu(c, T_1) \times eu(c)) = (2 \times 5) + (1 \times 3) = 13$. Besides, $au(\{a, c\}, T_1)$ is calculated as $u(\{a, c\}, T_1) / (|\{a, c\}|) = 13/2 = 6.5$.

Definition 2 (Utility and Average-Utility of an Itemset in a Database): The utility and average-utility of an itemset X in a database DB are denoted as $u(X, DB)$ and $au(X, DB)$, respectively. They are defined as:

$$u(X, DB) = \sum_{X \subseteq T_j \wedge T_j \in DB} u(X, T_j), \tag{3}$$

$$au(X, DB) = \frac{u(X, DB)}{|X|}. \tag{4}$$

For example, both of items a and c appear together in transactions T_1, T_5, and T_6. Therefore, $u(\{a, c\}, DB)$ and $au(\{a, c\}, DB)$ are calculated as $u(\{a, c\}, T_1) + u(\{a, c\}, T_5) + u(\{a, c\}, T_6) = 13 + 14 + 8 = 35$ and $u(\{a, c\}, DB) / (|\{a, c\}|) = 35/2 = 17.5$, respectively.

Definition 3 (Transaction Utility and Total Utility of a Database): The transaction utility of a transaction T_j and total utility of a database DB are denoted as $u(T_j)$ and $u(DB)$, respectively. They are defined as:

$$u(T_j) = \sum_{i_k \in I} u(i_k, T_j), \tag{5}$$

$$tu(DB) = \sum_{T_j \in DB} u(T_j). \tag{6}$$

For example, $u(T_1) = u(a, T_1) + u(c, T_1) + u(d, T_1) + u(f, T_5) + u(h, T_1) = 10 + 3 + 8 + 7 + 30 = 58$. Similarly, $u(T_2) = 38$, $u(T_3) = 19$, $u(T_4) = 86$, $u(T_5) = 30$, and $u(T_6) = 29$. Therefore, $tu(DB) = u(T_1) + u(T_2) + u(T_3) + u(T_4) + u(T_5) + u(T_6) = 260$.

Definition 4 (High Average-Utility Itemset): An itemset X is called as high average-utility itemset (HAUI) if its average-utility $au(X, DB)$ is not lower than a minimum threshold *minUtil* given by the user.

For example, consider the *minUtil* is set to 10% of $tu(DB) = 260$, which is 26. Therefore, itemset $\{a, c\}$ is not a HAUI since $au(\{a, c\}, DB) = 17.5 < 26$.

Note that, the average-utility of an itemset may be higher or lower than the average-utility of any of its subsets or supersets in the problem of HAUIM. So, to obtain a DC property in HAUIM, an anti-monotonic measure called the average utility upper bound (*auub*) is used in the previous studies [7], [17]. The following definitions are related to *auub*.

Definition 5 (Maximum Utility of a Transaction): The maximum utility of a transaction T_j is denoted as $mu(T_j)$ and defined as:

$$mu(T_j) = max\{u(i_k, T_j) | i_k \subseteq T_j\}. \tag{7}$$

For example, $mu(T_1) = max(u(a, T_1), u(c, T_1), u(d, T_1), u(f, T_1), u(h, T_1)) = max(10, 3, 8, 7, 30) = 30$.

Definition 6 (Average-Utility Upper-Bound): The average-utility upper-bound (*auub*) of an itemset X is denoted as $auub(X)$ and defined as:

$$auub(X) = \sum_{X \subseteq T_j \wedge T_j \in DB} mu(T_j). \tag{8}$$

For example, $auub(\{a, c\}) = mu(T_1) + mu(T_5) + mu(T_6) = 30 + 9 + 10 = 49$.

Definition 7 (Downward Closure Property of auub): If an itemset X has an *auub* value lower than *minUtil* then all supersets of X have *auub* values lower than *minUtil* [7]. Therefore, this downward closure property of *auub* can be used to prune search space while mining HAUIs.

For example, the $auub(\{d, g\}) = mu(T_2) = 16 <$ *minUtil* $= 26$. Therefore, none of the extensions of $\{d, g\}$ can be HAUI. There is no need to examine them.

IV. PROPOSED UPPER-BOUNDS

To prune the search space efficiently, this study introduces four new upper-bounds named as extension upper-bound (*eub*), tighter extension upper-bound (*teub*), bi-directional tighter extension upper-bound (*bteub*), and maximum remaining k-items extension upper-bound (*max-reub*$_k$). All of them are designed to determine whether extensions of itemsets can be pruned. In other words, they are proposed to prune the extensions of itemsets while ensuring that all HAUIs are found. Below, *eub*, *teub*, *bteub*, and *max-reub*$_k$ with their pruning strategies are mentioned. Lastly, their effectiveness are discussed by comparing the existing upper-bounds of the literature.

A. EXTENSION UPPER-BOUND (EUB)

The extension upper-bound (*eub*) is designed to determine itemsets whose extensions do not promise to contain any HAUI. Details are given below.

Definition 8 (Sum of the Largest n Utilities in a Transaction): The sum of the largest n utilities in a transaction T_j is denoted as $SL_n(T_j)$ and calculated by summing up the largest n utilities in T_j.

For example, transaction T_2 consists of items d, e, and g, where $u(d, T_2)$, $u(e, T_2)$, and $u(g, T_2)$ are 8, 16, and 14, respectively. Since the largest 2 utilities in T_2 are 16 and 14, $SL_2(T_2) = 16 + 14 = 30$.

Definition 9 (Extension Upper-Bound of an Itemset in a Transaction): The extension upper-bound (*eub*) of an itemset X in a transaction T_j, where $X \in T_j$, is denoted as $eub(X, T_j)$ and defined as:

$$eub(X, T_j) = \begin{cases} \dfrac{SL_{|X|+1}(T_j)}{|X|+1}, & \text{if } |T_j| \geq |X|+1 \\ 0, & \text{otherwise,} \end{cases} \tag{9}$$

where $|X|$ is the number of items that X has.

For example, $eub(d, T_2) = eub(e, T_2) = eub(g, T_2) = SL_2(T_2)/2 = 30/2 = 15$, $eub(\{d, e\}, T_2) = eub(\{d, g\}, T_2) = eub(\{e, g\}, T_2) = SL_3(T_2)/3 = 38/3 = 12.67$, and $eub(\{d, e, g\}, T_2) = 0$.

Definition 10 (Extension Upper-Bound of an Itemset in a Database): The extension upper-bound of an itemset X in a database DB is denoted as $eub(X, DB)$ and defined as:

$$eub(X, DB) = \sum_{X \subseteq T_j \wedge T_j \in DB} eub(X, T_j). \tag{10}$$

For example, itemset $\{d, e\}$ is seen in transactions T_2, T_4, and T_5. Therefore, $eub(\{d, e\}, DB) = eub(\{d, e\}, T_2) + eub(\{d, e\}, T_4) + eub(\{d, e\}, T_5) = ((16+14+8)/3) + ((40+30+6)/3) + ((9+8+8)/3) = 46.33$

Definition 11 (Extensions Promising Itemset (EPI)): If $eub(X) \geq minUtil$ is obtained, then itemset X is called as an extension promising itemset (EPI).

For example, $\{d, e\}$ is an EPI since $eub(\{d, e\}) = 46.33 \geq minUtil = 26$. In addition, $\{d, e\}$ can also be called as 2-EPI since $|\{d, e\}| = 2$.

For example, $eub(a)$, $eub(b)$, $eub(c)$, $eub(d)$, $eub(e)$, $eub(f)$, $eub(g)$, and $eub(h)$ are obtained as 47, 44, 72.5, 87.5, 58.5, 64, 24.5, and 20, respectively. Therefore, the set of 1-EPIs is obtained as $\{a, b, c, d, e, f\}$ since items g and h have *eub* values lower than $minUtil = 26$.

Theorem 1: None of the extensions of an itemset X has average-utility value greater than $eub(X)$.

Proof: Let Y be any extension of an itemset X, and $Tids(Y)$ and $Tids(X)$ be the sets of transactions including Y and X, respectively. Since, $u(Y, T_j)$ can be at most as $SL_{|Y|}(T_j)$ for each transaction $T_j \in Tids(Y)$, it is clear that $au(Y, T_j) \leq \dfrac{SL_{|Y|}(T_j)}{|Y|} \leq \dfrac{SL_{|X|+1}(T_j)}{|X|+1} = eub(X, T_j)$ holds for each $T_j \in Tids(Y)$. Moreover, $Tids(Y) \subseteq Tids(X)$ is also true, and so Theorem 1 is correct. ■

Pruning Strategy 1 (Pruning the Search Space Based on eub): Based on the Theorem 1, if $eub(X) < minUtil$, then none of extensions of the itemset X can be a HAUI. In other words, if an itemset is not an EPI, then its extensions are not HAUIs. Thus, there is no need to examine the extensions of itemsets if they are not EPIs.

For example, when $minUtil = 26$, items g and h are not EPIs, and so their extensions cannot contain any HAUI. Therefore, there is no need to examine the extensions of items g and h.

B. TIGHTER EXTENSION UPPER-BOUND (TEUB)

The search space of the problem can be represented as an enumeration tree based on a total processing order on items as mentioned in [9], [29]. This allows a tighter upper bound to be developed for reducing the search space more efficiently. For this reason, the second upper-bound named as tighter extension upper-bound (*teub*) is designed to determine itemsets whose extensions in the enumeration tree do not promise to contain any HAUI. To obtain *teub* values of itemset, their projected databases is used. Details are given below.

Definition 12 (Total Processing Order): The search space of the problem can be represented as an enumeration tree according to the *eub*-ascending order $\prec$, such that $i_1 \prec i_2 \prec \ldots \prec i_n$ where $eub(i_1) < eub(i_2) < \ldots < eub(i_n)$.

In this study, items of 1-EPIs are kept and sorted according to the above total processing order to examine the enumeration tree of the search space. For example, the total processing order $\prec$ is obtained as $\{b \prec a \prec e \prec f \prec c \prec d\}$ for the running example since $eub(b) = 44 < eub(a) = 47 < eub(e) = 58.5 < eub(f) = 64 < eub(c) = 72.5 < eub(d) < 87.5$. Fig. 1 shows the enumeration tree of the running example.

Definition 13 (Items That Can Extend an Itemset): Let X be an itemset, such that each item $X \subseteq$ 1-EPIs. The set of items that can extends X can be denoted as $EIs(X)$, which contains each item $y \in$ 1-EPIs comes after all items of X according to the $\prec$, that is;

$$EIs(X) = \{y | y, where\ \forall x \in X \prec y)\}. \tag{11}$$

For example, $EIs(a) = \{e, f, c, d\}$ and $EIs(\{a, c\}) = \{d\}$.

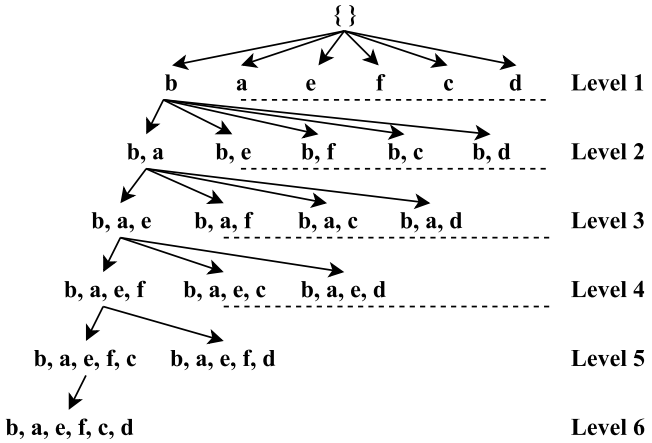


FIGURE 1. The enumeration tree of the running example.

Definition 14 (Projected Transaction Using an Itemset): Let X and T_j be an itemset and a transaction, respectively, such that $X \subseteq$ 1-EPIs and $X \subseteq T_j$. The projected transaction of T_j using X can be denoted as $PT_j(X)$, which contains each item $i \in T_j$, where $i \in \{X \cup EIs(X)\}$.

$$PT_j(X) = \{i | i \in \{X \cup EIs(X)\}, T_j\}. \quad (12)$$

For example, take item f and transaction $T_1 = \{(a : 2)\ (c : 1)\ (d : 1)\ (f : 7)\ (h : 2)\}$. Since $EIs(f) = \{c, d\}$, $PT_1(f)$ is obtained as $\{(f : 7)\ (c : 1)\ (d : 1)\}$. Similarly, $PT_4(f)$ and $PT_6(f)$ are obtained as $\{(f : 6)\ (c : 2)\ (d : 5)\}$ and $\{(f : 3)\ (c : 1)\ (d : 1)\}$, respectively.

Note that, the set of projected transaction of an itemset may contain the identical transactions, i.e., if two or more transactions have the same set of items, they can be called as identical transactions. Merging to the identical transactions into a new single transactions reduces to cost of the projected database scans as mentioned in [29]. The next definition describes the projected database of an itemset.

Definition 15 (Projected Database of an Itemset): The projected database of an itemset X can be denoted as $PDB(X)$, which contains each projected transaction of X after the identical projected transactions are merged into a single transaction. Let $IPTs(X) = \{PT_1(X), PT_2(X), PT_n(X)\}$ be a set of identical projected transactions using X. If they are merged into a new single projected transaction $NPT(X)$, the internal utility of each item i of $NPT(X)$ is defined as follows:

$$iu(i, NPT(X)) = \sum_{PT_j \in IPTs(X))} iu(i, PT_j(X)). \quad (13)$$

For example, since $PT_1(f)$, $PT_4(f)$, and $PT_6(f)$ contain the same set of items, they are identical projected transactions. If they are merged into a single transaction $NPT(f)$, the $iu(f, NPT(f))$ is calculated as $iu(f, PT_1(f)) + iu(f, PT_4(f)) + iu(f, PT_6(f)) = 7 + 6 + 3 = 16$. In similar, $iu(c, NPT(f))$ and $iu(d, NPT(f))$ are calculated as 4 and 7, respectively. Therefore, $NPT(f)$ is obtained as $\{(f : 16)\ (c : 4)\ (d : 7)\}$ after merging the identical projected transactions $PT_1(f)$, $PT_4(f)$, and $PT_6(f)$. As a result $PDB(f) = \{NPT(f)\} = \{(f : 16)(c : 4)(d : 7)\}$.

Based on above definitions, the tighter extension upper-bound can be modeled as follows.

Definition 16 (Tighter Extension Upper-Bound of an Itemset): The tighter extension upper-bound of an itemset X is denoted as $teub(X, PDB(X))$ and defined as:

$$teub(X, PDB(X)) = \sum_{PT_j(X) \in PDB(X)} eub(X, PT_j(X)). \quad (14)$$

For example, $teub(f, PDB(f))$ is calculated as $eub(f, NPT(f))$. Since $NPT(f) = \{(f : 16)(c : 4)(d : 7)\}$, $u(f, NPT(f))$, $u(c, NPT(f))$, and $u(f, NPT(f))$ are obtained as 16, 12, and 56, respectively. Thus, $teub(f, PDB(f)) = eub(f, NPT(f)) = (56 + 16)/2 = 36$.

Theorem 2: In the projected database of an itemset X (or on the sub-tree of an itemset X based on the enumeration tree), none of extensions of X has an average-utility greater than $teub(X, PDB(X))$.

Proof: The proof of Theorem 2 can be done as Theorem 1 is proved, since the values of *eub* and *teub* of itemsets are calculated in the same way. The difference is that the initial database is taken into account for calculation *eub* values while *teub* values are calculated by utilizing the projected databases. ■

Pruning Strategy 2 (Pruning the Search Space Based on teub): Based on Theorem 2, if $teub(X) < minUtil$, then none of extensions of an itemset X on the enumeration tree of the search space can be a HAUI. Thus, there is no need to examine them.

C. BI-DIRECTIONAL TIGHTER EXTENSION UPPER-BOUND (BTEUB)

The projected database of an itemset can also be used to figure out whether the extensions of its single-item extensions contain any HAUI in the enumeration tree. Therefore, we designed another upper-bound named bi-directional tighter extension upper-bound (*bteub*) to determine each itemset Y whose extensions do not promise to contain any HAUI on the projected database of an itemset X, where Y is a single-item extension of X based on the $\prec$. Details are given below.

Definition 17 (Bi-Directional Tighter Extension Upper-Bound of an Itemset): Let X and $Y = \{X \cup \alpha\}$ be two itemsets, where item $\alpha \in EIs(X)$. The bi-directional tighter extension upper-bound of Y is denoted as $bteub(Y, PDB(X))$ and defined as:

$$bteub(Y, PDB(X)) = \sum_{Y \subseteq PT_j(X) \in PDB(X)} eub(Y, PT_j(X)). \quad (15)$$

For example, let $Y = \{f, c\}$ considering that $X = \{f\}$ and $\alpha = \{c\}$. Therefore, $bteub(\{f, c\}, PDB(\{f\}))$ is calculated as $eub(\{f, c\}, NPT(f)) = (56 + 16 + 12)/3) = 28$.

Theorem 3: None of the extensions of an itemset Y which is obtained by extending an itemset X with an item $\alpha \in EIs(X)$ in

the enumeration tree of the search space has an average-utility greater than $bteub(Y, PDB(X))$.

Proof: Theorem 3 can be proved as the way that Theorem 1 and Theorem 2 are proved. ■

Pruning Strategy 3 (Pruning the Search Space Based on bteub): Based on Theorem 3, on the projected database of an itemset X, none of the extensions of $Y = \{X \cup \alpha\}$, where $\alpha \in EI(X)$, can be a HAUI if $bteub(Y, PDB(X)) < minUtil$. Thus, there is no need to examine them. In other words, k-items extensions of X in the enumeration tree of the search space, including an item α can be pruned by utilizing $bteub(Y, PDB(X))$, where $k \geq 2$. Therefore, it can be said that the proposed *bteub* have an ability to prune the search space, bi-directionally. To clarify, let $Y = \{X, c\}$ by assuming that $EIs(X) = \{a, b, c, d, e\}$ and $\alpha = c$, where $a \prec b \prec c \prec d \prec e$. Therefore, if $bteub(\{X, c\}) < minUtil$ holds, then none of $\{X, a, c\}$, $\{X, b, c\}$, $\{X, c, d\}$, and $\{X, c, e\}$ and none of their extensions can be a HAUI. Hence, the extensions of $\{X, c\}$ can be pruned bi-directionally, meaning both of backward extensions ($\{X, a, c\}$ and $\{X, b, c\}$) and forward extensions ($\{X, c, d\}$, $\{X, c, e\}$) of $Y = \{X, c\}$ can be directly removed from the enumeration tree of the search space.

D. MAXIMUM REMAINING K-ITEMS EXTENSION UPPER-BOUND (MAX-REUB$_K$)

The proposed *bteub* is useful to prune 2-item extensions of itemsets. However, some of the 2-item extensions of an itemset may still not be pruned by utilizing the *bteub* values of its single-item extensions. Thus, an interesting question arises as to whether the remaining k-items extensions of an itemset (that cannot be pruned by utilizing the pruning strategy of *bteub* (Pruning Strategy 3)) actually contain HAUI, where $k \geq 2$. To give an answer to this question, another upper-bound named maximum remaining k-items extension upper-bound ($max\text{-}reub_k$) is proposed. Details are given below.

Definition 18 (Utility Upper-Bounds of Itemset X and Item α for the Supersets of the Itemset $\{X \cup \alpha\}$): Let X and $\alpha \in EIs(X)$ be an itemset and item, respectively. Let $E_s(X \cup \alpha)$ be the set of the supersets of $\{X \cup \alpha\}$, i.e., both of the backward and forward extensions of X containing α. The utility upper-bounds of X and α for $E_s(X \cup \alpha)$ can be denoted as $uub(X, E_s(X \cup \alpha))$ and $uub(\alpha, E_s(X \cup \alpha))$, respectively. They are defined as:

$$uub(X, E_s(X \cup \alpha)) = \sum_{\alpha \in PT_j(X) \in PDB(X) \wedge |\{X \cup \alpha\}| < |PT_j(X)|} u(X, PT_j(X)), \quad (16)$$

$$uub(\alpha, E_s(X \cup \alpha)) = \sum_{\alpha \in PT_j(X) \in PDB(X) \wedge |\{X \cup \alpha\}| < |PT_j(X)|} u(\alpha, PT_j(X)). \quad (17)$$

For example, let $X = b$. In the *DB* given in Table 1, transactions $T_4 = ((b : 3)\ (c : 2)\ (d : 5)\ (e : 1)\ (f : 6))$ and $T_6 = ((a : 1)\ (b : 1)\ (c : 1)\ (d : 1)\ (f : 3))$ include b. Therefore, $PDB(b)$ is obtained as $\{PT_4(b), PT_6(b)\} = \{((b : 3)\ (e : 1)\ (f : 6)\ (c : 2)\ (d : 5)), ((b : 1)\ (a : 1)\ (f : 3)\ (c : 1)\ (d : 1))\}$.

However, item α can be a, e, f, c or d since $EIs(b) = \{a, e, f, c, d\}$. If $\alpha = a$, then $uub(b, E_s(\{b, a\}))$ is calculated as $u(b, T_6(b)) = 10$ and $uub(a, E_s(\{b, a\}))$ is calculated as $u(a, T_6(b)) = 5$. If $\alpha = e$, then $uub(b, E_s(\{b, e\}))$ is calculated as $u(b, T_4(b)) = 30$ and $uub(e, E_s(\{b, e\}))$ is calculated as $u(a, T_4(b)) = 4$. If $\alpha = f$, then $uub(\{b\}, E_s(\{b, f\}))$ is calculated as $u(b, T_4(b)) + u(b, T_6(b)) = 30 + 10 = 40$ and $uub(f, E_s(\{b, f\}))$ is calculated as $u(f, T_4(b)) + u(f, T_6(b)) = 6 + 3 = 9$. Similarly, if $\alpha = c$, then $uub(b, E_s(\{b, c\}))$ and $uub(c, E_s(\{b, c\}))$ are obtained as 40 and 9, respectively, and if $\alpha = d$, then $uub(b, E_s(\{b, d\}))$ and $uub(d, E_s(\{b, d\}))$ are obtained as 40 and 48, respectively.

Definition 19 (Maximum Remaining k-Items Extension Upper-Bound ($max\text{-}reub_k$)): Let X be an itemset and $REIs(X)$ be the set of each item α, where $\alpha \in EIs(X)$ and $bteub(\{X \cup \alpha\}, PDB(X)) \geq minUtil$. The maximum remaining k-items extension upper-bound of X is denoted as $max\text{-}reub_k(X)$, where $k \geq 2$. If $REIs(X)$ does not contain more than one item, then $max\text{-}reub_k(Y) = 0$. Otherwise, $max\text{-}reub_k(X)$ is calculated as $max\{reub_2(X), reub_3(X), \ldots, reub_n(X)\}$. Considering that $[uub(X : E_s(X \cup \alpha))]$ and $[uub(\alpha : E_s(X \cup \alpha))]$ be two lists that store each $uub(X, E_s(X \cup \alpha))$ and $uub(\alpha, Es(X \cup \alpha))$ values, respectively, where $\forall \alpha \in REIs(X)$, and are sorted in descending order, each $reub_k(X)$ (k-remaining extension upper-bound utility of X) is calculated as:

$$reub_k(X) = \frac{[uub(X : E_s(X \cup \alpha))][k-1]}{|X| + k} + \frac{\sum_{0 \leq j < k} [uub(\alpha : E_s(X \cup \alpha))][j]}{|X| + k}. \quad (18)$$

For example, let $X = b$. Recall, $minUtil = 26$. Since $bteub(\{b, a\}, PDB(b))$, $bteub(\{b, e\}, PDB(b))$, $bteub(\{b, f\}, PDB(b))$, $bteub(\{b, c\}, PDB(b))$, and $bteub(\{b, d\}, PDB(b))$ are calculated as 7.67, 25.33, 33, 33, and 33, respectively, $REIs(\{b\})$ is obtained as $\{f, c, d\}$. On the other hand, $uub(\{b\}, E_s(\{b, f\})) = uub(\{b\}, E_s(\{b, c\}))$, $uub(\{b\}, E_s(\{b, d\})) = 40$, $uub(\{f\}, E_s(\{b, f\})) = 9$, $uub(\{c\}, E_s(\{b, c\})) = 9$, and $uub(\{d\}, E_s(\{b, d\})) = 48$. Therefore, $[uub(b : E_s(b \cup \alpha))] = [40, 40, 40]$ and $[uub(\alpha : E_s(b \cup \alpha))] = [48, 9, 9]$.

By utilizing these two list, $reub_2(b)$ is calculated as $(40 + (48 + 9)) / (1 + 2) = 32.33$ and $reub_3(b)$ is calculated as $(40 + (48 + 9 + 9)) / (1 + 3) = 26.5$. As a result, $max\text{-}reub_k(\{b, e\})$ is obtained as $max\{reub_2(\{b, e\}), reub_3(\{b, e\})\} = \{32.33, 26.5\} = 32.33$.

Theorem 4: Let X be an itemset and $E_k = \{e_1, e_2, \ldots, e_k\}$ be any set of k-unique items, such that $k \geq 2$, $E_k \subseteq REIs(X)$. Therefore, none of the itemset obtained by extending X with any E_k has an average-utility which is greater than $max\text{-}reub_k(X)$.

Proof: Let $DB(\{X \cup E_k\})$ be a database that includes the set of transactions containing $\{X \cup E_k\}$ in $PDB(X)$. Thus, $au(\{X \cup E_k\}) = \frac{u(X, DB(\{X \cup E_k\}))}{|X| + k} + \frac{u(E_k, DB(\{X \cup E_k\}))}{|X| + k}$.

It is clear that $u(X, DB(\{X \cup E_k\}))$ can be at most equal to $[uub(X : E_s(X \cup \alpha))]$. Besides, $u(E_k, DB(\{X \cup E_k\}))$ can be at most equal to $\sum_{0 \le j < k}[uub(\alpha : E_s(X \cup \alpha))][j]$ is clear, too. Combining these two facts, it can be said that $au(X \cup E_k) \le reub_k(X)$ holds for each $k \ge 2$. As a result, Theorem 4 is correct since $max\text{-}reub_k(X) = \{reub_2(X), reub_3(X), \ldots, reub_n(X)\}$. ■

Pruning Strategy 4 (Pruning the Search Space With $max - reub_k$): Based on Theorem 4, none of the itemset obtained by extending an itemset X with any k-items from $REIs(X)$, $k \ge 2$, have average-utility value which is greater than $max\text{-}reub_k(X)$. Thus, if $max\text{-}reub_k(X)$ is lower than *minUtil*, then none of them is a HAUI. Therefore, there is no need to examine them.

The Pruning Strategy 4 is very efficient to prune the search space. However, an interesting question is arises that is how to determine quickly whether each k-*reub* value of an itemset X is lower than the *minUtil*. Therefore, for quickly determining of whether $\{reub_2(X), reub_3(X), \ldots, reub_n(X)\}$ contains of a value greater than or equal to *minUtil*, an algorithm named as Is-$max\text{-}reub_k$-lower-than-*MinUtil* algorithm (Algorithm 1) is developed based on the following lemmas (Lemmas 1 and 2).

Lemma 1: If $reub_k(X) \ge reub_{k+1}(X)$, then $reub_k(X) \ge reub_{k+1}(X) \ge reub_{k+2}(X) \ge \ge \ldots \ge reub_{k+n}(X)$ holds, where $k \ge 2$.

Lemma 2: If $reub_k(X) < minUtil$ and $[uub(\alpha : E_s(X \cup \alpha))][k] < minUtil$, then none of $reub_m(X)$ can be greater than *minUtil*, where $m \ge k \ge 2$.

Rationale: Lemmas 1 and 2 are clear since values in $[uub(X : E_s(X \cup \alpha))]$ and $[uub(\alpha : E_s(X \cup \alpha))]$ are sorted in descending order.

Algorithm 1 Is-$max\text{-}reub_k$-Lower-Than-*MinUtil*

Input : *minUtil*, $[uub(X : E_s(X \cup \alpha))]$, $[uub(\alpha : E_s(X \cup \alpha))]$.
Output: A boolean value (true or false).

```
1  Calculate reub_2(X);
2  if reub_2(X) ≥ minUtil then
3  |   return false;
4  k = 3;
5  while k ≤ |[uub(α : E_s(X ∪ α))]| do
6  |   if [uub(α : E_s(X ∪ α))][k − 1] < minUtil then
7  |   |   return true; // Based on Lemma 2
8  |   Calculate reub_k(X);
9  |   if reub_k(X) ≥ minUtil then
10 |   |   return false;
11 |   else if reub_{k−1}(X) ≥ reub_k(X) then
12 |   |   return true; // Based on Lemma 1
13 |   k = k + 1;
14 return true;
```

Algorithm 1 gives the pseudo-code of Is-$max\text{-}reub_k$-lower-than-*MinUtil* algorithm. It takes two lists $[uub(X : E_s(X \cup \alpha))]$ and $[uub(\alpha : E_s(X \cup \alpha))]$ which are sorted in descending order, where their size at least equal to 2. It returns a true or false value. The algorithm starts by calculating $reub_2(X)$ (Line 1). Then, it cheeks $reub_2(X)$ against to *minUtil*. If $reub_2(X)$ is not lower *minUtil* (Line 2), it returns false (Line 3). Otherwise, a while loop is performed by the algorithm (Lines 5-13), after k is assigned to 3 (Line 4). In the while loop, the situation mentioned by Lemma 2 is first controlled (Line 6). If the situation of Lemma 2 occurs, then the algorithm returns true (Line 7). Otherwise, the algorithm calculates $reub_k(X)$ (line 8) and checks it against to *minUtil* (Line 9). If $reub_k(X)$ is not lower *minUtil*, it returns false (Line 10). Otherwise, the algorithm controls the situation mentioned by Lemma 1 (Line 11). If the situation of Lemma 1 occurs, then the algorithm returns true (Line 12). If not, k is increased by 1 (Line 13). This while loop is proceeded until a value is returned by itself or $k \not\le |[uub(\alpha : E_s(X \cup \alpha))]|$. If any value is not returned in the while loop, the algorithm returns true (Line 14).

E. EVALUATION OF THE PROPOSED UPPER-BOUNDS BY COMPARING THE EXISTING UPPER-BOUNDS

To enhance the efficiency of solving the problem of HAUIM, several upper-bounds (UBs) have been proposed by the existing algorithms, such as *auub* [7], *lubau* and *tubau* [20], *mau* [21], *lub* and *rtub* [9], *mfuub* and *krtmuub* [10], and $\overline{aub_1}$, $\overline{aub}$, $\overline{iaub}$, and $\overline{laub}$ [22]. Readers who are interested in details of the existing UBs are referred to the related papers. In addition, four new UBs (*eub*, *teub*, *bteub*, and $max\text{-}reub_k$) have been proposed by this study.

The simple way to evaluate these UBs is comparing the values they produce for itemsets. However, as mentioned by [22], any pair of UBs UB_1 and UB_2 may be incomparable if $UB_1(X) \ge UB_2(X)$ and $UB_2(Y) \ge UB_1(Y)$ hold for different two itemsets X and Y. The second way of comparing UBs is to evaluate their stability [22]. For example, it can be said that UB_1 is more stable than UB_2 if UB_1 is tighter than *auub* while UB_2 and *auub* are incomparable. Another comparison can be done by evaluating their pruning abilities (e.g. by determining which UBs can prune the extensions of itemsets).

The evaluation of *eub*, *teub*, *bteub*, and $max\text{-}reub_k$ are discussed as follows.

(*i*). The proposed *eub* and the existing UBs *auub* and $\overline{aub_1}$ are designed to prune items by utilizing the initial DB. From this theoretical aspect, it is relevant to compare *eub* with *auub* and $\overline{aub_1}$.

Let 1-HEUBIs, 1-HAUUBIs, and 1-HAUB$_1$Is be the sets of items that are considered by *eub*, *auub*, and $\overline{aub_1}$ on examination of the k-itemsets ($k \ge 2$), respectively. Therefore, it can be said that |1-HEUBIs| $\le$ |1-HAUUBIs| since $eub(i) \le auub(i)$ holds for each item i based on the fact that $eub(i, T_j) \le tmu(i, T_j)$ is clear for each transaction T_j, $i \in T_j \in DB$. On the other hand, *eub* and $\overline{aub_1}$ are incomparable because of the reason is that if two different items x and y exist, then it is possible to obtain $eub(x) \le \overline{aub_1}(x)$ and $eub(y) \ge \overline{aub_1}(y)$. Thus, |1-HEUBIs|, and |1-HAUB$_1$Is| are also incomparable.

However, since both *auub* and $\overline{aub_1}$ are UBs on average-utility, items which are HAUIs cannot be pruned by them. Hence, |1-HAUUBIs| and |1-HAUB$_1$Is| can not be less than

|1-HAUIs|. On the other hand, |1-HEUBIs| may be less than |1-HAUIs| since *eub* values of items may be lower than their average-utilities.

(*ii*). The proposed *teub* and the existing UBs *mau*, *lub*, *rtub*, *mfuub*, *krtmuub*, *lubau*, *tubau*, $\overline{iaub}$, and $\overline{laub}$ are used to prune the enumeration tree of the search space based on a depth pruning strategy. In other words, for an itemset X, if any of these UBs produces a value that is lower than the *minUtil*, it can be used to discard the extensions of X in the enumeration tree of the search space without examining them. Therefore, it is relevant to compare the proposed *teub* with these existing UBs.

However, the proposed *teub* cannot be directly compared to these existing UBs, because the processing order of the items used for calculation of *teub* values is different from the processing order of the items used to calculate the values of these existing UBs.

On the other hand, the proposed *teub* and the existing UBs *mau*, *lub*, *rtub*, *mfuub*, and *krtmuub* are UBs on average-utility values of extensions of itemsets, meaning that they have an ability to prune the extensions of HAUIs. But, since the existing UBs *lubau*, *tubau*, $\overline{iaub}$, and $\overline{laub}$ are UBs on average-utility values of itemsets, *lubau*, *tubau*, $\overline{iaub}$, and $\overline{laub}$ cannot prune the extension of itemsets which are HAUIs.

In addition, the proposed *teub*, as well as the existing UBs *mau*, *mfuub*, *krtmuub*, and $\overline{iaub}$ cannot produce a value that is higher than *auub* value for any itemset. Conversely, it is possible that the existing UBs *lubau*, *tubau*, *lub*, *rtub*, and $\overline{laub}$ may produce higher values than *auub* values of some itemset.

(*iii*). The proposed *bteub* and the existing UB $\overline{aub}$ have an ability for bi-directionally pruning the extensions of itemsets on the enumeration tree of the search space. In other words, *bteub* and $\overline{aub}$ can prune all the two-items extensions of an itemset X in the enumeration tree of the search space, containing an item y based on the *bteub* and $\overline{aub}$ values of the itemset $\{X \cup y\}$, respectively. Therefore, it is relevant to compare the proposed *bteub* with the existing UB $\overline{aub}$.

However, the *bteub* and $\overline{aub}$ are incomparable. On the other hand, the values produced by $\overline{aub}$ cannot be lower than average-utilities of itemsets since $\overline{aub}$ an UB on average-utility. Thus, $\overline{aub}$ cannot be used to prune the extensions of HAUIs. Contrarily, the proposed *bteub* have an ability to prune the extensions of HAUIs.

Moreover, to calculate *bteub* values of all single-item extensions of an itemset X, only one scan of the projected database of X is enough. On the other hand, in order to calculate the $\overline{aub}$ value of each single item extension of X, the utility vector of each-single item extension of X must be obtained, which is costly.

(*iv*). The proposed *max-reub*$_k$ is used to determine whether all the remaining k-items extensions of itemsets in the enumeration tree of the search space can be pruned, $k \geq 2$. However, there is no existing UB have the same pruning ability that *max-reub*$_k$ has. This is one of the reasons why the HAUL-Growth algorithm, that is proposed by this study, prunes the search space more effectively than the existing algorithms. In addition, *max-reub*$_k$ is tighter than *auub*.

V. PROPOSED DATA STRUCTURES

To effectively solve the HAUIM problem, it can be said that the key issue is to use effective upper-bounds to reduce the search space. On the other hand, proposing data structures that allow quick calculation of average-utility and upper-bound values of itemsets is also very important. For this reason, this sections introduces two new data structures which are named as High Average-Utility List Tree (HAUL-Tree) and Information List (IL).

A. HIGH AVERAGE-UTILITY LIST TREE (HAUL-TREE)

In this section, the definition of the proposed HAUL-Tree structure is first given. Then, the construction process of a HAUL-Tree is described and illustrated with an example. Afterwards, the properties related to the HAUL-Tree data structure is presented, such as extracting projected database and calculating *teub* values by utilizing the HAUL-Tree data structure.

1) DEFINITION OF THE HAUL-TREE

The proposed HAUL-Tree is a compact representation of a transactional database with utility information. It stores only required information of items having *eub* values not lower than the user-defined utility threshold *minUtil*. Since different transactions can have several common items, their path overlaps in the HAUL-Tree based on the order of items *eub*-descending order. The HAUL-Tree data structure can be defined as follows:

- A HAUL-Tree is associated with a prefix itemset. Considering P is a prefix itemset, its HAUL-Tree is called as HAUL-Tree(P). The HAUL-Tree(P) stores the related information of (|P|+1)-EPIs which are single-item extensions of P, where each single-item exists in *EIs*(P). For the compactness, items having larger *eub* values are placed closer to the root in a HAUL-Tree.
- A HAUL-Tree(P) consists of a header table (HAUL-Tree(P).*HT*) and a root node (HAUL-Tree(P).*Root*) labeled as "*null*" with a set of item-prefix subtrees as its children.
- A HAUL-Tree(P).*HT* is a set of heads (or entries). Each head H is related to an item appeared in the HAUL-Tree(P), consisting of two fields which are called as $H.Item$ and $H.Link$. The field $H.Item$ identifies the item related to H while $H.Link$ points a node associated with $H.Item$.
- Each node N in the item-prefix subtrees has an item name called as $N.Item$ and is linked to its parent node, child nodes, and another node carrying the same item name. Besides, it stores the path utilities of items of P and the items that appears on the N-to-Root path (or the branch that starts with N and ends with the root node.) in a list called as path utility list (*PUL*) of N ($N.PUL$).

- The length of the *PUL* of a node N in a HAUL-Tree(P) is the sum of the number of items in P and the number of nodes seen on the N-to-Root path. Let $P = \{p_1, p_2, \ldots, p_{|P|}\}$ and $PNs = \{PN_1, PN_2, \ldots PN_m\}$ be the set of parent nodes of N, such that PN_1 is the parent node of N, PN_2 is the parent node of PN_1, ..., PN_m is the parent node of PN_{m-1}, and the root node is the parent node of PN_m. Therefore, starting from the first element of the $N.PUL$, $N.PUL$ stores the utility values of $p_1, p_2, \ldots, p_{|P|}$, $N.Item$, $PN_1.Item$, $PN_2.Item$, ..., $PN_{m-1}.Item$, and $PN_m.Item$, which are accumulated on the Root-to-N path.

2) CONSTRUCTION OF THE HAUL-TREE

The Insert algorithm (Algorithm 2) gives the pseudo-code of the process of inserting a transaction into the proposed HAUL-Tree data structure. It takes a prefix itemset P and its HAUL-Tree (HAUL-Tree(P), and a transaction T as inputs, where $P \in T$ and $|T| > |P|+1$. Note that, items of T must be in their *eub*-ascending order, such as $T = \{p_1, p_2, \ldots, p_{|P|}, i_1, i_2, \ldots, i_n\}$, where $\{p_1 \prec p_2 \prec \ldots \prec p_{|P|} \prec i_1 \prec i_2 \prec \ldots \prec i_n\}$.

The Insert algorithm starts with assigning the root of the HAUL-Tree(P) as the temporal root *TempR* (Line 1). After that, each item $item_j \notin |P|$ in the input T is inserted into the HAUL-Tree(P) one by one (Lines 2-24). $Item_j$ (j^{th} item of T) is inserted into the HAUL-Tree(P) as follows. If the *TempR* contains a child node C associated with $item_j$ (line 4), then the elements of $C.PUL$ are updated (Lines 5-10). Then, C is assigned as *TempR* (Line 11). The first $|P|$ elements of $C.PUL$ are associated with the prefix itemset P. Thus, each k^{th} element of $C.PUL$ is updated by adding the utility value of $item_k$ in T, where $k < |P|$ (Lines 5-6). The remaining elements of $C.PUL$ are associated with items represented by node C and its parent nodes. Thus, each i^{th} element of $C.PUL$, $i \geq |P|$, is updated by adding the utility value of associated $item_k$ in T, $k \geq j$ (Lines 7-10). Otherwise, a new node N for $item_j$ is generated as a child of *TempR* (Line 13) and each element of $N.PUL$ is set to the related item's utility value in T (Lines 14-19). Then, if the header table includes a head H for $N.Item$ (Line 20), N is linked to other nodes associated with $N.Item$ by the help of H (Line 21). Otherwise, it means that $N.Item$ appears for the first time in the HAUL-Tree(P). Therefore, a head for $N.Item$ is inserted into the header table of the HAUL-Tree(P) (Line 23). Lastly, N is assigned as *TempR* (Line 24). If $item_j$ is not the last item in T, the algorithm continues to proceed from Line 2 for the next item. Finally, it returns the HAUL-Tree(P) (Line 25).

Now, we demonstrate the construction of the HAUL-Tree($\emptyset$) based on the sample database (Table 1) and its external utility table (Table 2) by considering as $minUtil = 26$. Since $P = \emptyset$, the HAUL-Tree($\emptyset$) contains necessary information of 1-EPIs. Above, we have already obtained 1-EPIs $= \{a, b, c, d, e, f\}$ and the total processing order $\prec$ as $(b \prec a \prec e \prec f \prec c \prec d)$ when *minUtil* is set to 26.

Algorithm 2 Insert Algorithm

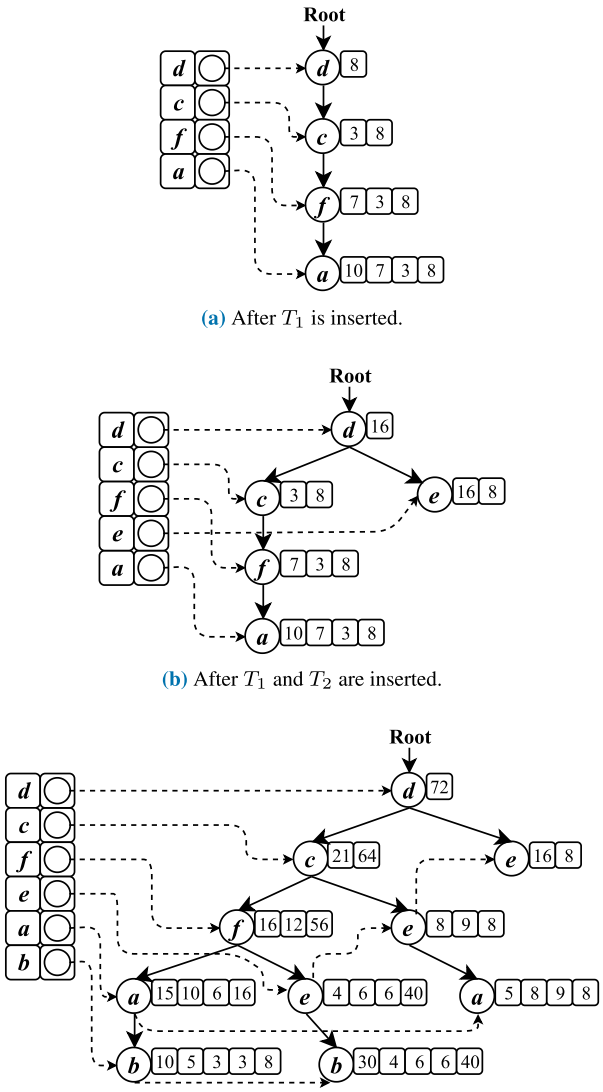
Input : P, a prefix itemset; *HAUL-Tree*(P), the HAUL-Tree of P; T, a transaction that contains P.
Output: *HAUL-Tree*(P), the HAUL-Tree of P.

```
1  TempR = HAUL-Tree(P).Root;
2  for (j = |T|−1, j ≥ |P|, j−−) do
3      item_j = j^th item of T;
4      if TempR has a child node C, where C.Item = item_j
       then
5          for (k = 0, k < |P|, k++) do
6              C.PUL[k] += u(item_k, T);
7          i = |C.PUL| − 1;
8          for (k = |T|−1, k ≥ j, k−−) do
9              C.PUL[i] += u(item_k, T);
10             i = i − 1;
11         TempR = C;
12     else
13         Generate a new node N as a child of TempR,
           where N.Item = item_j;
14         for (k = 0, k < |P|, k++) do
15             N.PUL[k] = u(item_k, T);
16         i = |N.PUL| − 1;
17         for (k = |T|−1, k ≥ j, k−−) do
18             N.PUL[k] = u(item_k, T);
19             i = i − 1;
20         if HAUL-Tree(P).HT has a head H related to
           N.Item then
21             Link N to other nodes associated with
               N.Item;
22         else
23             HAUL-Tree(P).HT ← H(N.Item, N);
24         TempR = N;
25 return HAUL-Tree(P);
```

Note that, from now on, we present each transaction T_j in the form of $\{item_1(u(item_1, T_j)), item_2(u(item_2, T_j)), \ldots, item_n(u(item_n, T_j))\}$ for the simplicity.

First, we insert the first transaction $T_1 = \{a(10), c(3), d(8), f(7), h(30)\}$. However, item h is not a 1-EPI. Therefore, it is removed from T_1 and the remaining items are sorted based on their *eub*-ascending order. After reorganization, T_1 becomes $\{a(10), f(7), c(3), d(8)\}$. The HAUL-Tree($\emptyset$) initially does not have any path. Thus, the reorganized T_1 is inserted in to the HAUL-Tree($\emptyset$) as a new path. Besides, four heads are added to the header table for items d, c, f, and a and linked to corresponding nodes. The HAUL-Tree($\emptyset$) after T_1 inserted is shown in Fig. 2a.

In figures, the links are shown with dashed lines and nodes are represented by circles with attached path utility list *PUL*. In the rest of the paper, we use the notation of ($N.Item$:$N.PUL$)) to mention a node N. For example, the node (f:(7,3,8)) in Fig. 2a represents the item f, where



(a) After T_1 is inserted.

(b) After T_1 and T_2 are inserted.

(c) The completely constructed HAUL-Tree(∅)

FIGURE 2. Construction of the HAUL-Tree(∅).

its $PUL = (7, 3, 8)$. Therefore, we understand that the utility values of items f, c, and d are 7, 3, and 8, respectively, in the (f:(7,3,8))-to-Root path.

Next, the second transaction $T_2 = \{d(8), e(16), g(14)\}$ will be inserted into the HAUL-Tree(∅) after it is reorganized. Item g is not a 1-EPI, and so T_2 is reorganized as $\{e(16), d(8)\}$. Since the root node of the HAUL-Tree(∅) has a child node (d:(8)) with item d, the node (d:(8)) becomes the temporal root *TempR*. Then, its *PUL* is updated as (16) by adding 8 to 8. Afterwards, we continue with item e. However, (d:(16)) has not a child node assigned to item e. Thus, a new node (e:(16,8)) is generated as a child of (d:(16)). There is no head for item e in the header table. Therefore, a head is inserted into the header table for item e and linked to the node (e:(16,8)). The HAUL-Tree(∅) after T_2 inserted is shown in Fig. 2b.

The next transaction to be inserted into the HAUL-Tree(∅) is $T_3 = \{a(5), g(14)\}$. Item g is not a 1-EPI. Thus, reorganized $T_3 = \{a(5)\}$. Since it has only one item, it is not inserted into the HAUL-Tree(∅). The remaining transactions T_4, T_5, are T_6 will be inserted into the HAUL-Tree(∅) after they are reorganized as $\{(b : 30), (e : 4), f(6), c(6), d(40)\}$, $\{(a : 5), e(8), c(9), d(8)\}$, and $\{(b : 10), (a : 5), f(3), c(3), d(8)\}$, respectively. The completely constructed HAUL-Tree(∅) for the running example is shown in Fig. 2c.

3) PROPERTIES RELATED TO THE HAUL-TREE

Property 1: Let P be an itemset. The projected database of any itemset $X = \{P \cup y\}$, such that item y appears in the header table of HAUL-Tree(P), can be easily obtained from the HAUL-Tree(P).

Proof: In a HAUL-Tree(P), for a node N, the utilities of items of P and the items that appears on the Root-to-N path are accumulated and stored in the $N.PUL$. Therefore, if $N.Item = y$, then the N-to-Root path represents a projected transaction using itemset $X = \{P \cup y\}$, $PT^*(X)$. Since we know that $\{p_1 \prec p_2 \prec \ldots, p_{|P|} \prec N.Item = y \prec PN_1.Item \prec PN_2.Item \prec \ldots \prec PN_m.Item\}$, and $N.PUL[0] = u(p_1, PT^*(X))$, $N.PUL[1] = u(p_2, PT^*(X))$, $\ldots$, $N.PUL[|P| - 1] = u(p_{|p|}, PT^*(X))$, $N.PUL[|P|] = u(N.Item, PT^*(X))$, $N.PUL[|P| + 1] = u(PN_1.Item, PT^*(X))$, $N.PUL[|P| + 2] = u(PN_2.Item, PT^*(X))$, $\ldots$, and $N.PUL[|P| + m] = u(PN_m.Item, PT^*(X))$. Thus, a projected transaction using itemset $X = \{P \cup y\}$ can be derived by visiting the nodes that appear on N-to-Root path and utilizing the *PUL* of N. Therefore, the projected database of an itemset $X = \{P \cup y\}$ can be easily obtained by examining the paths of the nodes assigned for item y in the HAUL-Tree(P). ■

For example, by utilizing the HAUL-Tree(∅) shown in Fig. 2c, the projected database of $\{\emptyset \cup e\} = e$ can be obtained as follows. From the head of e, we can reach to the node (e : (4, 6, 6, 40)). It is understood that there are four nodes on the (e : (4, 6, 6, 40))-to-Root path since $P = \emptyset$ and the PUL of (e : (40, 6, 6, 4)) consists of four values. Starting from (e : (40, 6, 6, 4)), these nodes are registered for items e, f, c, and d. Therefore, a projected transaction using e is obtained from this path as $\{e(4), f(6), c(6), d(40)\}$. The other nodes registered for item e in the HAUL-Tree(∅) are (e : (8, 9, 8)) and (e : (16, 8)). We can easily reach these two nodes by the help of link property of the proposed HAUL-Tree data structure. Therefore, by utilizing (e : (8, 9, 8))-to-Root and (e : (16, 8))-to-Root paths, two projected transactions using e are obtained as $\{e(8), c(9), d(8)\}$ and $\{e(16), d(8)\}$, respectively. Table 3 shows the projected database of e obtained by utilizing the HAUL-Tree(∅).

Property 2 (Calculating teub Values of Itemset by Utilizing the HAUL-Tree Data Structure): Let P be an itemset. Let y be an item that appears in the header table of the HAUL-Tree(P). Let $Ns(y) = \{N_1, N_2, \ldots, N_n\}$ be the sets of nodes in the HAUL-Tree(P), such that $N_j.Item = y$. Let $SL_n(N_j.PUL)$ be the sum of the largest n values in $N_j.PUL$. Therefore, by utilizing the HAUL-Tree(P), $teub(X, PDB(X))$, where

TABLE 3. The projected database of *e*, *PDB*(*e*).

Projected TID	Projected transactions with utilities
PT_1	$e(4), f(6), c(6), d(40)$
PT_2	$e(8), c(9), d(8)$
PT_3	$e(16), d(8)$

$X = \{P \cup y\}$, can be calculated as:

$$teub(X, PDB(X)) = \sum_{N_j \in Ns(y) \wedge N_j.PUL > |X|} \frac{SL_{|X|+1}(N_j.PUL)}{|X|+1}. \quad (19)$$

Proof: Let N be a node that appears in a HAUL-Tree(P), such that $N.Item = y$. Based on Property 1, we know that the N-to-Root path represents a projected transaction using the itemset $X = \{P \cup y\}$. Besides, the PUL of N stores the utility of each item that appears on the N-to-Root path, and so it is clear that the length of PUL of N gives the number of items that appears on this path. Thus, $teub(X, PDB(X))$ can be calculated as mentioned by Property 2. ■.

For example, consider the HAUL-Tree($\emptyset$) shown in Figure 2c. Let item y be e. Therefore, $Ns(e) = \{(e : (4, 6, 6, 40)), (e : (8, 9, 8))$ and $(e : (16, 8))\}$. Since $P = \emptyset$, $X = \{\emptyset \cup e\} = e$. As mentioned above, these nodes are easily traversed since they are linked to each other. Since the length of PUL of each these nodes are greater than $|X| = 1$, $teub(e, PDB(e))$ is calculated as $((40 + 6) / 2) + ((9 + 8) / 2) + ((16 + 8) /2) = 43.5$.

Property 2 is very useful to accelerate the mining process since the *teub* value of each single-item extension of P can be calculated directly from the HAUL-Tree(P), and thus the projected databases of non-EPIs are not needed to be obtained.

B. INFORMATION LIST (IL)

The Information List (IL) structure is designed to store related information of the single-item extensions of itemsets. It is utilized to determine which single-item extensions of itemsets are HAUIs, and to prune the search space based on the *bteub* and $max\text{-}reub_k$ upper-bounds.

Definition 20 (Information List (IL) Structure of an Itemset): Considering X is an itemset, the Information List of X is called as IL(X). The IL(X) is a set of entries, where each entry E consists of five fields which are $E.Item$, $E.au$, $E.bteub$, $E.uub(X, S(X \cup E.Item))$, and $E.uub(E.Item, S(X \cup E.Item))$. $E.Item \in EIs(X)$ represents the item that E is related to. $E.au$ and $E.bteub$ store average-utility and *bteub* values of $\{X \cup E.item\}$, respectively. $E.uub(X, S(X \cup E.Item))$ and $E.uub(E.Item, S(X \cup E.Item))$ stand for the utility upper-bound of X and $E.item$ for the extensions of $\{X \cup E.item\}$, respectively.

For example, the Information List of e, IL(e), can be constructed by utilizing the $PDB(e)$ which is given by Table 3. Let us take the first projected transaction $PT_1 = \{e(4), f(6), c(6), d(40)\}$ of $PDB(e)$. In PT_1, the average-utility of $\{e, f\}$, $\{e, c\}$, and $\{e, d\}$ are calculated as $5 = (4 + 6) / 2$, $5 = (4 + 6) / 2$, and $22 = (4 + 40) / 2$, respectively. Besides, the *bteub* value of each single-item extension of e in PT_1 are obtained as divided the sum of 3 greatest values in (4,6,4,40) by 3. Therefore, *bteub* values of $\{e, f\}$, $\{e, c\}$, and $\{e, d\}$ in T_1 is $17.33 = (40 + 6 + 6) / 3$. Moreover, in PT_1, $uub(e, S(e \cup f)) = uub(e, S(e \cup c)) = uub(e, S(e \cup d)) = 4$ while $uub(f, S(e \cup f))$, $uub(c, S(e \cup c))$, and $uub(f, S(e \cup f))$ are obtained as 6, 6, and 40, respectively. The IL(e) by utilizing of $PT_1 \in PDB(e)$ is examined is shown in Table 4.

TABLE 4. Obtained *IL*(*e*) by utilizing PT_1 of *PDB*(*e*).

$E.Item$ (α)	au (e, α)	$bteub$ (e, α)	uub $(e, S(e, \alpha))$	uub $(\alpha, S(e, \alpha))$
f	5	17.33	4	6
c	5	17.33	4	6
d	22	17.33	4	40

TABLE 5. Obtained *IL*(*e*) by utilizing PT_1 and PT_2 of *PDB*(*e*).

$E.Item$ (α)	au (e, α)	$bteub$ (e, α)	uub $(e, S(e, \alpha))$	uub $(\alpha, S(e, \alpha))$
f	5	17.33	4	6
c	13.5	25.67	12	15
d	30	25.67	12	48

TABLE 6. Obtained *IL*(*e*) by utilizing *PDB*(*e*).

$E.Item$ (α)	au (e, α)	$bteub$ (e, α)	uub $(e, S(e, \alpha))$	uub $(\alpha, S(e, \alpha))$
f	5	17.33	4	6
c	13.5	25.67	12	15
d	42	25.67	12	48

Now, we continue with the second projected transaction of $PDB(e)$, which is $PT_2 = \{e(8), c(9), d(8)\}$. The average-utility of $\{e, c\}$ and $\{e, d\}$ in PT_2 are calculated as $8.5 = (8 + 9) / 2$ and $8 = (8 + 8) / 2$, respectively. Besides, in PT_2, the *bteub* values of $\{e, c\}$ and $\{e, d\}$ are calculated as $(9 + 8 + 8) / 3 = 8.33$, and $uub(e, S(e \cup c)) = uub(e, S(e \cup d)) = 8$ while $uub(c, S(e \cup c)) = 9$ and $uub(d, S(e \cup d)) = 8$. The IL(e) is obtained as shown in Table 5 after Table 4 is updated by the related values obtained from $PT_2 \in PDB(e)$.

The last projected transaction of $PDB(e)$ is $PT_3 = \{e(16), d(8)\}$. The average-utility of $\{e, d\}$ in PT_3 is calculated as $12 = (16 + 8) / 2$. However, length of PT_3 is $2 \not> |e| + 1 = 2$. Thus, in PT_3, *bteub* of $\{e, d\}$, $uub(e, S(e \cup d))$, and $uub(d, S(e \cup d))$ equal to 0. The IL(e) after utilizing all the transactions of $PDB(e)$ is given in Table 6.

VI. PROPOSED HAUL-GROWTH ALGORITHM

This study proposes an algorithm named HAUL-Growth algorithm for efficiently mining of HAUIs by utilizing the proposed data structures HAUL-Tree and IL and

pruning the search space by proposed upper-bounds *eub*, *teub*, *bteub*, and *max-reub*$_k$. In this section, the overall process of HAUL-Growth algorithm is first mentioned. Then, its algorithmic description is given. Finally, an execution trace of the HAUL-Growth algorithm is illustrated with an example.

A. OVERALL PROCESS OF HAUL-GROWTH

HAUL-Growth performs two database scans to construct the initial HAUL-Tree(∅). In the first scan, it determines 1-HAUIs and 1-EPIs. In the second scan, it constructs the initial HAUL-Tree(∅). After the HAUL-Tree(∅) has been fully constructed, the high average-utility itemsets can then be mined by HAUL-Growth by performing a pattern growth approach utilizing the data stored in the HAUL-Tree(∅). Therefore, HAUL-Growth algorithm starts with extracting projected database and generating IL for each item in the header table of HAUL-Tree(∅). Considering, there is an item i in the header table. Since the initial prefix is ∅, the new prefix becomes i. The projected database $PDB(i)$ and Information List $IL(i)$ can easily be obtained by traversing the nodes associated with item i using the link property. From the $IL(i)$, we can quickly calculate of average utilities of 2-itemsets which are single-item extensions of item i. Besides, we can easily determine which single-item extensions of item i should be extended based on their *bteub* values stored in the $IL(i)$. Moreover, by utilizing the *uub* values stored in $IL(i)$, we can also determined whether remaining 2-items extensions of i can be pruned. In addition, the HAUL-Tree(i) can be easily constructed by utilizing the PDB(i). Then, the same process is performed for all the items appeared in the header table of the the HAUL-Tree(i). Consider that there is a head associated with item j appeared in the header table of HAUL-Tree(i). Selecting this head, the new prefix becomes $\{i, j\}$. Then PDB($\{i, j\}$) and IL($\{i, j\}$) are obtained by traversing the nodes associated with item j. Recursively, this process is performed by HAUL-Growth until it is not possible to construct any HAUL-Tree. Thus, the complete and the correct set of HAUIs are discovered.

B. ALGORITHMIC DESCRIPTION OF HAUL-GROWTH

The pseudo-code of the HAUL-Growth algorithm is given in Algorithm 3. The algorithm takes a database, *DB*, and a minimum utility threshold percent δ (in %) as input. It starts with collecting average-utility and *eub* values of items, and the total utility of the input *DB* ($tu(DB)$) via a database scan (Line 1). Then, the *minUtil* is obtained as δ% of $tu(DB)$ (Line 2). Next, items having *au* values not less than the *minUtil* are outputted as HAUIs (Line 3) and items having *eub* values not less than the *minUtil* are stored with their *eub* values in a list named 1-EPIs (Line 4). Afterwards, it initializes the HAUL-Tree(∅) with a root node labeled as null (Line 5). In the second database scan (Lines 6-10), each transaction $T_j \in DB$ is inserted into the HAUL-Tree(∅) by sorting remaining items in T_j based on their *eub*-ascending order after discarding items having lower *eub* values than the *minUtil*. Note that, if a transaction does not contain at least two items from 1-EPIs, it is not inserted into the HAUL-Tree(∅) (Line 8) since it cannot be used to extend any itemset. Once the second database scan is performed, items in the header table of HAUL-Tree(∅) are sorted based on their *eub*-descending order (Line 11). Finally, the Search algorithm (Algorithm 4) is called (Line 12) to recursively examine the extensions of itemsets in the enumeration tree of the search space using the necessary information stored in the HAUL-Tree(∅).

Algorithm 3 HAUL-Growth Algorithm

Input : *DB*, a database; δ, a minimum utility threshold percent (in %).
Output: HAUIs, the complete set of discovered high average-utility itemsets.

1 Scan *DB* to collect $au(i)$ and $eub(i)$ for each item $i \in DB$, and $tu(DB)$;
2 $minUtil = tu(DB) \times \delta$;
3 Output 1-HAUIs $\leftarrow \{i | au(i) \geq minUtil, \forall i \in DB\}$;
4 Keep 1-EPIs $\leftarrow \{i, eub(i) | eub(i) \geq minUtil, \forall i \in DB\}$;
5 Initialize the *HAUL-Tree*(∅);
6 **foreach** *transaction* T_j *in DB* **do**
7 Discard each item i from T_j if $i \notin$ 1-*EPIs*;
8 **if** $|T_j| \geq 2$ **then**
9 Sort items in T_j based on their *eub*-ascending order;
10 *HAUL-Tree*(∅) $\leftarrow$ **Insert**(*HAUL-Tree*(∅), T_j);
11 Sort items in *HAUL-Tree*(∅).*HT* based on their *eub*-descending order;
12 **Search**(∅, *HAUL-Tree*(∅), *minUtil*));

The pseudo-code of Search Algorithm is given in Algorithm 4. Starting with the bottom-most head in the header table of the input HAUL-Tree(P), it proceeds each line for each head H. Let the i^{th} header in the header table of the input HAUL-Tree(P) be HAUL-Tree(P).$HT[i]$. The Search algorithm performs the mining process for HAUL-Tree(P).$HT[i]$ in the following way. Firstly, the item represented by HAUL-Tree(P).$HT[i]$ is added to the prefix P to obtain the current prefix CP (Line 2) and the node N is linked to HAUL-Tree(P).$HT[i]$ is reached (Line 3). Afterwards, by traversing the nodes associated with N.*Item*, $teub(CP)$ is calculated (Line 4). If $teub(CP)$ is not lower than *minUtil* (Line 5), then the remaining lines are executed to examine the extensions of CP. Otherwise, the algorithm continues with the next head of the input HAUL-Tree(P). To examine the extensions of CP, projected database and Information List of CP are first obtained (Line 6). After that, two empty lists named as $[uub(CP : E_s(X \cup \alpha))]$, $[uub(\alpha : E_s(CP \cup \alpha))]$, and $REIs(CP)$ are initialized as null (Line 7). The $[uub(CP : E_s(X \cup \alpha))]$ and $[uub(\alpha : E_s(CP \cup \alpha))]$ will be used to store each $uub(CP, E_s(CP \cup \alpha))$ and each $uub(\alpha, E_s(CP \cup \alpha))$, respectively, such that $bteub(CP \cup \alpha) \geq minUtil$ for each α. The $REIs(CP)$ will be used to store the list of each α, where $bteub(CP \cup \alpha) \geq minUtil$.

Algorithm 4 Search Algorithm

Input : P, a prefix itemset; *HAUL-Tree*(P), the HAUL-Tree of P; *minUtil*, a minimum utility threshold.
Output: The set of high average-utility itemsets, HAUIs.

```
1  for (i = |HAUL-Tree(P).HT|-1, i > 0, i--) do
2      CP = P HAUL-Tree(P).HT[i].Item;
3      N = HAUL-Tree(P).HT[i].Link;
4      Calculate teub(CP) via traversing the nodes
       associated with N.Item;
5      if teub(CP) ≥ minUtil then
6          Obtain PDB(CP) and IL(CP) ;
7          [uub(CP : Es(CP ∪ α))], [uub(α : Es(CP ∪ α))],
           REIs(CP) = null;
8          for (j = 0, j < |IL(CP)|, i++) do
9              Ej ← IL(CP)[j];
10             if Ej.au ≥ minUtil then
11                 Output {CP ∪ Ej.item} as a HAUI.
12             if Ej.bteub ≥ minUtil then
13                 REIs(CP) ← Ej.item;
14                 [uub(CP : Es(CP ∪ α))] ←
                   Ej.uub(CP, S(CP, α));
15                 [uub(α : Es(CP ∪ α))] ←
                   Ej.uub(α, S(CP, α));
16         if |REIs(CP)| > 1 then
17             Sort [uub(CP : Es(CP ∪ α))] and
               [uub(α : Es(CP ∪ α))];
18             if ! Is-max-reub_k-lower-than-minUtil(
20             minUtil, [uub(CP : Es(CP ∪ α))],
               [uub(α : Es(CP ∪ α))]) then
21                 Initialize the HAUL-Tree(CP);
22                 foreach Tj in PDB(CP) do
23                     Discard each item i from Tj if i ∉
                       REIs(CP);
24                     if |Tj| > CP + 1 then
25                         Insert(HAUL-Tree(CP), Tj);
26                 Sort items in HAUL-Tree(CP).HT based
                   on eub-descending order;
27                 Search(CP, HAUL-Tree(CP), minUtil);
```

Next, each element of $IL(CP)$ is examined (Lines 8-15). If an element E_j of $IL(CP)$ having average-utility value not lower than *minUtil* (Line 10), the itemset $(CP \cup E_j.Item)$ is a HAUI (Line 11). If an element E_j of $IL(CP)$ having *bteub* value not lower than *minUtil* (Line 12), then $E_j.Item$, $uub(CP, E_s(CP \cup E_j.Item))$, are $uub(E_j.Item, E_s(CP \cup E_j.Item))$ are included in $REIs(CP)$, $[uub(CP : E_s(X \cup \alpha))]$, and $[uub(\alpha : E_s(CP \cup \alpha))]$, respectively (Line 13-15).

After that, the algorithm controls that if $REIs(CP)$ contain at least two items (Line 16). If so, after sorting $[uub(CP : E_s(X \cup \alpha))]$ and $[uub(\alpha : E_s(CP \cup \alpha))]$ in descending order (Line 18), Is-*max-reub*$_k$-lower-than-*minUtil* algorithm (Algorithm 1) is called to determine if *max-reub*$_k$ of CP is lower than *minUtil*. If Algorithm 1 returns false (Line 19), Search Algorithm is called to search HAUIs with the current prefix itemset CP (Line 26) after constructing HAUL-Tree(CP) (Lines 20-25) based on the projected database of CP. This process is operated in the recursive manner until there is no HAUL-Tree will be constructed, i.e., all HAUIs are discovered.

TABLE 7. Average-utility and *eub* values of items.

item	a	b	c	d	e	f	g	h
au	25	40	21	72	28	16	28	30
eub	47	44	72.5	87.5	58.5	64	24.5	20

TABLE 8. The projected database of b, *PDB*(b).

Projected TID	Projected transactions with utilities
PT_1	$(b:10), (a:5), (f:3), (c:3), (d:8)$
PT_2	$(b:30), (e:4), (f:6), (c:6), (d:40)$

C. EXECUTION TRACE OF THE HAUL-GROWTH

In this section, the execution trace of the HAUL-Growth algorithm is presented based on given database in Table 1 and Table 2. Consider δ is set to 10%.

As mentioned above, HAUL-Growth starts with collecting average-utility and *eub* values of items, and total utility of the given database. Table 7 shows the average-utility and *eub* values of items based on the Table 1 and Table 2. Besides, $tu(DB)$ is collected as 260. Thus, *minUtil* is calculated as $260 \times 10 / 100 = 26$.

Therefore, items b, d, e, g, and h are determined as HAUIs since their average-utilities are not lower than 26. On the other hand, a, b, c, d, e, and f are 1-EPIs since their *eub* values are not lower than 26. Thus, the total processing order on 1-EPIs is obtained as $\{b \prec a \prec e \prec f \prec c \prec d\}$. The second database scan is performed in order to construct the HAUL-Tree(∅). Note that, for the running example, the HAUL-Tree(∅) have already constructed and showed in Figure 2c.

To perform pattern growth process, the HAUL-Growth algorithm starts with the bottom-most head in the header table of the tree. Since the bottom-most head in the header table of the HAUL-Tree(∅) is associated with item b, the current prefix itemset CP becomes $\{\emptyset \cup b\} = b$. There are two nodes which associated with item b in HAUL-Tree(∅). These nodes are (b:(10,5,3,3,8)) and (b:(30,4,6,6,40)). By utilizing the PULs of these nodes, $teub(b)$ is calculated as $((10 + 8) / 2) + ((40 + 30) / 2) = 44 \geq 26$. Therefore, the extensions of b should be examined. The obtained $PDB(b)$ by traversing the paths of these nodes is given in Table 8. Based on the $PDB(b)$, the constructed $IL(b)$ is shown in Table 9.

From the $IL(b)$, itemset $\{b, d\}$ is determined as HAUI since $au(\{b, d\}) = 44 \geq minUtil = 26$. Besides, the extensions of itemsets $\{b, a\}$ and $\{b, e\}$ will be pruned from the search space since $bteub(\{b, a\}) = 7.67$ and $bteub(\{b, e\}) = 25.33$ are lower than 26. Therefore, REIs(b) is obtained as $\{f, c, d\}$. As a result, $[uub(b : E_s(b \cup \alpha))] = [40, 40, 40]$

TABLE 9. Information List of *b*, *IL*(*b*).

$E.Item$ (α)	au (b,α)	$bteub$ (b,α)	$uub(b,$ $S(b,\alpha))$	$uub(\alpha,$ $S(b,\alpha))$
a	7.5	7.67	10	5
e	17	25.33	30	4
f	24.5	33	40	9
c	24.5	33	40	9
d	44	33	40	48

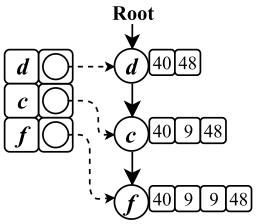


FIGURE 3. The HAUL-Tree(*b*).

TABLE 10. Information List of {*b*, *f*}, *IL*({*b*, *f*}).

$E.Item$ (α)	au (b,f,α)	$bteub$ (b,f,α)	$uub(\{b,f\},$ $S(b,f,\alpha))$	$uub(\alpha,$ $S(b,f,\alpha))$
c	19.33	26.5	49	9
d	32.33	26.5	49	48

and $[uub(\alpha : E_s(b \cup \alpha))] = [48, 9, 9]$. At this point, Is-*max-reub*$_k$-lower-than-*MinUtil* (Algorithm 1) algorithm is called to determine whether *max-reub*$_k(b)$ is lower than the *minUtil*. Is-*max-reub*$_k$-lower-than-*MinUtil* algorithm returns false since $reub_2(b) = (40 + (48 + 9)) / 3) = 32.33 \geq 26$, meaning that there may be HAUI(s) in the extensions of itemsets $\{b,f\}$, $\{b,c\}$, and $\{b,f\}$. Consequently, the mining process continues by constructing the HAUL-Tree(b). Note that each transaction of $PDB(b)$ will be inserted into the HAUL-Tree(b) if it has at least $3 = |b| + 2$ remaining items after ignoring items a and e. The constructed HAUL-Tree(b) is shown in Fig. 3.

The bottom-most head of HAUL-Tree(b) is related to item f. Therefore, CP becomes $\{b,f\}$ and $teub(\{b,f\})$ is calculated as $(48+40+9)/3 = 32.33 \geq 26$ by utilizing the node (f:[40,9,9,48]). Since $teub(\{b,f\})$ is not lower than *minUtil*, $PDB(\{b,f\})$ and $IL(\{b,f\})$ will be obtained. $PDB(\{b,f\})$ is obtained as $\{(b : 40), (f : 9), (c : 9), (d : 48)\}$. $IL(\{b,f\})$ is shown in Table 10.

Therefore, itemset $\{b,f,d\}$ is determined as a HAUI and REIs($\{b,f\}$) is obtained as $\{d,c\}$. Thus, Algorithm Is-*max-reub*$_k$-lower-than-*MinUtil* is called with $[uub(\{b,f\} : E_s(b,f,\alpha))] = [49, 49]$ and $[\alpha : E_s(b,f,\alpha))] = [48, 9]$ to determine whether 2-items extensions of $\{b,f\}$ can be pruned. Since $reub_2(\{b,f\}) = (49 + (48 + 9))/3 = 32.33 \geq 26$, Is-*max-reub*$_k$-lower-than-*MinUtil* algorithm returns false. The process continues to examine the search space with itemset $\{b,f\}$. The constructed HAUL-Tree($\{b,f\}$) is shown in Fig. 4.

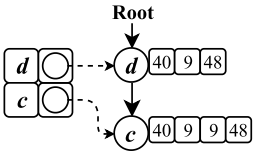


FIGURE 4. The HAUL-Tree({*b*, *f*}).

TABLE 11. Information List of {*b*, *f*, *c*}, *IL*({*b*, *f*, *c*}).

$E.Item$ (α)	au (b,f,c,α)	$bteub$ (b,f,c,α)	$uub(\{b,f,c\},$ $S(b,f,c,\alpha))$	$uub(\alpha,$ $S(b,f,c,\alpha))$
d	26.5	0	0	0

The bottom-most head HAUL-Tree($\{b,f\}$).*HT* is related to item c. There is only a node c:[40,9,9,48] which is related to item c in the HAUL-Tree($\{b,f\}$). By utilizing the node (c:[40,9,9,48]), $teub(\{b,f,c\})$ is calculated as $(48 + 40 + 9 + 9)/4 = 26.5 \geq 26$. Therefore, the algorithm continues by extracting $PDB(\{b,f,c\})$ and constructing $IL(\{b,f,c\})$. The obtained $PDB(\{b,f,c\}) = \{(b : 40), (f : 9), (c : 9), (d : 48)\}$.

$IL(\{b,f,c\})$ is shown in Table 11. $\{b,f,c,d\}$ is a HAUI since $au(\{b,f,c,d\}) = (48+40+9+9)/4 = 26.5$. However, $bteub(\{b,f,c,d\}) = 0$, and so the examination of the search space with $\{b,f,c\}$ is finished. The next head is related to item d, which is the top-most head, and thus the examination of the search space with itemset $\{b,f\}$ is also finished.

The next head in the HAUL-Tree(b).*HT* is related to item c. Thus, $CP = \{b,c\}$. From the node (c:(40,9,48)), $teub(\{b,c\}, PDB(b)) = (48 + 40 + 9) / 3 = 32.33 \geq 26$. However, $bteub(\{b,c,d\}) = 0$ since the length of the PUL of the node (c:[40,9,48]) is not greater than $3 = |\{b,c,d\}|$), and so $uub(\{b,c\}, S(\{b,c,d\}))$=$uub(\{d\}, S(\{b,c,d\}))$=0. On the other hand, au($\{b,c,d\}$)) is calculated as $(40 + 9 + 48)/3) = 32.33$, and thus itemset $\{b,c,d\}$ is determined as a HAUI. Since REIs($\{b,c\}$) = $\emptyset$, the examination of the search space with itemset $\{b,c\}$ is finished. Since the next head is the top-most head in the HAUL-Tree($\{b\}$).*HT*, the examination of the search space with b is also finished.

The algorithm continues for the next head in the header table of HAUL-Tree($\emptyset$), which is related to item a. $CP = a$. There are two nodes associated with item a in the HAUL-Tree($\emptyset$). These nodes are (a:(15,10,6,16)) and (a:(5,8,9,8)). Hence, $teub(a) = ((16 + 15) / 2) + ((9 + 8) / 2) = 24 \not\geq 26$. As a result, none of the extensions of a can be a HAUI.

The next head n the header table of HAUL-Tree($\emptyset$) is related to item e, $CP = e$. The nodes (e:(4,6,6,40)), (e:(8,9,8)), and (e:(16,8)) are related to item e. Therefore, $teub(e) = ((40 + 6) / 2) + ((9 + 8) / 2) + ((16 + 8) / 2) = 43.5 \geq 26$. Recall, IL($e$) have been already obtained and given in Table 6. By utilizing IL(e), $\{e,d\}$ is determined as a HAUI. Besides, REIs(e) = $\emptyset$ since $bteub(\{e,d\}) = 25.67$, $bteub(\{e,c\}) = 25.67$, and $bteub(\{e,f\}) = 17.33$ are lower than 26. Thus, the mining process with e is finished.

TABLE 12. Parameters of the used datasets.

Parameter	Explanation
#T	Total number of transactions
#I	Total number of unique items
MaxL	Maximum length of transactions
AvgL	Average length of transactions
Density	(AvgL / #T) × 100

TABLE 13. Characteristics of the used datasets.

Dataset	#T	#I	MaxL	AvgL	Density
Chess	3,196	75	37	37	49.3333
Mushroom	8,124	119	23	23	19.3277
Accident	340,183	468	51	33.8	7.2222
Pumsb	49,046	2113	74	74	3.5021
BMS	59,602	497	267	2.51	0.5050
T20I9N50D100K	99,163	50	38	16.54	33.0800

The pattern growth operations for remaining heads which are related to items f, c, and d in the HAUL-Tree($\emptyset$).HT are also performed in the same manner, and itemsets $\{f, d\}$, $\{f, c, d\}$, and $\{c, d\}$ are also determined as HAUIs.

VII. EXPERIMENTAL RESULTS

In this section, the performance of the proposed HAUL-Growth algorithm is compared with two state-of-the-art HAUIM algorithms which are TUB-HAUPM and dHAUIM algorithms. The reason is that dHAUIM outperforms all existing HAUIM algorithms as stated in [22], except the TUB-HAUPM algorithm. In addition, dHAUIM and TUB-HAUPM algorithms have not yet been compared.

All the algorithms were implemented using the Java programming language. The experiments were performed on the same computer equipped with an i5-5200U 2.2 GHz processor and 8 GBs of RAM.

Experiments were conducted on different types of datasets, including five real-life datasets (Accident, BMS, Mushroom, Chess, and Pumsb) and a synthetic dataset (T20I10N50D200K). The real-life datasets were obtained from the SPMF open source data mining library [32]. The synthetic dataset is generated using the the IBM Quest Synthetic Data Generator (obtained from [32]). Since the synthetic dataset does not contain internal and external values of items, we generated these values randomly based on the simulation model described in [6], [9], [22]. Therefore, external utilities of items are generated in the [1, 1000] interval by using a log-normal distribution and internal utilities of items are generated in the [1, 10] interval. The parameters of these datasets and their characteristics are presented in Table 12 and Table 13, respectively.

To evaluate the performance of each algorithm, the runtime, number of join operations, maximum memory usage, and the scalability tests were performed. The results are presented below.

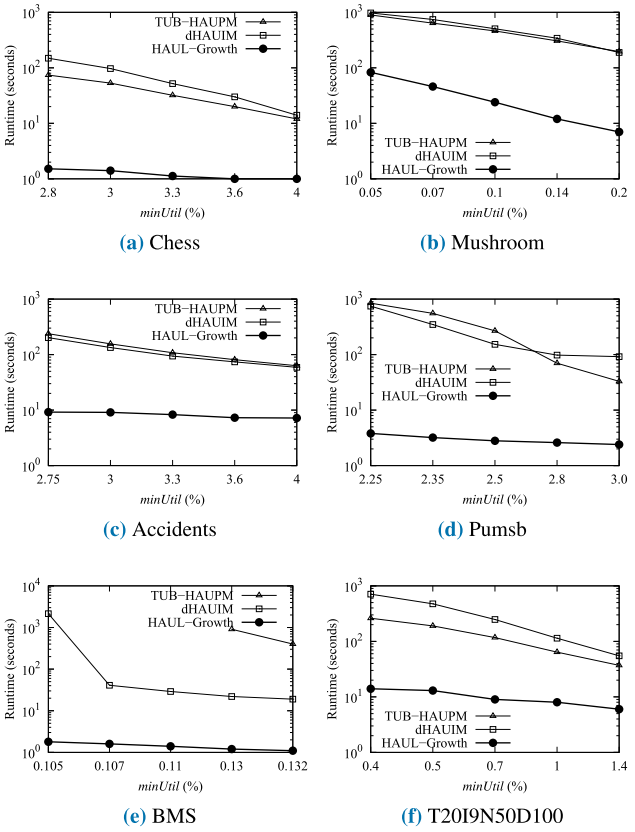


FIGURE 5. Runtimes for various *minUtil* values.

A. RUNTIME

In this subsection, the runtime performances of the algorithms are compared. Experiments are conducted on each dataset under various *minUtil* values.

Note that, for TUB-HAUPM on BMS dataset when $minUtil \leq 0.13\%$, we could not present the experimental results since TUB-HAUPM requires a very long runtime (more than 10^4 seconds) to handle BMS dataset under above mentioned *minUtil* setting.

Fig. 5 shows the results of the algorithms for each dataset. It can be seen that the runtime performances of the algorithms are getting worse when the *minUtil* is decreased. The reason is that the number of discovered HAUIs and performed join operations is increased when *minUtil* is reduced (see Fig. 6). However, it can be seen that the proposed HAUL-Growth algorithm provides a significant performance enhancement comparing to the other algorithms for all datasets for all considered *minUtil* values, especially for low *minUtil* values.

It is observed that based on the experiments conducted on Chess, Mushroom, Accidents, Pumsb, BMS, and T20I9N50D100K, the proposed HAUL-Growth algorithm is respectively up to 98, 27, 22, 196, 1193, and 51 times faster than dHAUIM and up to 49, 28, 24, 222, more than 7142, and 19 times faster than TUB-HAUPM.

Why the proposed HAUL-Growth performs so well can be explained as follows. An important reason is that proposed

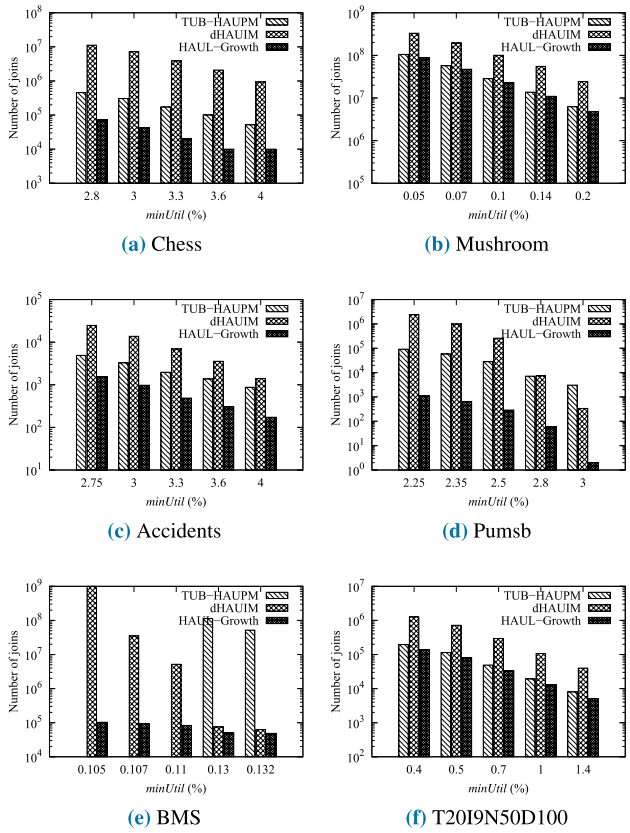


FIGURE 6. Number of join operations for various *minUtil*s.

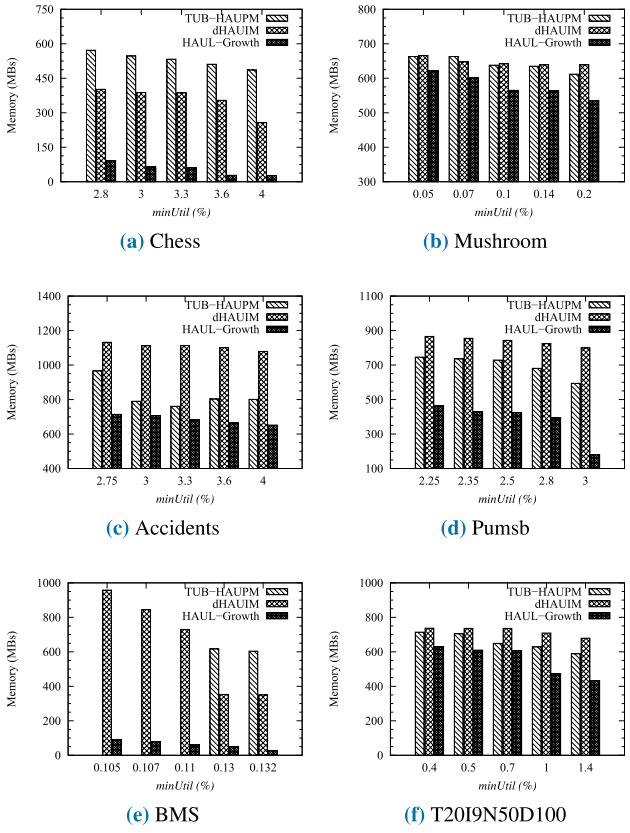


FIGURE 7. Memory usages for various *minUtil*s.

pruning strategies based on the proposed *eub*, *teub*, *bteub*, and *max-reub*$_k$ upper bounds are very efficient to prune the search space of HAUL-Growth. Thus, the number of join operations performed by HAUL-Growth is much less than the previous algorithms(see Fig. 6). Besides, HAUL-Growth compresses the necessary information of the given dataset by utilizing the HAUL-Tree data structure. Moreover, Information Lists of itemsets are quickly obtained by utilizing the link property of the HAUL-Tree data structure. All these proposed upper bounds and data structures allow HAUL-Growth algorithm to achieve better runtime performance.

On the other hand, when the dHAUIM and TUB-HAUPM algorithms are compared among themselves, it is hard to say which one is always better. The reason is that different datasets have different characteristics. Consequently, it can be said that different types of datasets has different effects on the runtime performances of dHAUIM and TUB-HAUPM algorithms.

B. EVALUATION OF THE NUMBER OF JOIN OPERATIONS

In this experiment, we will examine the number of joins operations performed by the compared algorithms to further understand their runtime performances. A join operation means the examination of a single-item extension of an itemset in the search space. Fig. 6 shows the experimental results of number of join operations tests.

As it can be seen in Fig. 6, for each experiment, the number of joins performed by HAUL-Growth is much less than dHAUIM and TUB-HAUPM thanks to the proposed upper-bounds and their pruning strategies. In particular, as it can be seen in Fig. 6e and Fig. 6d, when *minUtil* is reduced for BMS and Pumsb datasets, the number of joins performed by HAUL-Growth does not increase much while the number of joins performed by the others increased enormously. To sum up, the search space of HAUL-Growth is much tighter than the search space of dHAUIM and TUB-HAUPM.

C. EVALUATION OF THE MEMORY USAGE

In this experiment, the peak memory usage of the algorithms is compared. The results are shown in Fig. 7.

As it can be seen in Fig. 7, the memory usage of the algorithm increases by the *minUtil* decreases. This is reasonable, since as the *minUtil* is set lower, the search space is increased, as well as the memory usage of the algorithms.

It can be observed that the proposed HAUL-Growth requires the lowest memory usage for each experiments thanks to data structures it uses. In particular, for Chess (see Fig. 7a) and BMS (see Fig. 7e), it consumes very small memory space compared the other algorithms. This is because the transactions of Chess dataset have many items in common, meaning it is a very dense dataset. Thus, the HAUL-Tree data structure compresses the required information of Chess

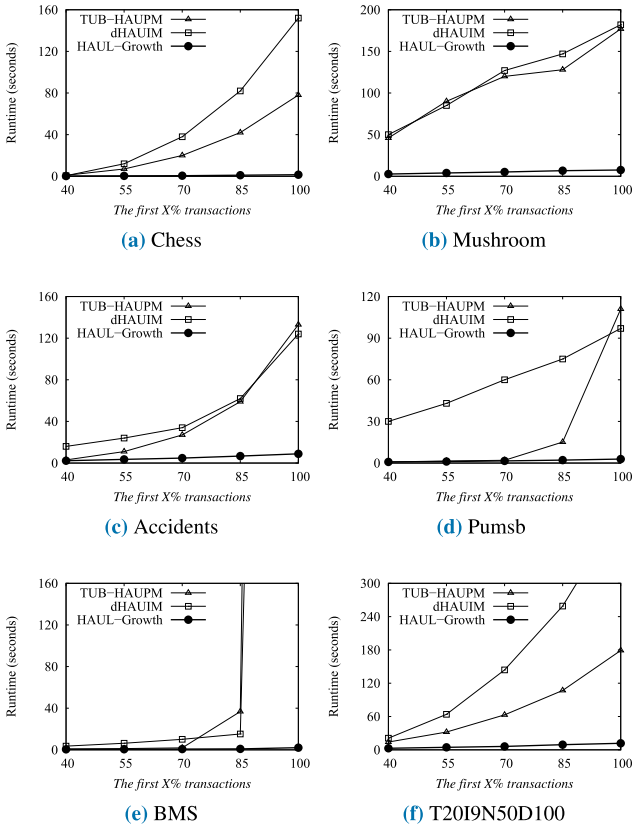


FIGURE 8. Scalability when the size of dataset is varied.

dataset very efficiently. On the other hand, for BMS dataset, it is observed that the proposed HAUL-Growth generated a small number of conditional HAUL-Trees to handle the experiments.

D. SCALABILITY

In this section, we evaluate the scalability performances of the algorithms. For this reason, we generated several datasets by resizing the each experimental dataset as containing the first $X\%$ transactions, where $X\%$ was varied from 40% to 100%. The experiments on Chess, Mushrooms, Accidents, Pumsb, BMS, and T20I9N50D100K were performed using fixed *minUtil* values of 7×10^4, 7×10^3, 6×10^6, 18×10^5, 10^5, and 12×10^6, respectively.

Fig. 8 presents the results of scalability tests of the algorithms in terms of runtime. It is observed that when dataset size increased, the runtime requirement for all the compared algorithm increases. This is reasonable since utilities of the itemsets get larger naturally when the size of dataset increases, as well as the discovered HAUIs. However, it can be seen that HAUL-Growth has the best performance for each experiment and the runtime gaps between HAUL-Growth and the other compared algorithms greatly increases when dataset size gets larger.

It is also observed that when database size increases HAUL-Growth has a very good scalability while the others faced with a scalability problem. It is because a larger dataset contains more transactions. Therefore, with a larger dataset, dHAUIM and TUB-HAUPM perform more computations in the join operations. On the other hand, increasing the size of the dataset does not effect much the computation time of the join operations performed by HAUL-Growth since the dataset is compressed by the HAUL-Tree data structure with overlapping of common parts of transactions over the related paths. Consequently, this experiment also shows that the proposed HAUL-Tree data structure has a significant contribution to the mining process.

VIII. CONCLUSION AND FUTURE WORKS

High-utility itemset mining (HUIM), which is an extension of well-known frequent itemset mining (FIM), takes into account utilities (such as, unit quantities and unit profits) of the itemsets. However HUIM is computationally complex due to the generation of huge number of itemsets with long lengths. To address this problem and extract more meaningful results, the problem of high average-utility itemset mining (HAUIM) was introduced. To improve the efficiency of mining HAUIs, several algorithms have been proposed.

For efficiently mining of HAUIs, this paper proposes a pattern growth approach which is called HAUL-Growth algorithm with introducing four novel upper-bounds (*eub*, *teub*, *bteub*, and *max-reub*$_k$) and two data-structures (HAUL-Tree and IL). The performance of the proposed HAUL-Growth is compared with the performances of the state-of-the-art dHAUIM and TUB-HAUPM algorithms. Experimental results show that the proposed HAUL-Growth outperforms dHAUIM and TUB-HAUPM algorithms in terms of runtime, number of join operations, memory consumption, and scalability.

As a future work, improving the efficiency of HAUL-Growth by proposing more tighter upper-bounds can be studied. Another future work can be seeking efficient strategies for incremental and interactive mining of HAUIs. Note that, the proposed HAUL-Tree data structure can be easily maintained when new transactions are inserted and existing transactions are deleted (or updated). Therefore, modifying the proposed HAUL-Growth in order to handle efficiently updating the discovered HAUIs with transaction insertion and deletion (or updating) can be seen as a good topic.

REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993. doi: 10.1145/170036.170072.

[2] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, 2000. doi: 10.1145/335191.335372.

[3] Z.-H. Deng and S.-L. Lv, "Fast mining frequent itemsets using nodesets," *Expert Syst. Appl.*, vol. 41, no. 10, pp. 4505–4512, 2014. doi: 10.1016/j.eswa.2014.01.025.

[4] Z.-H. Deng, "DiffNodesets: An efficient structure for fast mining frequent itemsets," *Appl. Soft Comput.*, vol. 41, pp. 214–223, Apr. 2016. doi: 10.1016/j.asoc.2016.01.010.

[5] H. Yao, H. J. Hamilton, and C. J. Butz, "A foundational approach to mining itemset utilities from databases," in *Proc. SIAM Int. Conf. Data Mining*, Apr. 2004, pp. 482–486. doi: 10.1137/1.9781611972740.51.

[6] Y. Liu, W.-k. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," in *Advances in Knowledge Discovery and Data Mining*. Berlin, Germany: Springer, 2005, pp. 689–695. doi: 10.1007/11430919_79.

[7] T.-P. Hong, C. H. Lee, and S. L. Wang, "Effective utility mining with the measure of average utility," *Expert Syst. With Appl.*, vol. 38, no. 7, pp. 8259–8265, 2011. doi: 10.1016/j.eswa.2011.01.006.

[8] V. S. Tseng, B.-E. Shie, C.-W. Wu, and P. S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 8, pp. 1772–1786, Aug. 2013. doi: 10.1109/tkde.2012.59.

[9] J. C.-W. Lin, S. Ren, P. Fournier-Viger, and T.-P. Hong, "EHAUPM: Efficient high average-utility pattern mining with tighter upper bounds," *IEEE Access*, vol. 5, pp. 12927–12940, 2017. doi: 10.1109/access.2017.2717438.

[10] J. M.-T. Wu, J. C.-W. Lin, M. Pirouz, and P. Fournier-Viger, "TUB-HAUPM: Tighter upper bound for mining high average-utility patterns," *IEEE Access*, vol. 6, pp. 18655–18669, 2018. doi: 10.1109/access.2018.2820740.

[11] M. Zihayat, H. Davoudi, and A. An, "Mining significant high utility gene regulation sequential patterns," *BMC Syst. Biol.*, vol. 11, no. 6, p. 109, 2017. doi: 10.1186/s12918-017-0475-4.

[12] U. Yun, D. Kim, E. Yoon, and H. Fujita, "Damped window based high average utility pattern mining over data streams," *Knowl.-Based Syst.*, vol. 144, pp. 188–205, Mar. 2018. doi: 10.1016/j.knosys.2017.12.029.

[13] H. Ryang and U. Yun, "High utility pattern mining over data streams with sliding window technique," *Expert Syst. Appl.*, vol. 57, pp. 214–231, Sep. 2016. doi: 10.1016/j.eswa.2016.03.001.

[14] D. Kim and U. Yun, "Efficient algorithm for mining high average-utility itemsets in incremental transaction databases," *Appl. Intell.*, vol. 47, no. 1, pp. 114–131, 2017. doi: 10.1007/s10489-016-0890-z.

[15] I. Yildirim and M. Celik, "FIMHAUI: Fast incremental mining of high average-utility itemsets," in *Proc. Int. Conf. Artif. Intell. Data Process. (IDAP)*, Sep. 2018, pp. 1–9. doi: 10.1109/idap.2018.8620819.

[16] G. C. Lan, T.-P. Hong, and V. S. Tseng, "Efficiently mining high average-utility itemsets with an improved upper-bound strategy," *Int. J. Inf. Technol. Decision Making*, vol. 11, no. 5, pp. 1009–1030, 2012. doi: 10.1142/s0219622012500307.

[17] C.-W. Lin, T.-P. Hong, and W.-H. Lu, "Efficiently mining high average utility itemsets with a tree structure," in *Proc. Asian Conf. Intell. Inf. Database Syst.*, Berlin, Germany, Springer-Verlag, 2010, pp. 131–139. doi: 10.1007/978-3-642-12145-6_14.

[18] T. Lu, B. Vo, H. T. Nguyen, and T.-P. Hong, "A new method for mining high average utility itemsets," in *Computer Information Systems and Industrial Management*. Berlin, Germany, Springer, 2014, pp. 33–42. doi: 10.1007/978-3-662-45237-0_5.

[19] J. C.-W. Lin, T. Li, P. Fournier-Viger, T.-P. Hong, J. Zhan, and M. Voznak, "An efficient algorithm to mine high average-utility itemsets," *Adv. Eng. Informat.*, vol. 30, no. 2, pp. 233–243, 2016. doi: 10.1016/j.aei.2016.04.002.

[20] J. C.-W. Lin, S. Ren, P. Fournier-Viger, T.-P. Hong, J.-H. Su, and B. Vo, "A fast algorithm for mining high average-utility itemsets," *Appl. Intell.*, vol. 47, no. 2, pp. 331–346, 2017. doi: 10.1007/s10489-017-0896-1.

[21] U. Yun and D. Kim, "Mining of high average-utility itemsets using novel list structure and pruning strategy," *Future Gener. Comput. Syst.*, vol. 68, pp. 346–360, Mar. 2017. doi: 10.1016/j.future.2016.10.027.

[22] T. Truong, H. Duong, B. Le, and P. Fournier-Viger, "Efficient vertical mining of high average-utility itemsets based on novel upper-bounds," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 2, pp. 301–314, Feb. 2019. doi: 10.1109/tkde.2018.2833478.

[23] C.-W. Lin, T.-P. Hong, and W.-H. Lu, "An effective tree structure for mining high utility itemsets," *Expert Syst. With Appl.*, vol. 38, no. 6, pp. 7419–7424, Jun. 2011. doi: 10.1016/j.eswa.2010.12.082.

[24] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "UP-Growth: An efficient algorithm for high utility itemset mining," in *Proc. 16th SIGKDD Int. Conf. Knowl. Discovery Data Mining*, New York, NY, USA, 2010, pp. 253–262. doi: 10.1145/1835804.1835839.

[25] M. Liu and J. Qu, "Mining high utility itemsets without candidate generation," in *Proc. 21st Int. Conf. Inf. Knowl. Manage.*, New York, NY, USA, 2012, pp. 55–64. doi: 10.1145/2396761.2396773.

[26] P. Fournier-Viger, C.-W. Wu, S. Zida, and V. S. Tseng, "FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning," in *Foundations of Intelligent Systems* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2014, pp. 83–92. doi: 10.1007/978-3-319-08326-1_9.

[27] J. Liu, K. Wang, and B. C. Fung, "Mining high utility patterns in one phase without generating candidates," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1245–1257, May 2016. doi: 10.1109/tkde.2015.2510012.

[28] S. Krishnamoorthy, "Pruning strategies for mining high utility itemsets," *Expert Syst. Appl.*, vol. 42, no. 5, pp. 2371–2381, 2015. doi: 10.1016/j.eswa.2014.11.001.

[29] S. Zida, P. Fournier–Viger, J. C.-W. Lin, C.-W. Wu, and V. S. Tseng, "EFIM: A fast and memory efficient algorithm for high-utility itemset mining," *Knowl. Inf. Syst.*, vol. 51, no. 2, pp. 595–625, Sep. 2016. doi: 10.1007/s10115-016-0986-0.

[30] H. Ryang and U. Yun, "Indexed list-based high utility pattern mining with utility upper-bound reduction and pattern combination techniques," *Knowl. Inf. Syst.*, vol. 51, no. 2, pp. 627–659, Sep. 2016. doi: 10.1007/s10115-016-0989-x.

[31] A. Y. Peng, Y. S. Koh, and P. Riddle, "MHUIMiner: A fast high utility Itemset mining algorithm for sparse datasets," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2017, pp. 196–207. doi: 10.1007/978-3-319-57529-2_16.

[32] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, and V. S. Tseng, "SPMF: A java open-source pattern mining library," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3389–3393, 2014.

IRFAN YILDIRIM received the B.Sc. degree in computer engineering from Kocaeli University, Kocaeli, Turkey, in 2009, and the M.Sc. degree in computer science from the City College of New York, New York, NY, USA, in 2014. He is currently pursuing the Ph.D. degree in computer engineering with Erciyes University, Kayseri, Turkey. His research interests include pattern mining, big data, and spatio-temporal data mining.

METE CELIK received the B.Sc. degree in control and computer engineering and the M.Sc. degree in electrical engineering from Erciyes University, Kayseri, Turkey, in 1999 and 2001, respectively, and the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, MN, USA, in 2008. He is currently a Faculty Member of the Department of Computer Engineering, Erciyes University, Turkey. His research interests include data analysis, big data, spatial databases, spatial data mining, spatio-temporal data mining, and location-based services. He is a member of the ACM.