IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 6 August 2023, accepted 27 August 2023, date of publication 13 September 2023, date of current version 28 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3314984

RESEARCH ARTICLE

FTKHUIM: A Fast and Efficient Method for Mining Top-K High-Utility Itemsets

VINH V. VU[1], MI T. H. LAM[1], THUY T. M. DUONG[1], LY T. MANH[1], THUY T. T. NGUYEN[1], LE V. NGUYEN[1], UNIL YUN[2], VACLAV SNASEL[3], (Senior Member, IEEE), AND BAY VO[4]

[1]Faculty of Information Technology, Ho Chi Minh City University of Industry and Trade, Ho Chi Minh 700000, Vietnam
[2]Department of Computer Engineering, Sejong University, Seoul 05006, Republic of Korea
[3]Faculty of Electrical Engineering and Computer Science, VŠB-Technical University of Ostrava, Poruba, 70800 Ostrava, Czech Republic
[4]Faculty of Information Technology, HUTECH University, Ho Chi Minh 700000, Vietnam

Corresponding author: Bay Vo (vd.bay@hutech.edu.vn)

ABSTRACT High-utility itemset mining (HUIM) is an important task in the field of knowledge data discovery. The large search space and huge number of HUIs are the consequences of applying HUIM algorithms with an inappropriate user-defined minimum utility threshold value. Determining a suitable threshold value to obtain the expected results is not a simple task and requires spending a lot of time. For common users, it is difficult to define a minimum threshold utility for exploring the right number of HUIs. On the one hand, if the threshold is set too high then the number of HUIs would not be enough. On the other hand, if the threshold is set too low, too many HUIs will be mined, thus wasting both time and memory. The top-k HUIs mining problem was proposed to solve this issue, and many effective algorithms have since been introduced by researchers. In this research, a novel approach, namely FTKHUIM (Fast top-k HUI Mining), is introduced to explore the top-k HUIs. One new threshold-raising strategy called RTU, a transaction utility (TU)-based threshold-raising strategy, has also been shown to rapidly increase the speed of top-k HUIM. The study also proposes a global structure to store utility values in the process of applying raising-threshold strategies to optimize these strategies. The results of experiments on various datasets prove that the FTKHUIM algorithm achieves better results with regard to both the time and search space needed.

INDEX TERMS Knowledge data discovery, high-utility itemset, top-k HUIM, threshold-raising strategy.

I. INTRODUCTION

Knowledge data discovery is the data processing technique to turn huge amounts of non-meaningful data into knowledge that is appropriate for the context and use in specific circumstances. HUIM is a significant research subject and has been widely applied in real life in fields such as bioinformatics, mobile commercial planning, analyzing users' actions and attitudes from their clicks on a website, cross-marketing [1], [2], and so on. HUIM can be thought of as a generalization of traditional Frequent Itemset Mining (FIM) [3], [4], [5], [6], finding the set of items where the frequency of occurring together is high (called FIs) in each database. Each record contains a list of the quantity of items that customers purchased, and each item also has a profit unit. FIM is only interested in the number of appearances of items and ignores other important information. Therefore, the results of FIM may contain frequent itemsets that have low profits. Instead of choosing products with high frequency, users tend to look for groups of high-profit items.

Assuming δ is the smallest utility value given by the user (threshold), HUIM's purpose is to discover itemsets whose utility is not less than δ. In this, the utility of the itemset is measured as the sum of all benefits that the retailer earns from all items in that itemset. Several efficient algorithms have been developed to resolve this real problem, such as Two-Phase [7], TWU-Mining [8], HUC-Prune [9], UP-Growth, and UP-Growth+ [10], among others. In particular, in recent years, with the development of many new storage structures such as UL, CUL, ECUS, and so on, as well as pruning strategies based on these, the runtime and memory consumption for HUI mining has been greatly reduced, as seen

The associate editor coordinating the review of this manuscript and approving it for publication was Chien-Ming Chen.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

with approaches such as with HUP-Miner [11], d2HUP [12], HUI-Miner [13], FHM [14], IMUP [15], EFIM [16], and HMiner [17]. Moreover, this problem has also been expanded to different types of databases, such as incremental databases with EIHI [18], databases with dynamic profits [19], adding constraints to itemsets for more meaningful HUIs such as CoHUI [20], [21], [22], [23], itemsets with the length constraint FHM+ [24], or closed HUIs (CHUIs) [25], [26]. However, choosing an appropriate value δ in HUIM is not a simple task, as it needs to be based on the properties and apportionment of the data, which are not normally known to the user. Determining the appropriate threshold will optimize the processing of the HUIM algorithm because the mining results differ greatly in size depending on the selected minimum threshold value δ. If the threshold value is chosen as a small value, a huge number of HUIs are found that cannot be used in practice and the performance of the HUIM algorithms is poor. Conversely, if the threshold value is chosen as a high value, the user cannot find any HUIs. So, the top-k HUIM is studied to solve this problem. Typical algorithms to achieve this are TKU [27], TKO [28], REPT [29], TONUP [30], kHMC [31], TKEH [32], THUI [33], and TKO-BPSO [34]. The top-k task has also been examined to solve the problem of negative profits, such as TKN [35], TopHUI [36], and so on.

Top-k HUIM is only interested in the number of k-HUIs that need to be discovered instead of the value of δ. Using the k parameter helps the user to limit the size of the output and easily specify the itemsets that have the most profit. The top-k HUIM is a current problem that has recently attracted the interest of many researchers. Top-k HUIM algorithms can be divided into two types: one-phase and two-phase algorithms.

Two-phase algorithms are two-step solving algorithms: step 1, generating possible candidates that satisfy the problem condition through the upper bounds and lower bounds values; step 2, determining the exact utility of the candidates and giving the final result of the problem. The execution time of two-phase algorithms is often slow and they tend to use a lot of memory. In contrast, the one-phase algorithms produce candidates and accurately define their utility in just one step, so they have better execution times and use less memory.

Mining top-k HUIs is a highly applicable problem in the real world and there have been many efficient approaches to solve it. In the top-k HUIM, two problems need to be solved: i) Beginning from the user-defined value k, the algorithm needs to increase the minimum threshold utility as quickly as possible; ii) using pruning strategies to reduce the search space of the problem and so reduce the unnecessary waste of execution time and memory usage. However, in previous works we found that increasing the threshold in the early stages is not as effective in some cases, such as RIU, PSD, etc. Therefore, we propose an effective strategy to increase the threshold in the early stages, namely RTU. By using RTU in the first stage of the mining, the minimum threshold utility of the algorithm increases faster, and thus the set of potential items is significantly reduced.

In addition, the use of local storage structures when the raising-threshold strategies are applied in the previously proposed approaches means that the utility values that are found cannot be reused. When each strategy to increase the minimum threshold is applied, the k highest utility values must be redefined from the beginning, wasting a lot of search time and decreasing the performance of the algorithm. To solve this limitation, we recommend using a global priorityQueue structure for storing the k highest utility values. This leads to a faster increase of the minimum threshold utility and reduces both the search space and execution time.

A. RESEARCH CONTRIBUTIONS

- A threshold-raising strategy is proposed for the top-k HUIM based on the TU value. This strategy makes the minimum threshold increase in the first steps of the top-k HUIM.
- We apply a global structure for the remaining utility values of the previous threshold-raising strategies that are used as a basis for the following strategies to update the threshold faster.
- A combination of threshold-raising and search space pruning strategies is utilized effectively.
- The FTKHUIM algorithm is proposed, which efficiently mines the top-k HUIs on both dense and sparse databases, especially with an improved execution time.

B. ORGANIZATION

The structure of this research is organized as follows. The second section briefly reviews related work on high-utility itemset mining and top-k utility itemset mining. The third section presents the basic theory, background definitions, notations, and examples in these fields. The fourth section shows the proposed FTKHUIM algorithm to effectively mine top-k HUIs. The evaluation of experimental results is shown in the fifth section, which incorporates all the effective techniques for raising the threshold presented in the previous sections. The sixth section then presents the conclusion and some directions for future studies.

II. LITERATURE REVIEW

In this part we summarize some problems regarding HUIM, top-k HUIM, and some of the proposed solutions, along with their advantages and disadvantages.

A. MINING HIGH-UTILITY ITEMSETS

Many effective HUI mining algorithms have been proposed, with great success in the data mining field. In 2005, Wu et al. suggested the Two-Phase (Y. Liu et al., 2005) algorithm. With this, phase 1 creates candidates with high TWU values, and phase 2 extracts HUIs with only one more database scan. Many other studies have also proposed mining HUIs in two phases, such as TWU-Mining [8], HUC-Prune [9], UP-Growth [10], etc.

However, using two-phase algorithms needs a large memory or long runtime if a vast number of candidates are created

in order to discover only a very small number of HUIs. To address this problem, the HUI-Miner algorithm [13] was recommended to exploit HUIs without generating candidates. The HUI-Miner introduced the Utility-List (UL) structure which stores not only an itemset's utility information (iutil), but also keeps utility information to define if the itemset should be truncated or not ($iutil + rutil \geq \varepsilon$). In the FHM algorithm [14], the EUCS table structure was proposed to store the itemsets' TWU values that have two items. This algorithm uses the EUCP strategy to remove candidates containing an itemset in the EUCS table whose $TWU < \varepsilon$ to decrease the search space. Subsequently, many studies have also focused on developing pruning strategies to discover HUIs effectively, such as HUP-Miner [11], d2HUP [12], IMUP [15], EFIM [16], HMiner [17], MIP [37], and HUI-PR [38]. The HMiner algorithm [17] combines many pruning strategies, such as TWU-Prune, EUCS-Prune, LA-Prune, C-Prune, and U-Prune, and uses the CUL (Compact Utility List) structure to store the essential information, so the database is scanned only once. In 2018, Deng proposed a data structure called PU–tree-Node list (PUN-list) and introduced a high-utility itemset mining approach using Pun-list (MIP) to find HUIs. The MIP algorithm [37] only used a single database scan to discover HUIs in one phase. Wu et al. then proposed the HUI-PR algorithm [38] to exploit HUIs in 2019. This algorithm introduced pruning strategies to decrease the algorithm's implementation time by decreasing access to non-essential nodes. The two upper bounds, slocU (the strict local utility) and ssubU (the strict sub-tree utility), are used to reduce the time needed in cases with many transactions in the database. In addition, extensive research has been studied to exploit HUIs using evolutionary algorithms.

B. MINING TOP-K HUIS

In practice, users often only focus on the utility itemsets that give them the most profit. Therefore, setting a minimum utility threshold δ to be able to exploit the best k HUIs is not a simple task. To resolve this issue, many top-k HUIs algorithms have been researched. Tseng et al. mentioned the top-k HUIM task and introduced TKU [27] and TKO [28]. Top-k HUIM usually focuses on two main issues: developing strategies to raise δ and constructing pruning strategies to minimize the search space.

TKU was the first algorithm, proposed in 2012, to resolve the top-k HUIs problem. TKU is a two-phase algorithm developed from UP-Growth. In the first phase, TKU focuses on finding candidates. In the second phase, the TKU scans the data again to correctly identify the top-k HUIs to be collected. The strategies used to increase thresholds in TKU are PE (Pre-evaluation) in the first phase, and then MC, MD, NU, and SE in the second phase. The PE strategy is applied during the first scan of the database. The main idea of this strategy is based on the lower bound of 2-itemsets that are stored in the PEM (PE matrix). For each transaction $T_r = \{i_1, i_2, i_3, \ldots, i_m\}$ in the database, the PE strategy updates the utility value of the itemsets associated with the first item with each remaining item in T_r and saves to PEM. After all the transactions are retrieved, the strategy assigns the k^{th} largest value in the PEM as the algorithm's current threshold. Although PE has raised the initial threshold for the TKU algorithm. But this strategy only focuses on exploiting the itemsets containing the first item of each transaction, and the values in PEM are not optimized. The TKU also uses four pruning strategies based on UP-Tree: DGU, DGN, DLN, and DLU.

Also based on the UP-Tree structure, in 2015 Ryang and Yun introduced an algorithm called REPT [29]. In REPT, the authors proposed three strategies to effectively raise δ in the first phase to accurately compute the utility of itemsets whose lengths are 1 or 2, namely RIU (Real Item Utilities), PUD (Pre-evaluation with Utility Descending order), and RSD (Raising the threshold with items in Support Descending order). The PUD strategy is an improvement of the PE strategy. Instead of combining the first item with the following items in an order for each transaction, the PUD chooses the most profitable item to associate with the remaining items and this combination can build the lower bound of the 2-itemsets better than PE. However, the item with the most profit is not necessarily the item that has the greatest utility in the transaction. The second strategy is the RIU strategy. This is a threshold-strategy based on the utility of the 1-itemsets in the database. The RIU is implemented by calculating the utility of the items correctly on the first scan of the database and choosing the k^{th} highest value to use for updating the minimum utility threshold. Meanwhile, the third strategy, namely RSD, is performed in the second scan of the database. The RSD strategy requires determining the support of all promising items whose TWU is greater than the current threshold value. RSD sorts promising items in descending order of support. Next, RSD will choose $k/2$ items with the highest support and $k/2$ items with the lowest support to form a set. As each transaction is scanned a second time, RSD inspects the 2-itemsets that are made up of the set created in the previous step in turn. The RSD calculates their utility and stores them in the RSD matrix (RSDM). RSD will search the k^{th} highest value in RSDM to decide whether to update the minimum utility threshold. A method to increase the threshold faster with the exact and pre-calculated utilities for determining a set of precise top-k high-utility patterns, termed SEP, is also proposed in the second phase. Both TKU and REPT generate a large number of candidates because they follow the two-phase model. They also scan the dataset repeatedly to obtain the exact utility of candidate itemsets and mine the actual top-k HUIs. One problem that exists in two-phase algorithms is that they need a lot of execution time as well as storage space, because they do not combine the generation of candidates and the precise calculation of their utility.

With regard this this specific weakness, many one-phase methods have been shown to tackle the top-k HUIs problem more efficiently. In 2016, using the UL structure, the TKO algorithm was shown to solve the top-k HUIs problem with one phase. Many strategies have also been applied in TKO,

such as RUC, RUD, and EPB, to increase the minimum threshold, and the results of the related experiments prove that TKO is better than TKU. However, storing the itemsets' utility information requires a huge among of space.

Duong et al. introduced the kHMC algorithm to explore top-k HUIs, which is also another one-phase algorithm. This algorithm introduced two strategies to reduce the search space: a threshold-increasing strategy called EUCST based on the EUCS structure, and a transitive extension pruning (TEP) strategy. Moreover, the algorithm also uses effective threshold-raising strategies such as RIU, COV (COVerage with utility descending order), and CUD (Co-occurrence with Utility Descending order). CUD is like the EUCS pruning strategy. Instead of calculating the TWU of itemset $A = \{a, b\}$, the CUD strategy computes $u(A)$ and stores it in a triangular matrix, namely CUDM. The k^{th} highest value in the CUDM matrix $\leq\delta$. This is easily proven because the values in CUDM are also the utility of itemsets with a size of two, which is the subset of all the sets in the database D. The COV strategy uses a list named COVL for storage of the utility values of itemsets, which is built step by step as follows: Step 1, all values in CUDM are saved to COVL; Step 2, for each 1-itemset i: j and i will be added to COVL if all the transactions containing j also contains i. Repeat step 2 until all i itemsets are processed. The k^{th} highest value in the COVL $\leq\delta$.

Liu et al. recommended the TONUP algorithm [30], which is an algorithm developed from the d2HUP algorithm by developing the CAUL structure to iCAUL. The TONUP algorithm has optimal threshold-raising methods and five strategies (ExactBorder, SuffixTree, AutoMateria, DynaDescend, OppoShift) to maintain the list of compact patterns, calculate the utility efficiently and estimate the upper-bound to minimize the search space.

In 2018, Singh et al. proposed the TKEH algorithm to find the answer to this problem. TKEH [32] is based on the database projection method, and besides applying effective threshold-raising strategies such as RIU, COV, and COD, the algorithm also takes advantage of database projection as well as the pruning strategies introduced in EFIM, and thus TKEH performs very well on both dense and sparse databases compared to previous algorithms.

In 2019, the one-phase THUI algorithm [33] was proposed, and this uses two structures to store data, the UL and LIU (Leaf Itemset Utility) matrix. In this study, LIU is the utility value of a continuous sequence of ordered items existing in the dataset. In addition to applying two strategies to increase the threshold (RUI, and RUC), THUI also proposes strategies such as LIU_E (LIU Exact), and LIU_LB (LIU Lower Bound) based on the LIU value of the itemsets that have been calculated and stored in the LIU matrix.

LIU_E is a threshold-raising strategy based on the utility of sequences of items sorted in a given total order $\succ$. Denote $(x_a \dots x_b) = \{x_a, x_{a+1}, x_{a+2}, \dots, x_b\}$, as the items are sorted in alphabetical order, then $(a \dots d) = \{a, b, c, d\}$ or $(b \dots d) = \{b, c, d\}$. Therefore, to save the utility of the $(x_a \dots x_b)$ sets, the algorithm will use a structure called the LIUM, a triangular matrix, similar to the EUCS structure, and $LIU\,(x_a \dots x_b) = \sum_{X=(x_a \dots x_b) \wedge T_j \in D} u(X, T_j)$. To calculate the LIU values for all the sets on the database, the algorithm will scan the database once more and calculate them similarly to the values of the EUCS table, so the complexity of the algorithm will be $O(l^2)$, where l is the average length of transactions in D. It is easy to see that the LIUM stores all the utility of sets in the form of sequences, so the k^{th} highest LIU value, denoted as $k - LIU$, then $k - LIU \leq \delta$. LIU_LB is a threshold-raising strategy developed from the LIU strategy to construct the lower bound of an itemset. This strategy is based on the following property: $u(A \cup B) \leq u(A) + u(B)$ therefore $u(A) \geq (A \cup B) - u(A)$. Let $ULB(B) = u(A \cup B) - u(A) = u(A \cup B) - \sum_{A \subset t_j} u(A, t_j)$, $LIULB((a, b, y)) = ULB(\{(a \dots b) - y\})$ and PQ_ALL is a set that contains all the LIU and LIULB values of the database, then the k^{th} highest value of $PQ\_ALL \leq \delta$. With the method determining LIULB as described above, the strategy only applies to sequence B whose size is greater than two, and it will gradually remove the items in this until B contains only two items at the beginning and end positions. However, the calculation process will take a lot of time, and in reality LIULB is only calculated for levels three or four, meaning beginning from itemset B the algorithm removes step by step up to four items.

The efficiency of the top-k HUIs algorithm depends greatly on the raising-threshold rate of the δ value during the mining process. The faster the threshold increases, the earlier and larger the number of candidates is pruned. However, the previous algorithms focus on the utilities of 1-itemsets (RIU) or 2-itemsets (RSD, CUD) or sequences of item sequences (LIU_E, LIU_LB) to increase the threshold in the early stages without using the transaction utilities to handle this issue. Another problem is that for each raising-threshold utility strategy applied, the top-k utility values must be redefined from the beginning. This causes an unnecessary waste of search time because the utility values found are not reused.

However, most top-k HUIM algorithms can only deal efficiently with databases whose sizes are small to medium-sized, and their performance degrades significantly with huge datasets or massive data. In 2021, Han et al. [39] presented a novel algorithm, namely PTM, to address this limitation. PTM divides the original dataset into partitions which contain the transactions whose prefix items are the same. PTM can define an itemset's utility in one partition and then utilizes the tree structure to exploit the required results. Moreover, PTM presents a method for raising the threshold faster using the average transaction utility. In 2022, Pham et al. proposed the TKO-BPSO algorithm [34]. The top-k problem has also been developed with regard to databases with some items that have negative profits, with the binary swarm optimization algorithm being applied to the top-k HUIM in this context. The TKO-BPSO algorithm works on a one-phase mechanism, using the method of increasing the minimum threshold RUC to decrease the search space. Furthermore, the algorithm

applies the sigmoid function in the process of updating the candidates. This means that the algorithm can reduce computational complexity, especially when a large database has many items.

Recently, many top-k HUIM methods have been upgraded to apply to databases where items have negative profits, the first of these being the TopHUI algorithm [36]. This algorithm was developed from the FHN algorithm [40] with an initial utility threshold of 0. In addition to inheriting the pruning strategies available in FHN, TopHUI applied some new strategies – RIU, RTU, RTWU, and RUC – for raising the minimum threshold. However, the PNU list structure is complex, so TopHUI has not been optimized for the storage space. In 2021, Sun et al. shared a method dealing with this issue in the THN algorithm [41], which was developed from the EIHI method and applies only a threshold-raising method, namely RTWU, for all its execution. The main idea of THN is to start from HUIs containing items with positive utility and then expand to items with negative utility to find HUIs in the top-k HUIs. However, the above algorithms have not yet provided a strategy to increase the effective threshold in the implementation process, and thus the execution performance is not very good, and all executions must be based on the RTWU value, which does not reflect the real utility of the itemsets. In 2022, Ashraf et al. studied the TKN algorithm [35] to mine the top-k HUIs on a negative database with many threshold-raising strategies, and they did not depend too much on the RTWU value. TKN applies the method of projecting and merging duplicate transactions introduced in EFIM to reduce the complexity of the calculation process. The algorithm also uses many threshold methods and pruning strategies to narrow the search space, such as PRIU, PLIU_E, PLIU_LB, PSU, PLU, EP, and EA. TKN also uses the UA array structure to quickly compute the utility of patterns.

III. PROBLEM STATEMENT AND RELATED NOTIONS

A. RELATED NOTATIONS

Let $D = \{T_1, T_2, \ldots, T_n\}$ be a transaction database containing n transactions. A transaction $T_j = \{x_h \in I \mid h = 1, 2, \ldots, N_j\}$, where N_j is the number of items in the transaction T_j. Each item $x_h (1 \le h \le m)$ in T_j has a pair value $iu(x_h, T_j)$ and $eu(x_h)$ called internal and external utility, respectively.

For example, Table 1 and Table 2 present database D and the profit of the items, respectively.

Definition 1: The $u(x_h, T_j) = iu(x_h, T_j) \times eu(x_h)$ is the utility of x_h in T_j. The $u(X, T_j) = \sum_{x_h \in X} (x_h, T_j)$ and $u(X) = \sum_{X \subseteq T_j \subseteq D} u(X, T_j)$ are utilities of itemset X in the transaction T_j and D.

For example, $u(a, T_1) = iu(a, T_1) \times eu(a) = 1 \times 4 = 4$, $u(b, T_3) = 2 \times 3 = 6$.

Definition 2: The transaction utility T_j, denoted as $tu(T_j)$ and $tu(T_j) = \sum_{X \subseteq T_j \wedge x_h \in X} u(x_h, T_j)$.

For example, $tu(T_1) = u(a, T_1) + u(b, T_1) + u(d, T_1) + u(e, T_1) = 4 + 6 + 4 + 3 = 17$, $tu(T_2) = 15$, $tu(T_3) = 21$.

TABLE 1. An example of a transaction database.

Tid	Transaction	Purchase Quantity (IU)	Utility (U)	Transaction Utility (TU)
1	*a b d e*	1 2 2 3	4 6 4 3	17
2	*b c e f*	1 5 1 2	3 5 1 6	15
3	*b c d e*	2 1 6 2	6 1 12 2	21
4	*c d e*	2 2 3	2 4 3	9
5	*a f*	1 1	4 3	7
6	*a b c d e*	1 1 4 4 1	4 3 4 8 1	20
7	*b c e f*	3 2 2 3	9 2 2 9	22

TABLE 2. Item profits.

Item	*a*	*b*	*c*	*d*	*e*	*f*
Profit ($)	4	3	1	2	1	3

Definition 3: $twu(X)$ is called the transaction-weighted utility (TWU) of an itemset X, and $twu(X) = \sum_{X \subseteq T_j \wedge T_j \in D} tu(T_j)$

For example, $twu(a) = tu(T_1) + tu(T_5) + tu(T_6) = 17 + 7 + 20 = 44$, $twu(b) = 95$.

Definition 4: An itemset X is called HUI if and only if $u(X) \ge \delta$.

Definition 5: The sets of k HUIs that have the highest utilities in D are called top-k HUIs.

***Problem statement*:** Given a transaction database D and k is the desired number of HUIs, the problem of mining top-k HUIs is to correctly identify the k HUIs in D that have the highest utility values.

For example, in Table 1, when $k = 4$, top − k HUIs = $\{\{c, d, e\} : 37, \{b, d\} : 39, \{b, c, e\} : 39, \{b, d, e\} : 45\}$.

A total order $\succ$ is applied for all items in the database, i.e., the items will be sorted in ascending order by TWU value. In Table 4, the ordering of items is $a \succ f \succ d \succ c \succ b \succ e$. The extension of an itemset X are the items listed after X in order $\succ$. For example, $\{c, b, e\}$ is the extension of $\{d\}$.

Definition 6: $ru(X, T_j) = \sum_{x_i \in (T_j/X)} u(x_i, T_j)$ is defined as the remaining utility of X in T_j with T_j/X being the itemset of all items after X in T_j.

Definition 7: The remaining utility of an itemset X in database D is calculated by $ru(X) = \sum_{X \subseteq T_j \in D} ru(X, T_j)$.

Definition 8: Given an itemset $X = \{x_1, x_2, ..., x_h\}$ and an extension item $y \in I$. The prefix utility of an itemset $Xy = \{x_1, x_2, ..., x_h, y\}$ in transaction T_j is defined as $pu(Xy, T_j) = u(X, T_j)$. If X is an empty set, the prefix utility of an itemset Xy in the transaction T_j will be 0.

Definition 9: The $cu(X, T_j)$, $cru(X, T_j)$, *and* $pu(X, T_j)$ are called closed utility, closed remaining utility and prefix utility of X in transaction T_j, respectively. And if $|X| > 1$ and $C(X - x_h) = S(T_j/\{X - x_h\})$, then $cu(X, T_j) = u(X, T_j)$, $cru(X, T_j) = ru(X, T_j)$ and, $cpu(X, T_j) = pu(X, T_j)$ or else they will all be 0. Where $S(T_j/X)$ is the number of items after X in T_j (in the mentioned above order $\succ$) and $C(X)$ is the size of the closed extensions of X.

For the running example, let $X = \{d, c\}$. In transaction T_3, $S(T_3/\{X - c\}) = |\{c, b, e\}| = 3$, $C(X - c) = |\{c, b, e\}| = 3$. Therefore, $cu(X, T_3) = U(X, T_3) = 13$. Similary, $cu(X, T_4) = 0$ as $S(T_4/\{X - c\}) = |\{c, e\}| = 2 \neq C(X - c) = |\{c, b, e\}| = 3$.

Definition 10: Closed utility, closed remaining utility and prefix utility of X in the database are defined as $cu(X), cru(X), cpu(X)$, respectively and $cu(X) = \sum_{X \subseteq T_j \in D} cu(X, T_j)$, $cru(X) = \sum_{X \subseteq T_j \in D} cru(X, T_j)$, and $cpu(X) = \sum_{X \subseteq T_j \in D} cpu(X, T_j)$

Definition 11: Non-closed utility, non-closed remaining utility and non-prefix utility of X in the database are defined as $nu(X), nru(X) npu(X)$ and $nu(X) = u(X) - cu(X)$, $nru(X) = ru(X) - cru(X)$, *and* $npu(X) = pu(X) - cpu(X)$

B. EFFECTIVE PRUNING STRATEGIES

To improve runtime and memory performance a number of pruning strategies are used to identify unexpected items that should be ignored, namely TWU–Prune, U–Prune, LA–Prune, C–Prune and EUCP.

Property 1 (TWU–Prune [42]): Let X be an itemset, if $twu(X) < \delta$, then X and all extensions of X are not HUI.

Property 2 (U–Prune [13]): Let X be an itemset, if $u(X) + ru(X) < \delta$, then X and all extensions of X are low utility itemsets, which means that they are not HUI.

Property 3 (LA–Prune [11]): Let X and Y be two given itemsets, if: $cu(X) + cru(X) + nu(X) + nru(X) - \sum_{\forall T_j \in D, X \subseteq T_j \wedge Y \nsubseteq T_j} nu(X, T_j) + nru(X, T_j) < \delta$ then $X'Y' \notin HUI, \forall X' \supseteq X and \forall Y' \supseteq Y$.

Property 4 (C–Prune [17]): Let X and Y be two given itemsets, if $cu(X) + cru(X) + \sum_{XY \subseteq T_j \in D} nu(X, T_j) + nru(X, T_j) < \delta$ then $X'Y' \notin HUI, \forall X' \supseteq X and \forall Y' \supseteq Y$

Property 5 ((EUCP [14]): Let X be an itemset. If the TWU of a 2-itemset $Y \subseteq X$ according to the constructed EUCS is less than the minimum utility threshold (δ), X and all extensions of X are not HUI, where EUCS is a triangular matrix that stores the TWU values of the 2-itemsets.

IV. FTKHUIM ALGORITHM

In this section, the research introduces the TU strategy for raising the threshold in the first database scan in Section A and the global structure for storing k-highest utilities which are explored by threshold-raising strategies in Section B. We will then also present the proposed algorithm, namely FTKHUIM, in Section C and give an illustration of its use in Section D, and complexity analysis in Section E.

A. TU THRESHOLD-RASING STRATEGIES

Two issues that greatly affect top − kHUIs mining problems in transactional databases are the methods of raising the minimum utility threshold and search space pruning strategies which are applied for maximum efficiency. If threshold-raising is done well, then early detection of a high-valued minimum utility threshold will help to eliminate many unpromising candidates. In contrast, of it is done poorly then the mining process needs a lot of time for those candidates which are not in the top − kHUIs.

Each transaction in the database is also an itemset that needs to be considered. Therefore, their utility can be used in the process of specifying the initial threshold for the algorithm. On the first database scan, the utility of transactions will be determined. The utility of a transaction T_c in D is denoted by $tu(T_c)$ and is defined as $tu(T_c) = \sum_{x \in T_c} u(x, T_c)$.

Property 6: Let $RTU = \{tu(T_1), tu(T_2), tu(T_3), \ldots, tu(T_n)\}$ be a set of utilities of transactions in $\mathcal{D}$ and $tu(t_K)$ is the k^{th} highest value in RTU. $tu(T_K)$ is not greater than the utilities of the itemsets in the top-k HUIs.

Proof: Let $\{i_1, i_2, \cdots, i_k\}$ be the items in the transaction T_j. Then $X = \{i_1, i_2, \cdots, i_k\}$ is also an itemset and $tu(T_j)$ is also the utility of X in transaction T_j.

Assuming database D has n transactions, D will generate a set as $\{X_1, X_2, X_2, \ldots, X_n\}$ with the utility set $\{tu(T_1), tu(T_2), tu(T_3), \ldots, tu(T_n)\}$, respectively. Each X_i is an element in the top-k search space, so the k^{th} highest value in the set $\{tu(T_1), tu(T_2), tu(T_3), \ldots, tu(T_n)\}$ can be used as the minimum utility threshold (δ) in mining the top-k HUIs in the database D. Therefore, the k^{th} highest utility value in D will not be less than $tu(T_K)$.

From this property, the utilities of the transactions can be applied to determine the minimum threshold value δ for top-k HUIs mining.

B. GLOBAL STORE STRUCTURE

In top-k HUIs mining algorithms, the minimum utility threshold-raising process uses an array or a queue to store the highest utility values found. From those values, the algorithm determines the δ value for the algorithm. Although many different threshold-raising strategies are used in turn throughout the problem-solving process, these methods are independent and do not reuse the maximum utilities found by the previous strategies. Therefore, when a threshold-raising strategy is applied, it first must find enough k utility values which are greater than the current threshold to fill full the stored structure, then it can start to consider updating the new threshold. This takes time to execute. Moreover, the k utility values that are greater than the current threshold found in the previous thresholding strategy are completely ignored. To solve this limitation, this study proposes using the global priorityQueue structure to store the k maximum utilities in all thresholding processes of the algorithm. Storing in a global structure will make it possible for the algorithm to reuse those maximum utility values and δ value can increase faster.

C. FTKHUIM ALGORITHM

One problem in the previously proposed algorithms is that the initial thresholding is all based on the RIU strategy, meaning the value δ of in top-k HUIM is calculated based on the utility of 1- itemsets in the database. However, it is easy to see that each transaction in D is also an itemset, and the TU value is also its utility. It is thus possible to rely on the TU of the

transactions to determine the value of δ for the top-k HUIs problem.

Furthermore, when applying the threshold-raising strategies the utility values stored in a priority queue or any type of data structure must be initialized without reusing the stored values in previous strategies, leading to long execution times as well as inefficiency. Therefore, our proposed method develops the presented threshold-raising techniques, reuses the found high utility values, and predicts which of the previous strategies can be used as a prerequisite for the following strategies. However, this will face the following obstacle: the duplication of candidates in the processing. To solve this, we use the hashmap structure to check whether the candidates are the same or not.

In this algorithm, we will apply all the search space pruning strategies that have been applied effectively in HUIM to make the top-k HUIs mining more effective.

Algorithm 1 The FTKHUIM algorithm

Input: D: a transaction database
k : number of itemsets
Output: top − kHUIs: the set of k highest utility itemsets.
1 Assign $\delta = 1$.
2 Scan D to compute the TU of each transaction, TWU, and utilities for all 1-itemsets.
3 $\delta \longleftarrow$ the k^{th} highest TU value by RTU strategy;
4 Update δ by RIU strategy;
5 Sort items in increasing according to their TWU (the total order $\succ$);
6 Eliminate all items i that $twu\,(i) < \delta$ from all transactions, sort items in each transaction by $\succ$, and remove the null transactions.
7 Compute LIU and update δ using the LIU_E strategy
8 Initialize *1-CULs*;
9 Initialize hashmap *HT* and save the transaction utilities of transactions.
10 **for each** *ts* in *D* **do**
11 **if** *ts* is duplicated with any previous transaction **then**
12 Update TidInfo in *CULs* that contain items in *ts*.
Update the *HT*
13 **Else**
14 Insert new TidInfo to *CULs* that contain items *ts*;
Insert new tran into *HT*
15 **Endif**
16 **end for**
17 $\delta \longleftarrow$ the k^{th} highest *tu* value in *HT*;
18 Build the EUCS and CUDM;
19 $\delta \longleftarrow$ the k^{th} highest utility in CUD;
20 $\delta \longleftarrow$ the k^{th} highest utility in LIU_LB strategy;
21 top − kHUIs = **ExplorekHUI** $(\emptyset, 1 - CULs, \delta, k)$;

The FTKHUIM is the key procedure of our algorithm. The input of FTKHUIM includes a transaction database D, and k is the desired number of highest utility itemsets. The output of the procedure is a list of k highest utility itemsets in the database. In the first step, algorithm initializes the utility threshold as 1. Then, the scanning database is done to calculate the *tu*, *twu*, and *u* of all items in the database. The TU strategy is applied by selecting the k^{th} highest *tu* value for δ value. Next, algorithm uses the RIU strategy to update the value for δ. Thereby non-potential items, whose TWU value is less than the δ current threshold, are removed. The others will be put in ascending order based on their TWU value. After adjusting the database, the algorithm calculates the LIU table and updates σ once more based on the LIU strategy. In this way, the module constructs all 1-CULs and builds EUCS and CUDM to prepare for the search space. It then applies the CUD and LUI_LB strategies for raising the δ value. Finally, the ExploreKHUI procedure is called to exploit the top − kHUIs satisfying all the requirements of the problem.

Algorithm 2 ExplorekHUI

Input: P: the itemset prefix, *CULs*,
δ : current thresholds
k : number of itemsets
Output: top − kHUIs: the set of k highest utility itemsets
1 **for** each position i in *CULs* **do**
2 $X = P \cup CULs[i].item$;
3 $u = CULs[i].nu + CULs[i].cu$;
4 $ru = CULs[i].nru + CULs[i].cru$;
5 **if**$(u \geq \delta)$ **then**
6 Update top − kHUIs and δ
7 **endif**
8 **if**$(u + ru \geq \delta)$ **then**
9 $exCULs = buildCUL(|X|, CULs, i, \delta)$;
10 **if** $(exCULs! = NULL)$ **then**
11 $ExplorekHUI(X, exCULs, \delta, k)$;
12 **endif**
13 **endif**
14 **endfor**

The ExplorekHUI algorithm is implemented recursively to detect all itemsets belonging to the top − kHUIs in D. The algorithm will traverse each CUL to check whether an itemset satisfies the current δ threshold or not. If an itemset X has a utility greater than δ, X will be added to the top − kHUIs and the value of δ will be updated if necessary. The U–Prune pruning strategy is also applied to see if it is necessary to expand itemset X. If $u + ru \geq \delta$, the algorithm will be executed and construct CULs at the next level to find undiscovered HUIs in D.

Algorithm 3, namely buildCUL, is responsible for generating (k+1)-CULs from k-CULs by combining k-CULs in the position *start* with k-CULs from *start* onwards. To decrease the time consumption and memory needed for this process, the algorithm has applied various of the proposed effective pruning strategies, such as U-Prune, C-Prune, EUCS-Prune, and LA-Prune.

Algorithm 3 buildCUL

Input: l: length of itemset, start: starting position, CULs, δ : current utility thresholds
Output: *exRCULs*: the list CUL extensions of CULs[start]

```
1  size = |CULs| − start;
2  exSize = size;   //real quantity of elements of exCULs
3  x = CULs[start].item;
4  Initialize LAU[size]; //use for LA-Prune
5  Initialize CUTIL[size]; //use for C-Prune
6  for i from 0 to size − 1 do
7      y = CULs[start + i].item;
8      if (EUCS [x, y] ≥ δ) then
9          Initialize exCULs[i];
10         Initialize LAU[i], CUTIL[i];
11     else
12         exCULs[i] = NULL;
13         exSize = exSize − 1;
14     endif
15 endfor
16 Update cru of exCUL in front of exCUL assigned NULL
17 vt = start − 1;
18 for each TidInfo in CULs[vt].TranSet do
19     tid = TidInfo.tid;
20     for j from 0 to size − 1 do
21         //if tid is not in CULs[j + vt].TranSet
22         if (CULs[j + vt].TranSet does not
           contain tid) then
23             Update LAU[j];
24             if (LAU[j] < δ) then
25                 exCULs[j] = NULL;
26                 exSize = exSize − 1;
27                 Des = exCUL [j] .cu−
                               exCUL[j].pu;
28                 Update cru of exCULs in font of
                   position j by Des;
29                 Continue;
30             endif
31     else
32         Update CUTIL[j];
33     endif
34 endfor
35 if (tid contains in all CULs after CULs[vt])
   then   //complete transaction
36     Update cpu, cu, cru, C_δ in all exCULs;
37 else
38     if (tid is duplicated with the previous
       transactions) then
39         Update value in TidInfo;
40     else
41         Insert new TidInfo in all exCULs[j] that
               CULs[j + start] contains tid;
42     endif
43 endif
44 endfor
45 Remove exCULs[j] that exCULs[j] = NULL or
   CUTIL [j] < δ
46 return exRCULs
```

D. ILLUSTRATION

First, let $\delta = 1$ and $k = 4$. Next, FTKHUIM scans the database from Table 1 and calculates the TU of each transaction, as shown in Table 3.

TABLE 3. TU values of each transaction.

Item	1	2	3	4	5	6	7
Profit ($)	17	15	21	9	7	20	22

From the results in Table 3, apply the threshold-raising TU strategy to increase the δ value TU, and δ is updated to 17.

Next, the algorithm calculates the TWU values of the items to use in TWU pruning. We obtain the value of TWU as shown in Table 4.

TABLE 4. TWU value.

Item	a	b	c	d	e	f
twu	44	95	87	67	104	44

In the next step, the algorithm applies the RIU strategy to update the minimum threshold value δ (if possible).

TABLE 5. RIU values.

Item	a	b	c	d	e	f
RIU	12	27	14	28	12	18

Since the RIU of the items is not large, we cannot update the value of δ (the current value of δ at 17). In this step, the algorithm puts the RIU values that satisfy the threshold into a priority queue named priorityQueue, then *priorityQueue* = {18, 27, 28}. Based on the TWU values in the previous step, the algorithm removes the items where $twu < \delta$. In Table 4, the TWU values are relatively large compared to δ, so no items are excluded from the dataset. However, the items are ordered by ascending TWU: $a \succ f \succ d \succ c \succ b \succ e$. From there, the items in the transaction are reordered and any empty transactions are removed (because some 1-itemsets can be eliminated in the previous step).

TABLE 6. Modified database by total order of items.

Tid	Transaction	Purchase Quantity (IU)	Utility (U)	Transaction Utility (TU)
1	a d b e	1 2 2 3	4 4 6 3	17
2	f c b e	2 5 1 1	6 5 3 1	15
3	d c b e	6 1 2 2	12 1 6 2	21
4	d c e	2 2 3	4 2 3	9
5	a f	1 1	4 3	7
6	a d c b e	1 4 4 1 1	4 8 4 3 1	20
7	f c b e	3 2 3 2	9 2 9 2	22

Next, the FTKHUIM determines the values of the LIU matrix, and the results are presented in Table 7.

The algorithm then applies the threshold-raising LIU_E strategy to update the value δ. When applying the LIU strategy, it uses the values stored in the *priorityQueue* and inserts

TABLE 7. LIU matrix.

a	b	c	d	e	f
b	7	0	0	0	0
c		0	0	0	0
d			31	34	37
e				33	39
f					36

new utilities that are equal to or greater than the current threshold δ, as shown below:

TABLE 8. Priority values.

Priority	δ
18, 27, 28, 31	18
27, 28, 31, 34	27
28, 31, 34, 37	28
31, 34, 37, 33	31
33, 34, 37, 39	33

In the end, the value of $\delta = 34$.

In steps 8 to 16, the algorithm builds 1-CUL structures for 1-itemsets. In the process, the algorithm also aggregates duplicate transactions and stores these new TU values in a hash table, namely HT. In these steps, the algorithm only stores the TU values of transactions that have more than two items. This is because with those transactions that only contain one or two items, their TU values are used in the RIU strategy and the CUD strategy, respectively. We thus have the following TU values: {9, 17, 21, 37, 20}. These values will be used to update δ again in the next step of the TU threshold-raising strategy.

In step 17, the TU strategy continues to be applied after the duplicate transactions are merged (as shown in Table 6, transaction 2 and transaction 7 will be merged). If we construct a new *priorityQueue*, this means that the values in the *priorityQueue* in the LIU_E strategy are not reused. The newly calculated TU values are {9, 17, 21, 37, 20}. Then, there are only 37 greater than $\delta = 34$ to be inserted into the new *priorityQueue*. This makes it impossible to raise the value of δ. If we use the previously stored *priorityQueue*, it will raise the value of δ and make the search space more compact. From the currently stored tuple {34, 37, 39, 36} when considering $tu = 37$, the value 37 will be inserted into the *priorityQueue* and *priorityQueue* becomes {36, 37, 39, 37} and δ receives the value 36.

Step 18. The algorithm will build EUCS and CUD at the same time because the two processes are similar. This is because reusing the *priorityQueue* will help the algorithm increase the threshold to 37 when the CUD strategy (step 19) is implemented with the values of CUDM shown in the following table:

To increase δ, the algorithm continuously applies the LIU_LB strategy in step 20. However, in this case the value

TABLE 9. CUD matrix.

a	b	c	d	e	f
b	7	20	8	17	12
c	0	22	27	18	
d	31	39	37		
e	33	23			
f	36				

of $\delta = 37$ remains unchanged because the value of LIU_LB is not greater than the current threshold.

With the value δ found and based on the 1-CUL structure, the ExplorekHUI algorithm will identify all the HUIs belonging to the top $-$ kHUIs to be mined. In the ExploreKHUI algorithm, the CULs will be considered in turn to determine whether the utilities of itemsets are equal to or greater than δ. If it is satisfied, it will put these itemsets in the top $-$ kHUIs and can eliminate an itemset that has the lowest utility when the top $-$ kHUIs already contain k HUIs and update the value for δ. The EUCP and U-prune pruning strategies will be applied to consider whether the existing CUL needs to be extended. If the expansion condition is relevant, the algorithm will combine the current CUL with the following CULs to form a new list of CULs. From there, the ExploreKHUI algorithm will be called recursively so that it can inspect all the cases of HUIs that exist in the database and can become a candidate for the top $-$ kHUIs.

In the end, the algorithm gives the following result:

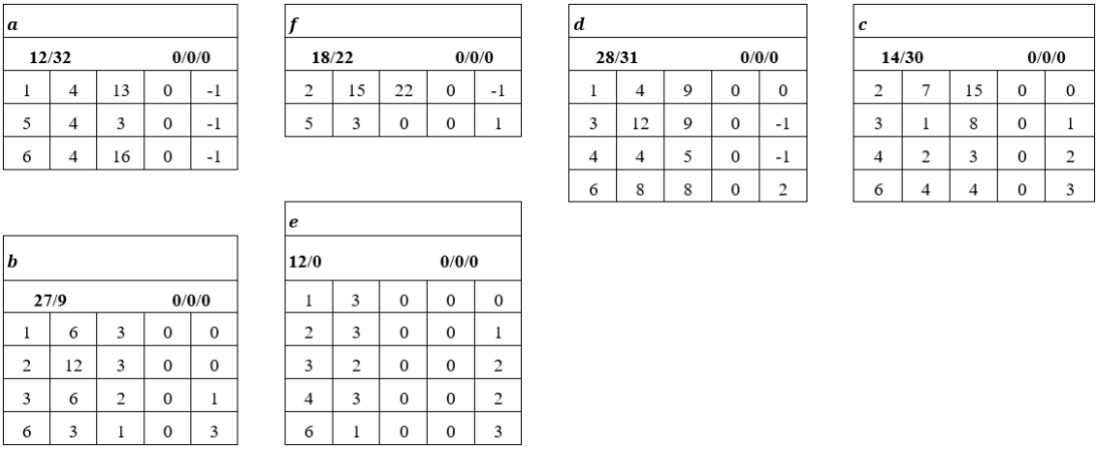
$$\text{top-k HUIs} = \{\{c, d, e\} : 37, \{b, d\} : 39, \{b, c, e\} : 39, \{b, d, e\} : 45\}.$$

E. COMPLEXITY ANALYSIS

In this section, we will estimate the complexity of the FTKHUIM algorithm. Let n, m be the number of transactions and the number of entries in the database, respectively. The FTKHUIM algorithm consists of two main stages: the first stage of increasing the threshold and building 1-CUL (steps 1-21) and the second stage of determining the top-k HUIs (Step 22).

From step 1 to step 4, the FTKHUIM algorithm scans the database for the first time to calculate the TU and RIU values which are used for threshold-raising strategies. In the worst case, the complexity is $O(nm)$. From steps 5 to 7, the algorithm scans the database for a second time to reorder the items in the transaction, build the LIU matrix and implement the LIU_E strategy. Because of scanning and sorting, its complexity is $O(nmlogm)$. In steps 8 to 16 the algorithm builds 1-CUL with the complexity of $O(nm)$. From steps 17 to 21, FTKHUIM builds EUCS, CUDM and uses CUD, LIU_LB strategies, so the complexity is $O(nm^2)$. The complexity of the first stage is then $O(nm + nmlogm + nm^2) = O(nm^2)$.

Step 22, ExplorerKHUI is executed, this is a recursive algorithm that will traverse all the itemsets in the search



a

12/32			0/0/0	
1	4	13	0	-1
5	4	3	0	-1
6	4	16	0	-1

f

18/22			0/0/0	
2	15	22	0	-1
5	3	0	0	1

d

28/31			0/0/0	
1	4	9	0	0
3	12	9	0	-1
4	4	5	0	-1
6	8	8	0	2

c

14/30			0/0/0	
2	7	15	0	0
3	1	8	0	1
4	2	3	0	2
6	4	4	0	3

b

27/9			0/0/0	
1	6	3	0	0
2	12	3	0	0
3	6	2	0	1
6	3	1	0	3

e

12/0			0/0/0	
1	3	0	0	0
2	3	0	0	1
3	2	0	0	2
4	3	0	0	2
6	1	0	0	3

FIGURE 1. 1-CUL structures for 1-itemsets.

space. In the worst case, when all items are potential candidates, the number of itemsets to consider is $l = 2^m - 1$.

As such, in the worst case the complexity of the algorithm is $O(nm^2 + l)$ and is equivalent to the complexity of the HMiner algorithm. The execution time of the algorithm is much lower and depends on the parameters as well as the efficiency of the threshold-raising strategies and the search space pruning strategies.

Similarly, we also estimate the complexity of the TKEH and THUI algorithms. In general, the complexity of these algorithms is the same. The number of candidates pruned during the manipulation of the threshold-raising and pruning strategies will show the effectiveness of the proposed algorithm.

V. PERFORMANCE EVALUATION

To evaluate the feasibility of the FTKHUIM algorithm, we implemented it in Java on a Dell Latitude 7490 computer with a 2.6GHz Core i7 processor, 16GB memory, and Windows 10 operating system. The algorithm has also been used in experiments on datasets published on the SPMF website [43], including both sparse and dense databases. The detailed properties of these databases are presented in Table 10. During the experiment, the results are compared with the state-of-the-art methods that have been published recently on the same topic – mining top-kHUIs in terms of both running time and memory consumption – namely the TKEH [32] and THUI [33] algorithms.

A. THE EFFECTIVENESS OF THE RTU STRATEGY

The TKEH and THUI algorithms apply the RUI threshold-raising strategy to determine the minimum utility value δ before proceeding to remove unpromising items. Meanwhile, the FTKHUIM algorithm applies both RTU and RIU strategies before removing unpromising items from the database. To evaluate the effectiveness of the RTU strategy, the study makes an experiment to measure the achieved value and the number of promising items that the algorithm retains to exploit in the FTKHUIM algorithm in three cases: Using RTU in conjunction with RIU, using RIU only, and using RTU only. The results are presented in Table 11.

TABLE 10. Dataset characteristics.

Dataset	Transactions	Items	Avg. Trans. Length	Density
Chainstore	1,112,949	46,086	7.3	0.00016
Retail	88,162	16,470	10.3	0.00063
Fruithut	181,970	1,265	3.6	0.0028
Footmart	4,141	1,559	4.4	0.0028
Accidents	340,183	468	33.8	0.0722
Mushroom	8,124	119	23.0	0.1933
Connect	67,557	129	43.0	0.3333
Chess	3,196	75	37.0	0.4933

The Chainstore database is characterized by a large number of items, but the ratio of the average length of transactions to the number of transactions of Chainstore is very small (only less than 0.0007%), so the RTU strategy is not more effective than the RIU strategy. Because the number of transactions is very large compared to the average length, the frequency of occurrence of items in the database is many times larger than the length of the transaction (more than 152,000 times). This results in the utilities of single items far outweighing the utilities of transactions, making the RIU strategy more efficient than the RTU strategy.

However, for the other databases where the number of items is small, especially when the number of items is less than the number K, then the RIU strategy cannot increase the initial threshold for the mining process. Therefore, using only the RIU strategy, the minimum utility threshold δ remains at 1 or increases insignificantly. In these cases in particular, the RTU strategy is much more effective in increasing the

TABLE 11. Evaluation of the effectiveness of the RTU strategy.

k	δ			Promising items			δ			Promising items		
	RTU + RIU	RIU	RTU	RTU + RIU	RIU	RTU	TU + RIU	RIU	RTU	RTU + RIU	RIU	RTU
	Chainstore						**Retail**					
500	641529	641529	32204	11824	11824	32585	5115	5115	759	6668	6668	12415
1000	378012	378012	25926	15249	15249	34059	2956	2956	669	8266	8266	12797
2000	223041	223041	22979	18750	18750	34834	1552	1552	574	10275	10275	13225
5000	98767	98767	21904	24529	24529	35145	448	444	448	13833	13854	13833
10000	45524	45524	16197	30105	30105	36969	349	88	349	14369	16027	14369
k	**Fruithut**						**Footmart**					
500	30100	30100	8984	1033	1033	1191	9272	9272	4887	1557	1557	1558
1000	7792	4497	7792	1199	1230	1199	5544	5544	4114	1558	1558	1558
2000	6730	1	6730	1203	9998	1203	2854	1	2854	1559	1559	1559
5000	5263	1	5263	1223	9998	1223	1	1	1	1559	1559	1559
10000	4136	1	4136	1233	9998	1233	1	1	1	1559	1559	1559
k	**Accidents**						**Mushroom**					
500	849	1	849	379	468	379	537	1	537	119	119	119
1000	826	1	826	380	468	380	504	1	504	119	119	119
2000	803	1	803	380	468	380	466	1	466	119	119	119
5000	769	1	769	386	468	386	395	1	395	119	119	119
10000	741	1	741	391	468	391	1	1	1	119	119	119
k	**Connect**						**Chess**					
500	957	1	957	129	129	129	755	1	755	74	75	74
1000	935	1	935	129	129	129	712	1	712	74	75	74
2000	911	1	911	129	129	129	648	1	648	74	75	74
5000	874	1	874	129	129	129	1	1	1	75	75	75
10000	840	1	840	129	129	129	1	1	1	75	75	75

initial threshold δ. For example, with the Fruithut database when k = 1, 000, if applying the RIU strategy the value of δ only reaches 4,497 but with RTU the δ increases to 7,792 and this helps the number of promising items only 1,199 compared to 1,230. Therefore, the performance of the algorithm is improved. Similar to the other approaches, due to the small number of items compared to the desired number K, the RIU strategy will not be able to eliminate any unpromising items. In the Accidents and Fruithut databases, the number of promising items is reduced to 20% and 800% when the RTU strategy is applied, respectively.

With our proposal, the RTU strategy will not be more efficient than the RIU strategy if the number of transactions in the database is lower than K, although this is rarely the case. Furthermore, to handle this problem and be able to take advantage of both strategies, the algorithm used a combination of these two strategies in the first stage of the FTKHUIM algorithm to get the best possible value for δ.

B. THE EFFECTIVENESS OF THE GLOBAL STORAGE STRUCTURE

The runtime and space consumption of the top-k HUIs mining algorithms are highly dependent on the minimum utility threshold δ. The higher the value of δ is, the smaller the search space and the more the runtime is optimized. To evaluate the effectiveness of using a global storage structure, this

study also conducted an experiment on two separate cases: using and not using the global storage structure, namely priorityQueue. The experiment records the 05 values of the minimum utility threshold at different points when every threshold-raising strategy finishes, namely $\delta 1$ to $\delta 5$. The experimental results are presented in Table 12.

From the results shown in Table 12, it is easy to see that with all test cases and test databases the global priorityQueue structure always makes the minimum utility threshold increase faster and higher compared to other approaches. This result is a consequence of the algorithm using effective utility values higher than the current minimum utility threshold detected in the applied pruning strategies. As such, when a new candidate satisfying δ is found, the algorithm will immediately consider whether it is possible to raise the current threshold without waiting to find enough k values for local storage. Local storage is a common priority, and is only initialized each time a new threshold-raising strategy is applied.

With two very sparse databases, Chainstore and Retail, the δ value before calling the ExplorekHUI to determine the top-k HUIs increased by 149% and 129%, respectively. This is the basis to help the algorithm eliminate more unpromising itemsets. Similar results were found with the Fruithut and Foodmart databases, with δ growth rate being 128% and 157%, respectively. In many stages the δ even increased by more than 154% and 157% for Fruithut and Foodmart.

With the Accidents and Mushroom databases, which are quite dense, using a global storage structure gives relatively good results in experimental cases. The minimum utility threshold increases from 123% to 171% when the global priorityQueue is applied to the Mushroom database. Similarly, in experimenting with Accidents the minimum threshold value increases from 109% to 123%, and when $k = 5000$ it increases to 941%. In addition, the δ value sometimes increases more than 18 times compared to when not applying the global priorityQueue. With two databases, Connect and Chess, the application of global priorityQueue is more effective in terms of intermediate thresholding strategies, such as stage 4 increased from 179% to 259% in Connect and from 156% to 370% in Chess. However, after the final thresholding strategy is applied the δ value is still larger, but not by too much.

In summary, by using the global storage structure during threshold-raising, the algorithm finds the minimum utility δ is higher. This results in a larger number of candidates being pruned in the search process.

C. RUNTIME

According to the results in Figure 2 and Figure 3, it is not difficult to see that the FTKHUIM algorithm performs more efficiently in terms of time than the two other approaches. In most cases, for all k values the execution time of the FTKHUIM is relatively stable and very fast. Meanwhile, with TKEH and THUI, the larger the k value is, the longer the

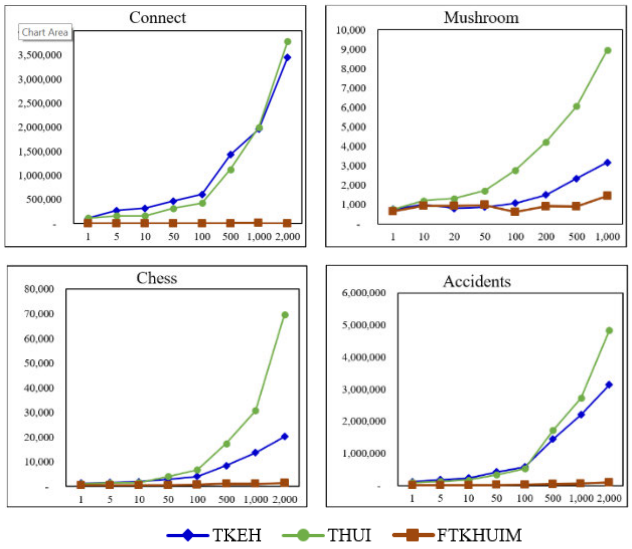


FIGURE 2. The runtime of the algorithms on dense datasets.

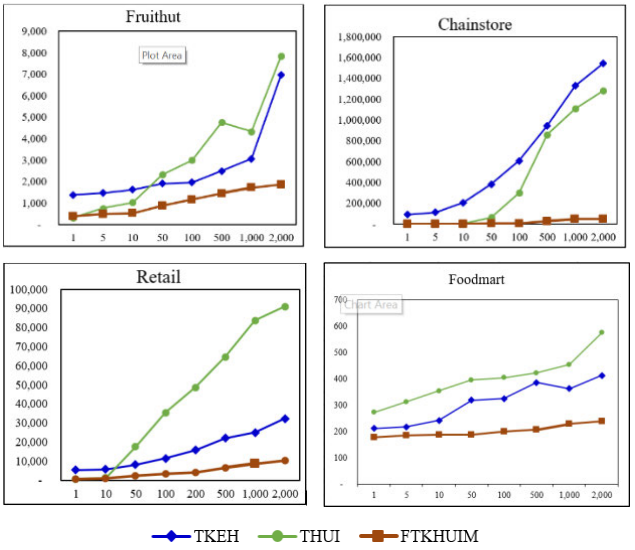


FIGURE 3. The runtime of the algorithms on sparse datasets.

running time. This is because when the database is scanned for the first time in the FTKHUIM algorithm, the threshold δ is raised by the TU values of the transactions. This threshold is initialized earlier and faster than with the other two algorithms, and pruning strategies can eliminate more candidates. In addition, the technique of reusing the utility values of the previous threshold-raising strategies to apply with the next strategy makes the δ value increase more efficiently. For this reason, more candidates will be eliminated, and faster, in the top-kHUIs.

In addition, many effective pruning strategies are applied in the FTKHUIM algorithm, like LA-Prune, C-Prune, and so on. This makes the execution time of our algorithm significantly shorter compared to that of the other two algorithms, TKEH and THUI. For dense databases, specifically

TABLE 12. Evaluation of the effectiveness of the global PriorityQueue.

k	Global structure	$\delta 1$	$\delta 2$	$\delta 3$	$\delta 4$	$\delta 5$	$\delta 1$	$\delta 2$	$\delta 3$	$\delta 4$	$\delta 5$
		Chainstore					**Retail**				
500	NO	641529	641529	641529	641529	641529	5115	5115	5115	8324	8324
	YES	641529	647860	648441	849600	849600	5115	5220	5220	10725	10725
1,000	NO	378012	378012	378012	378012	378012	2956	2956	2956	5478	5478
	YES	378012	381444	381807	562989	562989	2956	3016	3016	7004	7004
2,000	NO	223041	223041	223041	251459	251459	1552	1552	1552	3585	3585
	YES	223041	225050	225594	369265	369265	1552	1572	1578	4517	4517
5,000	NO	98767	98767	98767	152308	152308	448	448	449	1932	1932
	YES	98767	99882	100140	210145	210145	448	462	669	2390	2390
k		**Fruithut**					**Foodmart**				
500	NO	30100	30100	30100	348214	348214	9272	9272	9272	9272	9272
	YES	30100	42960	42984	412358	412358	9272	9272	9273	9273	9273
1,000	NO	7792	7792	7934	222840	222840	5544	5544	5544	5544	5544
	YES	7792	9353	12234	257950	257950	5544	5544	6175	6175	6175
2,000	NO	6730	6730	6798	127330	127330	2854	2854	2854	2854	2854
	YES	6730	6730	8019	145355	145355	2854	2854	4366	4478	4478
5,000	NO	5263	5263	5320	51397	51397	1	1	1	1998	1998
	YES	5263	5263	5746	56810	56810	1	1	1	2610	2610
k		**Accidents**					**Mushroom**				
500	NO	849	7176	7176	2809946	10570639	537	537	537	42047	42047
	YES	849	131244	131244	4352487	11537483	537	537	576	51556	63530
1,000	NO	826	826	838	1456978	7685142	504	504	504	19990	19990
	YES	826	826	1113	1947852	8578698	504	504	521	24334	29122
2,000	NO	803	803	809	690011	4549270	466	466	466	5164	5164
	YES	803	803	832	855666	5583677	466	466	475	7061	8848
5,000	NO	769	769	771	152033	152033	395	395	395	395	395
	YES	769	769	779	183507	1431129	395	395	401	480	487
k		**Connect**					**Chess**				
500	NO	957	370590	370590	1240779	14407474	755	6788	6788	51445	492910
	YES	957	555818	555818	2315950	14443633	755	11490	11490	80309	495703
1,000	NO	935	935	935	599661	13927632	712	712	712	22111	447661
	YES	935	935	984	1108080	13934380	712	712	797	38843	449437
2,000	NO	911	911	911	198919	13238185	648	648	648	2727	389822
	YES	911	911	929	355939	13250121	648	648	698	10074	392332
5,000	NO	874	874	874	8108	12078011	1	1	1	1	290834
	YES	874	874	882	20991	12091665	1	1	1	653	293366

the Connect dataset, with k = 1, 000 and k = 2, 000, the runtime of FTKHUIM is faster than TKEH and THUI by over 500 and 1,000 times, respectively. For the Chess database, FTKHUI is also 8, 10, and 13 times faster than TKEH and 17, 30, and 68 times faster than THUI at k = 500, 1, 000, and 2, 000. For the Accidents database in particular, our proposed algorithm is 300 to 400 times faster when k is 500 or bigger. Based on this, the threshold-raising strategies and proposed algorithm achieved exceptionally good results on dense databases.

Although for sparse databases like Chainstore, Fruithut, Retail, and Footmart, the FTKHUIM algorithm is not much better than TKEH and THUI, its runtime is always shorter. In particular, the larger the value of k is, the longer the execution time of TKEH and THUI. Sometimes, they are two to 10 times – even up to 20 times – more than FTKHUIM's. For example, in Chainstore, one of the large and very sparse databases, if k is less than 10 the execution time of FTKHUI is twice as fast as that of THUI, but it is 40 to 80 times faster than that of TKEH. And the larger the value of k is, the faster FTKHUIM always is, being at least 10 times and up to 30 times faster than both algorithms. Similar results are found when we carry out experiments on three other databases: Fruithut, Retail, and Footmart.

This result is reasonable, because FTKHUIM adopts the threshold-raising strategy using the TU value at the beginning of the algorithm. This strategy increases the threshold value δ from 0 to a higher value. Meanwhile, THUI and TKEH do not use this strategy, and so the δ value is just initialized at 0. As these databases are sparse, the number of closed transactions is low, so applying the RCUL structure in the algorithm does not achieve any great efficiency, and the search speed is slower than the search speed of dense databases.

D. MEMORY USAGE

Figure 4 shows the memory consumed for dense datasets with all three experimental algorithms. For Connect, the memory usage of the FTKHUIM algorithm is better than with the two other approaches, especially for THUI. Using a lot of complex structures to store the information needed for the pruning and threshold-raising strategies is the main reason why THUI's memory usage is two or three times more than that of FTKHUIM and TKEH. Meanwhile, the FTKHUIM algorithm has pruned a large number of candidates and significantly reduced the search space. Moreover, in the RCUL structure many closed and duplicate transactions are merged to optimize the memory usage of FTKHUIM. Since testing on Mushroom, the memory usage of FTKHUIM is still the best in all cases (except for k = 50), and is always less than 100MB. In many cases, the memory consumption of the THUI and TKEH algorithms is two to four times more than that of the proposed algorithm. For the Chess dataset, THUI's memory usage is less at thresholds of 50, 100, 500, 1,000, and 2,000. FTKHUIM consumes less memory on smaller experimental thresholds.

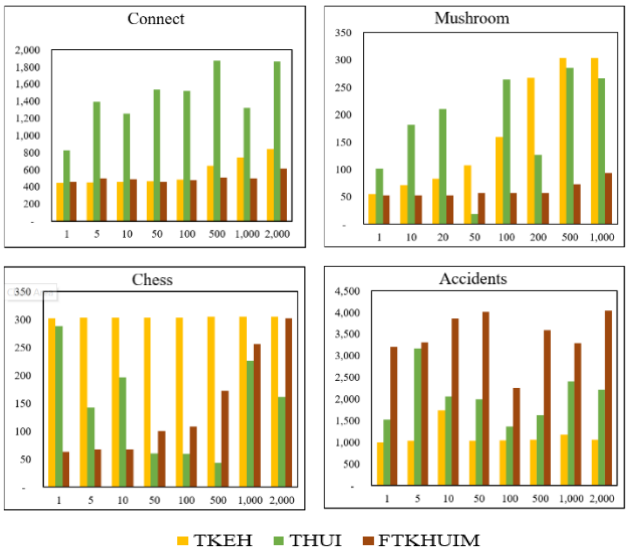


FIGURE 4. Memory usage of the algorithm on dense datasets.

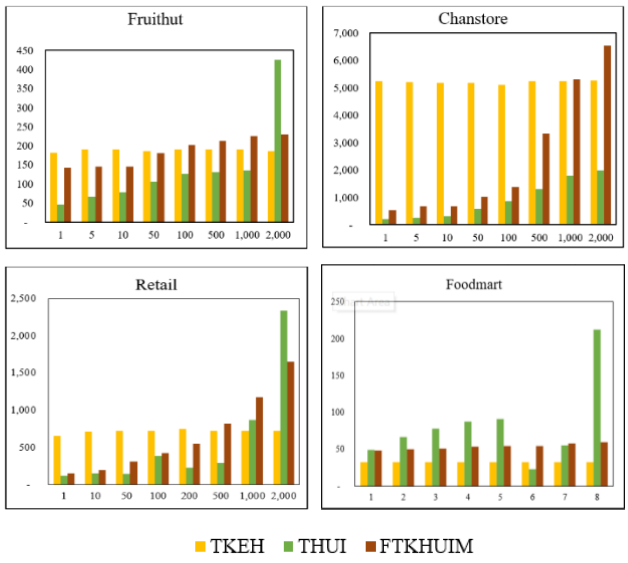


FIGURE 5. Memory usage of the algorithms on sparse datasets.

Particularly for the Accidents dataset, the FTKHUI algorithm is not effective on memory and it always consumes two to three times more memory than the other two algorithms. This is because Accident is a thick database and there are many approved candidates. This leads to storing these candidates to check for duplicates which increases the processing speed, making the amount of memory FTKHUIM needs spike and end up higher than with the other two algorithms.

Figure 5 shows the experimental results of the FTKHUIM, TKEH, and THUI algorithms on sparse datasets such as Fruithut, Chainstore, Retail, and Footmart. For the first three datasets, the memory usage that THUI consumes is always less than that used by the others, but with Footmart TKEH proves to be superior. For the Fruithut dataset, the memory

consumption of FTKHUIM is equivalent to that of the TKEH algorithm and slightly more than that of the THUI algorithm. For the Chainstore dataset, TKEH uses many times more memory than THUI and FTKHUI for small k values. When k is small, the amount of memory that FTKHUIM needs to use is much smaller than that of TKEH, by about 8 to 10 times. For large k (1,000, 2,000), the proposed algorithm has a spike in memory usage. For Retail, when the k value is small, the memory usage of FTKHUIM is also low, equivalent to THUI's and two to four times less than TKEH's.

In sparse databases, the average transaction length is very small, with few items. Therefore, the combination of items in the dataset is low, so the number of itemsets containing many items with high utility is not much. Moreover, the pruning strategies which FTKHUIM applies are effective, so the search space is compacted.

Sometimes transactions containing multiple items also appear in these databases. These transactions have high transaction utilities, so the TU thresholding strategy is applied to make the initial threshold higher in the first steps of the proposed algorithm. This is the reason why many candidates are eliminated at the outset. Because FTKHUIM must save the considered candidates to avoid duplicates, it is not more optimal concerning memory use than THUI in the test cases.

With regard to the Footmart dataset, the storage space that TKEH uses is the most optimal, while THUI is the least efficient, and even when $k = 2,000$ the amount of memory THUI needs is five times higher than with the other two algorithms. The FTKHUIM algorithm is stable and uses 40 to 60MB.

VI. CONCLUSIONS AND FUTURE WORK

In this study, we have introduced a strategy to raise the threshold to determine the initial minimum threshold for the top-k HUIM problem. The research considers each transaction in the database as an itemset and its transaction utility (TU) as a candidate's utility that can be useful for increasing the threshold. Therefore, TU values were computed and applied to update the threshold for the top-k HUIM from scanning the database for the first time. In other algorithms, because raising-threshold strategies are applied they have to re-determine k number of utility values that satisfy the current threshold. If k is a large value, this task also needs much runtime, and the algorithm needs much more time. To address this issue we build a global storage structure for utility values and use it in all the raising-threshold strategies applied throughout the mining process. With this technique, the algorithm's performance gets better because the threshold is increased more rapidly. Finally, the FTKHUIM algorithm is introduced to exploit the top-k HUIs efficiently in terms of both runtime and memory usage. The FTKHUIM algorithm uses the CUL storage structure, combines many thresholding strategies such as TU, RUI, LIU, COD, and COV, and applies search space pruning strategies such as C-prune, U- prune, LA-prune, EUCP, and so on to exploit top-k HUIs efficiently. In terms of time, the algorithm is very good on both dense and sparse datasets, and especially effective on dense datasets.

In future studies, we will focus more on optimizing memory usage for top-k HUIM on sparse databases. At the same time, our research will improve the algorithm so that it can be used on dynamic databases and negative utility databases.

REFERENCES

[1] H. M. Huynh, L. T. T. Nguyen, B. Vo, A. Nguyen, and V. S. Tseng, "Efficient methods for mining weighted clickstream patterns," *Exp. Syst. Appl.*, vol. 142, Mar. 2020, Art. no. 112993, doi: 10.1016/j.eswa.2019.112993.

[2] B. E. Shie, H. F. Hsiao, V. S. Tseng, and P. S. Yu, "Mining high utility mobile sequential patterns in mobile commerce environments," in *Proc. Int. Conf. Database Syst. Advanced Appl.* (Lecture Notes in Computer Science), vol. 6587, 2011, pp. 224–238, doi: 10.1007/978-3-642-20149-3_18.

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

[4] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using FP-trees," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 10, pp. 1347–1362, Oct. 2005, doi: 10.1109/TKDE.2005.166.

[5] B. Vo, T. Le, F. Coenen, and T.-P. Hong, "Mining frequent itemsets using the N-list and subsume concepts," *Int. J. Mach. Learn. Cybern.*, vol. 7, no. 2, pp. 253–265, Apr. 2016, doi: 10.1007/s13042-014-0252-2.

[6] B. Vo, T. Le, T.-P. Hong, and B. Le, "Fast updated frequent-itemset lattice for transaction deletion," *Data Knowl. Eng.*, vols. 96–97, pp. 78–89, Mar. 2015, doi: 10.1016/j.datak.2015.04.006.

[7] Y. Liu, W. K. Liao, and A. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," in *Proc. Pacific–Asia Conf. Knowl. Discovery Data Mining* (Lecture Notes in Computer Science), vol. 3518, 2005, pp. 689–695, doi: 10.1007/11430919_79.

[8] B. Le, H. Nguyen, and B. Vo, "An efficient strategy for mining high utility itemsets," *Int. J. Intell. Inf. Database Syst.*, vol. 5, no. 2, pp. 164–176, Mar. 2011. [Online]. Available: https://www.inderscienceonline.com/doi/abs/10.1504/IJIIDS.2011.03897

[9] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong, and Y.-K. Lee, "HUC-prune: An efficient candidate pruning technique to mine high utility patterns," *Int. J. Speech Technol.*, vol. 34, no. 2, pp. 181–198, Apr. 2011, doi: 10.1007/s10489-009-0188-5.

[10] 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.

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

[12] J. Liu, K. Wang, and B. C. M. 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.

[13] M. Liu and J. Qu, "Mining high utility itemsets without candidate generation," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manag.*, New York, NY, USA, Oct. 2012, p. 55, doi: 10.1145/2396761.2396773.

[14] 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 *Proc. Int. Symp. Methodologies Intell. Syst.* (Lecture Notes in Computer Science), vol. 8502, 2014, pp. 83–92, doi: 10.1007/978-3-319-08326-1_9.

[15] 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, May 2017, doi: 10.1007/s10115-016-0989-x.

[16] 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, May 2017, doi: 10.1007/s10115-016-0986-0.

[17] S. Krishnamoorthy, "HMiner: Efficiently mining high utility itemsets," *Exp. Syst. Appl.*, vol. 90, pp. 168–183, Dec. 2017, doi: 10.1016/j.eswa.2017.08.028.

[18] P. Fournier-Viger, J. C.-W. Lin, T. Gueniche, and P. Barhate, "Efficient incremental high utility itemset mining," in *Proc. ASE BigData SocialInformatics*, Oct. 2015, pp. 1–6, doi: 10.1145/2818869.2818887.
[19] L. T. T. Nguyen, P. Nguyen, T. D. D. Nguyen, B. Vo, P. Fournier-Viger, and V. S. Tseng, "Mining high-utility itemsets in dynamic profit databases," *Knowl.-Based Syst.*, vol. 175, pp. 130–144, Jul. 2019, doi: 10.1016/j.knosys.2019.03.022.
[20] P. Fournier-Viger, Y. Zhang, J. C.-W. Lin, D.-T. Dinh, and H. Bac Le, "Mining correlated high-utility itemsets using various measures," *Log. J. IGPL*, vol. 28, no. 1, pp. 19–32, Jan. 2020, doi: 10.1093/jigpal/jzz068.
[21] W. Gan, J. C.-W. Lin, P. Fournier-Viger, H.-C. Chao, and H. Fujita, "Extracting non-redundant correlated purchase behaviors by utility measure," *Knowl.-Based Syst.*, vol. 143, pp. 30–41, Mar. 2018, doi: 10.1016/j.knosys.2017.12.003.
[22] W. Gan, J. C.-W. Lin, H.-C. Chao, H. Fujita, and P. S. Yu, "Correlated utility-based pattern mining," *Inf. Sci.*, vol. 504, pp. 470–486, Dec. 2019, doi: 10.1016/j.ins.2019.07.005.
[23] B. Vo, L. V. Nguyen, V. V. Vu, M. T. H. Lam, T. T. M. Duong, L. T. Manh, T. T. T. Nguyen, L. T. T. Nguyen, and T.-P. Hong, "Mining correlated high utility itemsets in one phase," *IEEE Access*, vol. 8, pp. 90465–90477, 2020, doi: 10.1109/ACCESS.2020.2994059.
[24] P. Fournier-Viger, J. C. W. Lin, Q. H. Duong, and T. L. Dam, "FHM+: Faster high-utility itemset mining using length upper-bound reduction," in *Proc. Int. Conf. Ind., Eng. Appl. Appl. Intell. Syst.* (Lecture Notes in Computer Science), vol. 9799, 2016, pp. 115–127, doi: 10.1007/978-3-319-42007-3_11.
[25] L. T. T. Nguyen, V. V. Vu, M. T. H. Lam, T. T. M. Duong, L. T. Manh, T. T. T. Nguyen, B. Vo, and H. Fujita, "An efficient method for mining high utility closed itemsets," *Inf. Sci.*, vol. 495, pp. 78–99, Aug. 2019, doi: 10.1016/J.INS.2019.05.006.
[26] T. D. D. Nguyen, L. T. T. Nguyen, L. Vu, B. Vo, and W. Pedrycz, "Efficient algorithms for mining closed high utility itemsets in dynamic profit databases," *Exp. Syst. Appl.*, vol. 186, Dec. 2021, Art. no. 115741, doi: 10.1016/J.ESWA.2021.115741.
[27] C. W. Wu, B.-E. Shie, V. S. Tseng, and P. S. Yu, "Mining top-k high utility itemsets," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2012, pp. 78–86, doi: 10.1145/2339530.2339546.
[28] V. S. Tseng, C.-W. Wu, P. Fournier-Viger, and P. S. Yu, "Efficient algorithms for mining top-k high utility itemsets," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 1, pp. 54–67, Jan. 2016, doi: 10.1109/TKDE.2015.2458860.
[29] H. Ryang and U. Yun, "Top-k high utility pattern mining with effective threshold raising strategies," *Knowl.-Based Syst.*, vol. 76, pp. 109–126, Mar. 2015, doi: 10.1016/J.KNOSYS.2014.12.010.
[30] J. Liu, X. Zhang, B. C. M. Fung, J. Li, and F. Iqbal, "Opportunistic mining of top-n high utility patterns," *Inf. Sci.*, vol. 441, pp. 171–186, May 2018, doi: 10.1016/J.INS.2018.02.035.
[31] Q.-H. Duong, B. Liao, P. Fournier-Viger, and T.-L. Dam, "An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies," *Knowl.-Based Syst.*, vol. 104, pp. 106–122, Jul. 2016, doi: 10.1016/j.knosys.2016.04.016.
[32] K. Singh, S. S. Singh, A. Kumar, and B. Biswas, "TKEH: An efficient algorithm for mining top-k high utility itemsets," *Int. J. Speech Technol.*, vol. 49, no. 3, pp. 1078–1097, Mar. 2019, doi: 10.1007/s10489-018-1316-x.
[33] S. Krishnamoorthy, "Mining top-k high utility itemsets with effective threshold raising strategies," *Exp. Syst. Appl.*, vol. 117, pp. 148–165, Mar. 2019, doi: 10.1016/J.ESWA.2018.09.051.
[34] N. N. Pham, Z. Kominkova Oplatkova, H. M. Huynh, and B. Vo, "Mining top-k high utility itemset using bio-inspired algorithms," in *Proc. IEEE Workshop Complex. Eng. (COMPENG)*, Jul. 2022, pp. 1–5, doi: 10.1109/COMPENG50184.2022.9905433.
[35] M. Ashraf, T. Abdelkader, S. Rady, and T. F. Gharib, "TKN: An efficient approach for discovering top-k high utility itemsets with positive or negative profits," *Inf. Sci.*, vol. 587, pp. 654–678, Mar. 2022, doi: 10.1016/J.INS.2021.12.024.
[36] W. Gan, S. Wan, J. Chen, C.-M. Chen, and L. Qiu, "TopHUI: Top-k high-utility itemset mining with negative utility," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2020, pp. 5350–5359, doi: 10.1109/BIGDATA50022.2020.9378288.
[37] Z. H. Deng, "An efficient structure for fast mining high utility itemsets," *Appl. Intell.*, vol. 48, no. 9, pp. 3161–3177, Sep. 2018, doi: 10.1007/s10489-017-1130-x.
[38] J. M.-T. Wu, J. C.-W. Lin, and A. Tamrakar, "High-utility itemset mining with effective pruning strategies," *ACM Trans. Knowl. Discovery Data*, vol. 13, no. 6, pp. 1–22, Nov. 2019, doi: 10.1145/3363571.
[39] X. Han, X. Liu, J. Li, and H. Gao, "Efficient top-k high utility itemset mining on massive data," *Inf. Sci.*, vol. 557, pp. 382–406, May 2021, doi: 10.1016/J.INS.2020.08.028.
[40] J. C.-W. Lin, P. Fournier-Viger, and W. Gan, "FHN: An efficient algorithm for mining high-utility itemsets with negative unit profits," *Knowl.-Based Syst.*, vol. 111, pp. 283–298, Nov. 2016, doi: 10.1016/J.KNOSYS.2016.08.022.
[41] R. Sun, M. Han, C. Zhang, M. Shen, and S. Du, "Mining of top-k high utility itemsets with negative utility," *J. Intell. Fuzzy Syst.*, vol. 40, no. 3, pp. 5637–5652, Mar. 2021, doi: 10.3233/JIFS-201357.
[42] B. Le, H. Nguyen, T. A. Cao, and B. Vo, "A novel algorithm for mining high utility itemsets," in *Proc. 1st Asian Conf. Intell. Inf. Database Syst.*, Apr. 2009, pp. 13–17, doi: 10.1109/ACIIDS.2009.55.
[43] 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. [Online]. Available: https://dl.acm.org/doi/10.5555/2627435.2750353

VINH V. VU received the M.S. degree in information technology from the Ho Chi Minh City University of Technology, Ho Chi Minh, Vietnam, in 2019. His research interests include association rules, classification, text processing, mining high utility itemset, and preserving privacy in data mining.

MI T. H. LAM received the M.S. degree in the mathematical foundation for computers and computing systems from the University of Science, Vietnam National University, Ho Chi Minh, Vietnam, in 2013. She is currently a Lecturer with the Ho Chi Minh City University of Industry and Trade, Vietnam. Her research interests include association rules, classification, text processing, and mining high utility itemset in data mining.

THUY T. M. DUONG received the M.S. degree in computer science from the University of Science, Vietnam National University, Ho Chi Minh, Vietnam, in 2013. She is currently a Lecturer with the Ho Chi Minh City University of Industry and Trade, Vietnam. Her research interests include association rules, classification, text processing, and mining high utility itemset in data mining.

LY T. MANH received the M.S. degree in information systems from Military Technical Academy, Hanoi, Vietnam, in 2010. She is currently a Lecturer with the Ho Chi Minh City University of Industry and Trade, Vietnam. Her research interests include association rules, classification, text processing, and mining high utility itemset in data mining.

THUY T. T. NGUYEN received the B.S. degree in information technology from the University of Science, Vietnam National University, Ho Chi Minh, Vietnam, in 2003, and the M.S. degree in business administration from the University of Economics Ho Chi Minh City, in 2013. She is currently a Lecturer with the Ho Chi Minh City University of Industry and Trade, Vietnam. Her research interests include association rules, classification, text processing, and mining high utility itemset in data mining.

LE V. NGUYEN received the M.S. degree in data transmission and computer networks from the Posts and Telecommunications Institute of Technology, Vietnam, in 2011. He is currently a Lecturer with the Ho Chi Minh City University of Industry and Trade, Vietnam. His research interests include association rules, classification, text processing, and mining high utility itemset in data mining.

UNIL YUN received the Ph.D. degree in computer science from Texas A&M University, College Station, TX, USA, in 2005. He is currently a Full Professor with the Department of Computer Engineering, Sejong University, Seoul, South Korea. He has published more than 200 research articles in refereed journals and international conferences. His research interests include data mining, information retrieval, database systems, artificial intelligence, and digital libraries. He is an Associate Editor (Editorial Board Member) of *Knowledge-Based Systems*, *PLoS ONE*, and *Electronics*.

VACLAV SNASEL (Senior Member, IEEE) has given more than ten plenary lectures and conference tutorials in his research areas. He has authored/coauthored several refereed journals/conference papers and book chapters. He has published more than 400 articles (147 is recorded at Web of Science). He has supervised many Ph.D. students from Czech Republic, Jordan, Yemen, Slovakia, Ukraine, and Vietnam. His research and development experience includes over 25 years in the industry and academia. He works in a multi-disciplinary environment involving artificial intelligence, multidimensional data indexing, conceptual lattice, information retrieval, semantic web, knowledge management, data compression, machine intelligence, neural networks, web intelligence, data mining, and applied to various real-world problems.

BAY VO received the Ph.D. degree in computer science from the University of Science, Vietnam National University, Ho Chi Minh, Vietnam, in 2011. He is currently an Associate Professor with the Ho Chi Minh City University of Technology, Vietnam. His research interests include association rules, classification, mining in the incremental database, distributed databases and privacy-preserving in data mining, and soft computing.