Applied Intelligence
https://doi.org/10.1007/s10489-018-1316-x

# TKEH: an efficient algorithm for mining top-k high utility itemsets

**Kuldeep Singh[1] · Shashank Sheshar Singh[1] · Ajay Kumar[1] · Bhaskar Biswas[1]**



**Abstract**

High utility itemsets mining is a subfield of data mining with wide applications. Although the existing high utility itemsets mining algorithms can discover all the itemsets satisfying a given minimum utility threshold, it is often difficult for users to set a proper minimum utility threshold. A smaller minimum utility threshold value may produce a huge number of itemsets, whereas a higher one may produce a few itemsets. Specification of minimum utility threshold is difficult and time-consuming. To address these issues, top-k high utility itemsets mining has been defined where $k$ is the number of high utility itemsets to be found. In this paper, we present an efficient algorithm (named TKEH) for finding top-k high utility itemsets. TKEH utilizes transaction merging and dataset projection techniques to reduce the dataset scanning cost. These techniques reduce the dataset when larger items are explored. TKEH employs three minimum utility threshold raising strategies. We utilize two strategies to prune search space efficiently. To calculate the utility of items and upper-bounds in linear time, TKEH utilizes array-based utility technique. We carried out some extensive experiments on real datasets. The results show that TKEH outperforms the state-of-the-art algorithms. Moreover, TKEH always performs better for dense datasets.

**Keywords** High utility itemsets · Utility mining · Itemset mining · Top-k itemset mining · Threshold raising strategies

## 1 Introduction

High utility itemsets (HUIs) mining is the subfield of Frequent itemset mining (FIM). In FIM, an item is frequent if that item has the frequency more than minimum support threshold [1]. FIM uses anti-monotonic (downward closure) property to reduce the search space. This property states that if an item is infrequent, then none of it's supersets are frequent. FIM algorithms suffer from two main drawbacks. Firstly, FIM algorithms consider that item has a single purchase quantity. For example, if a customer has bought two pens, five pens or ten pens, it is viewed as the same. Secondly, all items have the same importance (e.g., unit profit or price). For example, if a customer has bought an expensive diamond or just an ordinary pen, it is viewed as being equally important. But these assumptions or limitations often do not hold in real applications. In real-life, retailers and corporate managers are interested to find the itemsets which are more profitable. Therefore, FIM does not fulfill the requirements of the user in real-life. To address these issues, HUIs mining field came in the limelight.

HUIs mining algorithms include the quantity and importance (price or profit) of the items. Therefore it is an important research problem in data mining. An itemset is called HUIs if that has more utility value than user-specified minimum utility ($min\_util$) threshold. HUIs mining have many applications such as mobile commerce [17], user behavior analysis [18], website clickstream analysis [3], cross-marketing analysis [24] and cross-marketing analysis in streaming datasets [12]. HUIs mining is harder compared to FIM because HUIs mining algorithms do not follow the anti-monotonic property. Hence, the FIM strategies and methods cannot be directly applied into HUIs mining field. To overcome these issues, Liu et. al proposed $TWU$ (transaction weighted utility) based overestimation method [15]. Several HUIs mining algorithms follow $TWU$ based pruning property such as UP-Growth [21], UP-Growth+ [19], and MU-Growth [26].

✉ Kuldeep Singh
kuldeep.rs.cse13@iitbhu.ac.in

Shashank Sheshar Singh
shashankss.rs.cse16@iitbhu.ac.in

Ajay Kumar
ajayk.rs.cse16@iitbhu.ac.in

Bhaskar Biswas
bhaskar.cse@iitbhu.ac.in

[1] Department of Computer Science and Engineering, Indian Institute of Technology (BHU), Varanasi, 221–005, India

Published online: 25 October 2018

One of the limitations of HUIs mining is to set the $min\_util$ threshold. The users do not know which $min\_util$ threshold is good for their requirements. Specifying the $min\_util$ threshold is very crucial because it directly affects the number of HUIs. If the $min\_util$ threshold is set too high, a few HUIs are found. If the $min\_util$ threshold is set too low, a huge number of HUIs are found. A Few or huge HUIs are not actionable. To solve this problem top-k HUIs based mining was proposed. In top-k HUIs mining value of $k$ is needed instead of $min\_util$ threshold. The users need to specify a parameter $k$ indicating the number of itemsets to be found. Therefore, specifying $k$ is more easier than specifying $min\_util$. However, developing efficient algorithms for mining such itemsets is not an easy task. There are three major challenges discussed below.

1. top-k HUIs mining algorithms consume more time and memory compared to simple HUIs mining algorithms. Hence, a compact dataset storage structure or dataset cost reduction technique is required.
2. $min\_util$ is not given in advance; the algorithm needs to start the from 0 or 1 $min\_util$. Hence, the challenge is how to raise the $min\_util$ automatically without missing any top-k HUIs.
3. Pruning search space is also a big challenge to reduce the candidate itemset because $min\_util$ is always set to 0 or 1. Hence, an efficient pruning strategy is also required.

To address these challenges, an efficient top-k HUIs mining algorithm is needed. The key contributions of this paper are as follows:

- We utilize transaction merging and dataset projection techniques to reduce the dataset scanning cost. These techniques reduce the dataset as larger items are explored.
- We employ three $min\_util$ threshold raising strategies for the $min\_util$ automatically and efficiently.
- We adopt $EUCP$ and $sup$ pruning strategies to search space efficiently.
- We utilize an efficient technique, that is utility array ($UA$), to calculate the utility of items and upper-bounds in linear time.

The rest of the paper is organized as follows. Section 2 describes the related work on top-k high utility mining. Section 3 defines the preliminaries, key definitions, and notations used in this paper. Section 4 presents the efficient dataset scanning techniques, pruning strategies, array-based utility counting techniques and the proposed algorithm. Section 6 provides detailed experimental analysis and results. Section 7 includes the discussion on strategies utilize in the proposed algorithm. Finally, Section 8 gives concluding remarks.

## 2 Related literature

One of the earliest works in HUIs mining field is Two-Phase [15] algorithm which mines the itemsets in two-phases. Two-Phase algorithm presents the $TWU$ based overestimation concept which follows the downward closure property in HUIs mining. In literature, some other algorithms which follow two-phase model to mine HUIs such as U-Mining [23], UP-Growth [21], and UP-Growth+ [19].

Two-phase model based algorithms suffer from two major problems. First they scans the dataset multiple times and second they generate a huge number of candidates. To overcome these limitations, one-phase algorithms are presented. HUI-Miner [14] is one of the earliest works in one-phase algorithms. It presents utility-list structure to store the informations of items. Later on FHM and other one-phase algorithms are proposed such as [7], HUP-Miner [9], EFIM [27] and HMiner [10] algorithms are proposed. Discussed above algorithms suffers from the one main problem, to specify the $min\_util$ threshold which is not an easy task for the end users.

In order to overcome the problem of specifying the $min\_util$, top-k HUIs algorithms have been proposed [4, 5, 11, 16, 20, 22, 25, 28]. Top-k HUIs mining algorithms, initially set the internal $min\_util$ threshold to one or zero. In the next step, the procedures automatically raise the internal threshold using some strategies. Important things about internal threshold raising strategies are that algorithms do not miss any top-k HUIs. And often some strategies are introduced to prune the search space. At last, the algorithms terminate and find top-k HUIs. Top-k HUIs algorithms can be divided broadly into two categories.

### 2.1 Two-phase algorithms

The first and most popular algorithm for mining top-k HUIs is TKU [22] which is an extension of UP-Growth algorithm [19, 21]. TKU follows two-phase model and inherits their useful properties. In the first phase, potential top-k HUIs (PKHUIs) are generated and the algorithm uses four strategies (PE, MD, NU and MC) to raise the internal $min\_util$ threshold and prune the search space. In the second phase, top-k HUIs are identified from the set of PKHUIs that are discovered in first-phase. TKU uses fifth strategy (SE) to sort the candidate itemsets and raise the internal threshold. To improve the performance of TKU, Ryang and Yun proposed an algorithm named REPT [16]. REPT also relies on the UP-Tree structure. REPT improves the performance of TKU by applying additional strategies that raise the border $min\_util$ rapidly. One of the limitations of REPT is that user has to set parameter $N$ to control the effectiveness of the proposed strategy. It is very

**Table 1** An overview of top-k High utility itemsets mining algorithms

| Algorithm | Structure | Pruning strategy | State-of-the-art algorithms | Utility value | Base algorithm |
|---|---|---|---|---|---|
| Two-phase algorithms | | | | | |
| TKU, 2016 [22] | UP-tree | PE, NU, MD, MC & SE | UP-Growth & HUI-Miner | Positive only | UP-Growth [21] |
| REPT, 2015 [16] | UP-tree | PUD, RIU, RSD & SEP | TKU, UP- Growth+ & MU-Growth [26] | Positive only | TKU [22] |
| T-HUDS, 2014 citeT-HUDS | HUDS-tree | PerfixUtil | TKU & HUMPS [2] | Positive only | UP-Growth [21] |
| TUS, 2013 [25] | TUSlist | Pre-insertion | TUSNaive (self) | Positive only | – |
| One-phase algorithms | | | | | |
| TKO, 2016 [20] | utility list | RUC,RUZ & EPB | UP-Growth & HUI-Miner | Positive only | HUI-Miner [14] |
| kHMC, 2016 [5] | utility-list | RIU, CUD, & COV | TKO & REPT | Positive only | FHM [7] |
| TKUL-Miner, 2016 [11] | utility-list | FSD, RUZ, & FCU | TKU, UPGrwoth+, REPT | Positive only | FHM [7] |
| KOSHU, 2017 [4] | utility list | EMPRP, PUP & CE2P | FOSHU | Positive & negative | FOSHU [8] |

tough for end users to choose an appropriate value for $N$ and the choice of $N$ greatly influences the performance of REPT. But still, 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. Table 1 shows the overview of top-k HUIs mining algorithms.

Zihayat et al. proposed an algorithm named T-HUDS for mining top-k HUIs over data streams [28]. T-HUDS proposed a novel over-estimate utility model called PrefixUtil model. This model follows compressed FP-tree like data structure, HUDS-tree and it has two auxiliary lists, maxUtilList and MIUList. These lists are designed to store the information that is needed for computing PrefixUtil and for initializing and dynamically adjusting the internal threshold. Yin et al. have proposed a new algorithm named TUS for mining top-k high utility sequential patterns [25]. It introduced a pre-insertion and a sorting strategy to raise the internal utility threshold. To improve the performance of TUS, three effective strategies are introduced; two for raising the $min\_util$ threshold and one for reducing the search space.

## 2.2 One-phase algorithms

In order to overcome the limitations of two-phase algorithms, one-phase algorithm has been introduced. Tseng et al. proposed an algorithm named TKO to mine top-k HUIs [20] which outperforms TKU and REPT. TKO utilizes the utility-list structure of the HUI-Miner. It presents novel pruning strategies RUC, RUZ and EPB to improve the performance.

kHMC is another one-phase algorithm proposed by Duong et al. which relies on the utility-list structure [5]. kHMC is an extension of the FHM algorithm [7]. It employs two pruning strategies called EUCPT and Transitive Extension Pruning strategy (TEP). EUCPT avoids the join operations for calculating the utilities of itemsets as in FHM. The TEP strategy reduces the search space using a novel upper-bound on the utilities of itemsets. kHMC introduces three strategies named RIU, CUD, and COV to raise internal $min\_util$ effectively. The COV strategy introduces a novel concept of coverage. The concept of coverage can be employed to prune the search space or to raise the threshold in top-k HUIs mining. Moreover, kHMC proposes a new pruning strategy named TEP for reducing the search space. Another utility-list and the EUCS based top-k HUIM algorithm is TKUL-Miner [11]. It adopts several strategies called FSD, RUZ, and FCU to raise the $min\_util$ rapidly and prune the search space effectively.

Recently, an on-shelf based top-k HUIs mining algorithm is proposed, KOSHU [4]. It introduced three pruning strategies to speed-up the execution named EMPRP, PUP

**Table 2** A transactional dataset

| $T_{ID}$ | Transaction |
|---|---|
| $T_1$ | A(2), B(2), D(1), E(3) |
| $T_2$ | B(1), C(5), E(1), F(2) |
| $T_3$ | B(2), C(1), D(3), E(2) |
| $T_4$ | C(2), D(1), E(3) |
| $T_5$ | A(2), F(1) |
| $T_6$ | A(2), B(1), C(4), D(2), E(1) |
| $T_7$ | B(3), C(2), E(2), F(3) |

and CE2P. KOSHU is a top-k version on the FOSHU algorithm [8].

## 3 Preliminaries and problem definition

**Definition 3.1** (**Transaction dataset**) Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of distinct items. A set $X \subseteq I$ is called an itemset. $D = \{T_1, T_2, ..., T_n\}$ be a transaction dataset where each transaction is represented by $T_j \in D$ where $n$ is the total number of transactions in the dataset $D$.

Consider the sample transactional dataset $D$ which is given in Table 2. This dataset contains seven transactions $(T_1, T_2, ..., T_7)$. Transaction $T_1$ indicates that items $A$, $B$, $D$ and $E$ appear with quantity respectively $2, 2, 1$ and $3$. Table 3 indicates the external utility of each item.

**Definition 3.2** (**Internal Utility** $(IU(x, T_j))$) Let $x$ be an item. Internal utility of item $x$ in a transaction $T_j \in D$ is defined as quantity of $x$ in $T_j$.

**Definition 3.3** (**External Utility** $(EU(x))$) Let $x$ be an item. The external utility of item $x$ is defined as $EU(x)$.

**Definition 3.4** (**Utility of an item in a transaction**) Utility of item $x$ is defined as $U(x, T_j) = IU(x, T_j) \times EU(x)$.

**Definition 3.5** (**Utility of an itemset in a transaction**) Let $X$ be an itemset. Utility of an Itemset $X$ is defined as $U(X, T_j) = \sum_{x \in X \wedge X \subseteq T_j} U(x, T_j)$.

**Definition 3.6** (**Utility of an itemset in a dataset** $D$) Utility of an itemset $X$ is defined as $U(X) = \sum_{X \subseteq T_j \in D} U(X, T_j)$. For example, $U(AE) = U(AE, T_1) + U(AE, T_6) = 7 + 5 = 12$.

**Table 3** External utility value

| Item | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
|---|---|---|---|---|---|---|
| External Utility | 2 | 3 | 1 | 4 | 1 | 3 |

**Definition 3.7** (**Transaction Utility**) Transaction utility $T_j$ is defined as $TU(T_j) = \sum_{x \in T_j} U(x, T_j)$.

**Definition 3.8** (**Transaction weighted utility of an itemset** $X$ **in the dataset** $D$) Transaction Weighted utility of an Itemset $X$ in the dataset $D$ is defined as $TWU(X) = \sum_{X \in T_j \wedge T_j \in D} TU(T_j)$.

**Property 3.1** ($TWU$ **based overestimation**) *If the* $TWU$ *value of itemset* $X$ *is greater than or equal to utility value of itemset* $X$*, that is* $TWU(X) \geq U(X)$*, then the itemset is assumed as overestimated.*

**Property 3.2** ($TWU$ **based pruning**) *If the* $TWU$ *value of the itemset* $X$ *is less than user-defined threshold* ($min\_util$)*, that is* $TWU(X) < min\_util$*, then the itemset cannot be included for further processing.*

**Definition 3.9** (**High utility itemset**) An itemset $X$ is called high utility itemset if the $U(X) \geq min\_util$ threshold. Otherwise, the itemset is low utility.

Two-phase based algorithms suffer from multiple dataset scans and generate lots of candidates. To overcome these limitations, one-phase algorithms are proposed. One-phase algorithms are more efficient than two-phase algorithms concerning execution time and memory space. Most of the one-phase algorithms use utility-list based structure and remaining utility pruning strategy to prune the search space [7, 9, 13, 14, 27].

**Definition 3.10** (**Remaining utility of an itemset in a transaction**) The remaining utility of itemset $X$ in transaction $T_j$ denoted by $RU(X, T_j)$ is the sum of the utilities of all the items in $T_j/X$ in $T_j$ where $RU(X, T_j) = \sum_{i \in (X, T_j)} U(i, T)$ [7, 9, 14].

**Definition 3.11** (**Utility-list structure**) The utility-list structure contains three fields, $T_{id}, iutil$, and $rutil$. The $T_{id}$ indicates the transactions containing itemset $X$, $iutil$ indicates the $U(X)$, and the $rutil$ indicates the remaining utility of itemset $RU(X, T_j)$

**Property 3.3** (**Pruning search space using remaining utility**) *For an itemset* $X$*, if the sum of* $U(X) + RU(X)$ *is less than* $min\_util$*, then itemset* $X$ *and all its supersets are low utility itemsets. Otherwise, the itemset is HUIs.*

The detail and proof of remaining utility upper bound ($REU$) based upper bound is given in [14]. The utility-list

based algorithms use the remaining utility based pruning strategies.

**Definition 3.12** (**Top-k high utility itemset**) An itemset $X$ is top-k HUIs if there are less than $k$ items which have the utility larger than the utility of itemset $X$.

## 4 Proposed TKEH algorithm

In this section, we give a step-by-step analysis of the proposed algorithm named TKEH. Section 4.1 describes the search space and shows the techniques to find larger itemsets from items. Section 4.2 describes the EUCS structure that raises the $min\_util$ threshold. Section 4.3 describes the dataset cost reduction techniques to reduce the dataset scanning. Section 4.4 describes the threshold raising techniques. Section 4.5 describes the pruning strategies. Section 4.6 introduces array-based utility counting technique. Finally, Section 4.7 gives the pseudo-code of the proposed algorithm.

### 4.1 The search space

The search space of the top-k HUIs mining problem can be represented as a set-enumeration tree as in [14]. The items can be explored using depth-first-search in the set-enumeration tree starting from the root which is an empty set. We sort the items in increasing order of $TWU$ values that reduce the search space [7, 14]. Some definitions related to exploration of itemsets in the set-enumeration tree are given below.

**Definition 4.1** (**Extension of an item**) Let $\alpha$ be an itemset. The set of items that are used to extend the itemset $\alpha$ is denoted by $E(\alpha)$ and is defined as $E(\alpha) = \{z \mid z \in I \wedge z \succ x, \forall x \in \alpha\}$.

**Definition 4.2** (**Extension of an itemset**) For the itemset $\alpha$, $Z$ is an extension of $\alpha$ that appears in a sub-tree of $\alpha$ in the set-enumeration tree. If $Z = \alpha \cup W$ for an itemset $W \in 2^{E(\alpha)}$. $Z$ is a single-item extension of $\alpha$ that is a child of $\alpha$ in the set-enumeration tree. If $Z = \alpha \cup \{z\}$ for an item $z \in E(\alpha)$.

For example $\alpha = \{C\}$. The set $E(\alpha)$ is $\{B, E\}$. And single-item extensions of $\alpha$ are $\{C, B\}$ and $\{C, E\}$. The itemsets extensions of $\alpha$ is $\{C, B, E\}$.

### 4.2 Concept of co-occurrence structure

In top-k-HUIs mining, a key challenge is to design efficient techniques to raise the $min\_util$. These techniques should be efficient regarding time and memory usage. We employ $EUCST$ (Estimated Utility Co-occurrence Pruning Strategy with Threshold) strategy to raise the $min\_util$ and prune the search space. This strategy is proposed by FHM algorithm [7] and after that is improved by kHMC algorithm [5]. The following paragraph first describes the $EUCS$ structure and then $EUCST$ strategy.

**Table 4** $TWU$ values of items as $\succ$ order

| Item | A | F | D | C | B | E |
|---|---|---|---|---|---|---|
| TWU | 44 | 44 | 67 | 87 | 95 | 104 |

**Definition 4.3** (**EUCS structure**) EUCS structure is a set of triples of the form $(x, y, z) \in I^* \times I^* \times \mathbb{R}^+$. A triple $(x, y, z)$ indicates that $TWU(\{x, y\}) = z$.

In the running example, EUCS is constructed for transaction $T_1$ as shown in Table 8. Consider itemsets $\{A, D\}$ and $\{D, B\}$. The $TWU$ values for these itemsets are respectively $TWU(\{A, D\}) = 37$ and $TWU(\{D, B\}) = 58$.

**Property 4.1** *Let $X$ be an itemset. If $TWU(X) < min\_util$ then for any extension $y$ of $X$, $U(Y, X) < min\_util$.*

The $EUCS$ can be implemented by a triangular matrix as in [7]. We implemented the $EUCS$ using hashmap instead of triangular matrix. The hashmap based implementation is more efficient. The $EUCS$ only stores the values that $TWU \neq 0$. Therefore fewer items are in $EUCS$. The proposed algorithm scans the dataset twice as the other efficient HUIs mining algorithm. The first scan calculates the $TWU$ of each 1-item(s). Then the items are sorted according to the $TWU$ values in non-decreasing order as suggested in [14]. This sorting order can be denoted by $\succ$ order. Table 4 shows the $TWU$ value for the running example as $\succ$ order. The second scan constructs the $EUCS$. Each cell in Table represents $TWU$ values for the itemsets. Table 8 represents $EUCS$ implementation of the running example. The creation of $EUCS$ structure for transaction $T_1$ is shown in Table 5 where $T_1$=$(A, 4)$, $(D, 4)$, $(B, 6)$, $(E, 3)$.

**Table 5** $EUCS$ map for transaction $T_1$

| Item | A | F | D | C | B |
|---|---|---|---|---|---|
| F | | | | | |
| D | 17 | | | | |
| C | | | | | |
| B | 17 | | 17 | | |
| E | 17 | | 17 | | 17 |

**Table 6** $EUCS$ map up-to transaction $T_2$

| Item | A | F | D | C | B |
|---|---|---|---|---|---|
| F | | | | | |
| D | 17 | | | | |
| C | | 15 | | | |
| B | 17 | 15 | 17 | 15 | |
| E | 17 | 15 | 17 | 15 | 32 |

**Table 8** Final $EUCS$ Map

| Item | A | F | D | C | B |
|---|---|---|---|---|---|
| F | | | | | |
| D | 37 | | | | |
| C | | 37 | 50 | | |
| B | 37 | 37 | 58 | 78 | |
| E | 37 | 37 | 67 | 87 | 95 |

For this transaction, tuples are $(A, D, 17)$, $(A, B, 17)$, $(A, E, 17)$, $(D, B, 17)$, $(D, E, 17)$, $(B, E, 17)$ as shown in Table 5. Since none of these values are present in the $EUCS$ map, the values are inserted in it directly. Using the same phenomena, we can create the $EUCS$ for all the transactions. If the $TWU$ is already for the item then we simply add the $TWU$ values and update the $EUCS$. As $T_2 = (F, 6)$, $(C, 5)$, $(B, 3)$, $(E, 1)$. For transaction $T_2$ tuples are $(F, C, 15)$, $(F, B, 15)$, $(F, E, 15)$, $(C, B, 15)$, $(C, E, 15)$, $(B, E, 15)$. Since tuple $(B, E, 15)$ is already available in $EUCS$ map its values are updated as shown in Table 6. The same process is repeated for the next transactions. Table 7 shows all the item without pruning. The itemsets having less $TWU$ value than $min\_util$ (colored in brown in Table 7) are removed from EUCS Table 8 shows the final $EUCS$ for the running example after eliminate the items using Property 4.1 where $min\_util$ is 30. EUCS structure based pruning is called EUCP and is proposed in [7].

## 4.3 Dataset scanning techniques

To reduce the dataset scanning cost, TKEH employs dataset projection and transaction merging techniques.

### 4.3.1 Dataset scanning using projection

The proposed algorithm calculates the utility and upper-bounds of itemsets by scanning the dataset. TKEH creates EUCST and CUDM structure and calculates RIU at the same time. As the dataset can be very large, there is a need to reduce the cost of dataset scanning. Hence, dataset projection is required. We simply observe during the depth-first search for any itemset $\alpha$, all items that do not belong to $E(\alpha)$ can be ignored while scanning the dataset. Hence, we do not calculate the utility and upper bound on their utility of these items such as $x$. A dataset without these items is known as projected dataset.

**Table 7** $EUCS$ Map with all $TWU$

| Item | A | F | D | C | B |
|---|---|---|---|---|---|
| F | 7 | | | | |
| D | 37 | 0 | | | |
| C | 20 | 37 | 50 | | |
| B | 37 | 37 | 58 | 78 | |
| E | 37 | 37 | 67 | 87 | 95 |

**Definition 4.4** (**Projected dataset**) The projection of a transaction $T$ using an itemset $\alpha$ is denoted by $\alpha - T$ and defined as $\alpha - T = \{i \mid i \in T \wedge i \in E(\alpha)\}$. The projection of a dataset $D$ using an itemset $\alpha$ is denoted by $\alpha - D$ and defined as the multi-set $\alpha - D = \{\alpha - T \mid T \in D \wedge \alpha - T \neq \emptyset\}$.

For example, the projected dataset of an itemset $\alpha = D$. The projected dataset $(\alpha - D)$ for item $\alpha$ contains four transactions: $\alpha - T_1 = \{B, E\}, \alpha - T_3 = \{C, B, E\}, \alpha - T_4 = \{C, B\}$ and $\alpha - T_6 = \{C, B, E\}$.

The cost of dataset scans is greatly reduced using dataset projection techniques. The size of transactions gets smaller as the algorithm explores larger itemsets. Implementing dataset projection in the algorithm is a tough task and various inefficient approaches are out there to perform it. EFIM algorithm presents an efficient technique to perform dataset projection [27]. We adopt efficient dataset projection technique presented by EFIM algorithm.

### 4.3.2 Dataset scanning using transaction merging

The cost of performing dataset scans can be further reduced using an efficient transaction merging technique. After performing dataset projection, there exist a lot of identical transactions. Hence, transaction merging technique is performed after dataset projection technique.

**Definition 4.5** (**Transaction merging**) Transaction merging technique identifies identical transactions and replaces them with a single transaction. $Ta_1, Ta_2, \ldots, Ta_m$ in a dataset $D$ by a single new transaction $T_M = Ta_1 = Ta_2 = \ldots = Ta_m$ where the quantity of each item $x \in T_M$ is defined as $IU(x, T_M) = \sum_{j=1\ldots} IU(x, Ta_j)$.

Applying transaction merging technique on projected datasets achieves a much higher reduction in the size of the dataset.

**Definition 4.6** (**Projected transaction merging**) Let the identical transaction as $Ta_1$, $Ta_2$, $Ta_3$, $Ta_n$ in the dataset $\alpha - D$ is replaced by a new transaction $T_M = Ta_1 = Ta_2 = Ta_3 = Ta_n$ and quantity of these identical transactions $x \in T_M$ is defined as $IU(x, T_M) = \sum_{i=1,\ldots,n} IU(x, Ta_i)$.

For example, Let us consider dataset $D$ and the projected dataset $\alpha - D$ where $\alpha = \{C\}$ contains transactions $\alpha - T_2 = \{E, F\}$, $\alpha - T_3 = \{D, E\}$, $\alpha - T_4 = \{D, E\}$, $\alpha - T_6 = \{D, E\}$ and $\alpha - T_7 = \{E, F\}$. Hence, transactions $\alpha - T_3$, $\alpha - T_4$ and $\alpha - T_6$ can be merged and replaced by a new transaction $T_m = \{D, E\}$ where $IU(D, T_m) = 6$ and $IU(E, T_m) = 6$. Also transactions $\alpha - T_2$, and $\alpha - T_7$ can be replaced by a new transaction $T_{m1} = \{E, F\}$ where $IU(E, T_m) = 3$ and $IU(F, T_m) = 5$

Transaction merging technique is desirable to reduce the size of the dataset. The main problem to implement this technique is to identify the identical transactions. To achieve this, we need to compare all transactions with one another. Therefore, the technique to compare all the transactions to each is not an efficient technique. To efficiently implements this techniques, we follow the technique proposed in EFIM [27].

## 4.4 Threshold raising strategies

**Algorithm 1** RIU_strategy

**Input**: set of RIU values for all items, k: desired number of HUIs.
**Output**: Raised $min\_util$.
1 Sort $RIU$ values.
2 Set $min\_util$ to the $k^{th}$ largest $RIU$ value.
3 **return** $min\_util$;

### 4.4.1 RIU strategy

RIU (Real item utilities) strategy is proposed by REPT algorithm [16]. We adopt this strategy to raise the $min\_util$ threshold. In first dataset scan, RIU or utility value of all the items are calculated as $\sum_{T_j \in D} U(x, T_j)$ and denoted by RIU($x$). For example, item $A$ occurs in transactions $T_1$, $T_5$, and $T_6$. The utility of item $A$ in these transactions are $U(A, T_1) = 4$, $U(A, T_5) = 4$, and $U(A, T_6) = 4$. Hence the RIU(A) $U(A, T_1) + U(A, T_5) + U(A, T_6) = 4 + 4 + 4 = 12$. Similarly, the utility of all the items are calculated.

Let RIU = $\{RIU_1, RIU_2, \ldots, RIU_n\}$ be the list of utilities of items in $I$. We first sort the list of RIU values using an efficient sorting algorithm. Then the RIU strategy raises the $min\_util$ value to $k^{th}$ largest value in the sorted RIU list. This new value now is used as $min\_util$ threshold by the algorithm until the threshold is increased again using another threshold raising strategy. For example, if $k = 2$ then the dataset is scanned and the utility of items is calculated. The second largest value in the list RIU is 27. Therefore, the value of $min\_util$ is increased to 27. This new $min\_util$ value is then used by the algorithm until it is again increased by another raising strategy.

**Algorithm 2** CUD_strategy

**Input**: CUDM:matrix, k: desired number of HUIs.
**Output**: Raised $min\_util$.
1 Extract $k$ largest value from the CUDM matrix using any efficient data structure such as priority queue.
2 Set $min\_util$ to the extracted value iff it is greater than the previous $min\_util$.
3 **return** $min\_util$;

### 4.4.2 CUD strategy

We utilize CUD strategy to increase $min\_util$ threshold using the utilities of 2-itemsets stored in the EUCS structure. CUD strategy is adopted from kHMC algorithm [5]. CUD strategy utilizes the same structure as used by EUCSP pruning strategy. The EUCS structure contains a pair of items having a $TWU$ no less than $min\_util$ which may thus be a HUIs. Hence, values on EUCS can be considered for raising the threshold. The structure for storing the utilities of pairs of items is called the CUD utility Matrix (CUDM). CUD strategy is applied after the RIU strategy.

**Algorithm 3** COV_strategy

**Input**: EUCST: matrix, k: desired number of HUIs.
**Output**: Raised $min\_util$.
1 **foreach** *item* $x \in I$ **do**
2  $I(x).COV \leftarrow \emptyset$;
3  **foreach** *each item* $y \in I$ *and* $y \succ x$ **do**
4  **if** $EUCST(x, y) = TWU(x)$ **then**
5  $I(x).COV \leftarrow I(x).COV \cup y$;
6 Extract $k^{th}$ largest value from the COVL using priority queue.
7 Set $min\_util$ to the extracted value.
8 **return** $min\_util$;

### 4.4.3 COV strategy

We utilize COV (Coverage) strategy to raise the $min\_util$ threshold. COV strategy stores the utilities of pairs of items

in structure named coverage list (COVL). To construct the COVL, we need to store all the values of CUDM into COVL. Then, COV strategy inserts the combinations of $i$ with all subsets of its coverage Ç($i$) in the COVL where item $i \in I$. After all items are processed, the construction of COVL is completed. The Algorithm 3 shows the COVL construction and $min\_util$ threshold raising process.

## 4.5 Pruning strategies

In HUIs mining, a key challenge is to design effective pruning strategies. For this purpose, we adopt $sup$ pruning strategy proposed by EFIM algorithm [27]. This strategy is based on sub-tree utility which was also introduced in EFIM algorithm. We also utilize EUCS based pruning strategy name EUCP.

### 4.5.1 Prune search space using *EUCP*

We find the $TWU$ value for each itemset using $EUCS$. Hence, $EUCS$ is also utilized to prune the search space.

**Definition 4.7** (**Pruning using** $EUC$ ($EUCP$)) Let $X$ be an itemset, If $TWU(X) < min\_util$, then for any extension $Y$ of $X$, $U(XY) < min\_util$.

### 4.5.2 Prune search space using sub-tree utility

**Definition 4.8** (**Sub-tree Utility**) For an itemset $\alpha$, and an item $x \in E(\alpha)$ that can be extended $\alpha$ to follow the depth-first search to the sub-tree. The sub-tree utility ($su$) of the item $x$, if $\alpha$ is $su(\alpha, x) = \sum_{T \in (\alpha \cup \{x\})}[U(\alpha, T) + U(x, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{x\})} U(i, T)]$.

Moreover in the running example $\alpha = \{A\}$. We have that $su(\alpha, B) = (4 + 6 + 5) = 15$, $su(\alpha, C)$=$(4 + 4 + 9) = 17$.

**Property 4.2** (**Pruning using** $su$ ($sup$)) *For an itemset $\alpha$ and an item $x \in E(\alpha)$, the utility value of $su(\alpha, x) \geq U(\alpha \cup \{x\})$ and accordingly, $su(\alpha, x) \geq U(x)$ keeps the extension $x$ of $\alpha \cup \{x\}$.*

Assume an itemset $z = x \cup y$, then the relationship between the proposed upper-bounds are as $TWU(z) = EUCP(x, y) \geq REU(z) = su(x, y)$ holds [7, 27].

In rest of the paper, we refer to items having $su$ and $TWU$ as $Primary$ and $Secondary$ respectively.

$Primary$ and $Secondary$ items: Assume an itemset $X$. The $Primary$ items of $\alpha$ is a set defined as $Primary(\alpha) = \{x \mid x \in E(\alpha) \wedge su(\alpha, x) \geq min\_util\}$. The $Secondary$ items of $\alpha$ is a set defined as $Secondary(\alpha) = \{x \mid x \in E(\alpha) \wedge TWU(\alpha, x) \geq min\_util\}$. Since $TWU(\alpha, x) \geq su(\alpha, x)$, $Primary(\alpha) \subseteq Secondary(\alpha)$.

## 4.6 Calculate upper bounds using utility array

We utilize an efficient array-based utility counting technique, that is $UA$. This technique is used to calculate the utility for $sup$ in linear time and space.

**Definition 4.9** (**Utility Array**) For the set of items $I$ appear in a dataset $D$, the $UA$ is an array of length $|I|$ that have an entry denoted by $UA[x]$ for each item $x \in I$. Each entry is called $UA$ that is used to store a utility value.

*Calculating the $TWU$ of all items using $UA$:* $UA$ is initialized to 0. Then the $UA[x]$ for each item $x \in T_j$ is calculated $UA[x] = UA[x] + TU(x, T_j)$ for each transaction $T_j$ in the dataset $D$. After the dataset is scanned, the $UA[x]$ contains $TWU(x)$ where each item $x \in I$.

*Calculating the $su(\alpha)$:* $UA$ is initialized to 0. Then the $UA[x]$ for each item $x \in T_j \cap E(\alpha)$ is calculated $UA[x] = UA[x] + U(\alpha, T) + U(x, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{x\})} U(x, T)$ for each transaction $T_j$ in the dataset $D$. After the dataset is scanned, the $UA[x]$ contains $su(\alpha, x)$ $\forall x \in I$ where each item $x \in E(\alpha)$.

**Algorithm 4** Proposed TKEH algorithm

**Input**: $D$: a transaction dataset, $k$: desired number of HUIs.
**Output**: Top-k high utility itemsets.

1 $\alpha \leftarrow \emptyset$;
2 $min\_util \leftarrow 1$.
3 Create a priority queue $kPatterns$ of size $k$. Scan $D$, compute $TWU(\alpha)$ for all items using $UA[x]$.
4 Compute $RIU(\alpha)$ for all items $k \in I$ and store these RIU values them in a hashMap.
5 **RIU_strategy**(hashmap RIU, k).
6 Calculate $Secondary(\alpha) = \{x \mid x \in E(\alpha) \wedge TWU(\alpha, x) \geq min\_util\}$.
7 Let $\succ$ be the total order of $TWU$ increasing values on $Secondary(\alpha)$
8 Scan $D$, remove item $x \notin Secondary(\alpha)$ from the transactions $T_j$ and delete empty transactions $T_j$;
9 Sort all the remaining transactions in $D$ according to $\succ_T$;
10 Build CUDM structure and COVL structure.
11 **CUD_strategy**(hashmap CUDM, k).
12 **COV_strategy**(hashmap EUCST,k). Scan $D$, compute $su(\alpha, x)$ of each item $x \in Secondary(\alpha)$, using $UA[x]$;
13 $Primary(\alpha) = \{x \mid x \in Secondary(\alpha) \wedge su(\alpha, x) \geq min\_util\}$;
14 $search(\alpha, D, Primary(\alpha), Secondary(\alpha), min\_util$, kPatterns);
15 **return** Top-k HUIs;

**Algorithm 5** The *Search* procedure

**Input**: $\alpha$ : an itemset, $\alpha - D$ : a projected dataset, $Primary(\alpha)$ : the primary items of $\alpha$, $Secondary(\alpha)$: secondary items of $\alpha$, $min\_util$ and $kPatterns$: priority queue of k items

**Output**: The set of top-k HUIs that are extension of $\alpha$

1 **foreach** *item* $x \in Primary(\alpha)$ **do**
2 $\beta \leftarrow \alpha \cup \{x\}$
3 Scan $\alpha - D$, compute $U(\beta)$, and create $\beta - D$;
4 **if** $U(\beta) \geq min\_util$ **then**
5 add $\beta$ in $kPatterns$. And raise the $min\_util$ to the top of priority queue element's utility.
6 Scan $\beta - D$, compute $su(\beta, x)$ and $TWU(\beta)$ where item $x \in Secondary(\alpha)$, using two $UAs$
7 $Primary(\beta) = \{x \in Secondary(\alpha) \mid su(\beta, x) \geq min\_util\}$;
8 $Secondary(\beta) = \{x \in Secondary(\alpha) \mid TWU(\beta) \geq min\_util\}$;
9 $search(\beta, \beta - D, Primary(\beta), Secondary(\beta), min\_util, kPatterns)$;

### 4.7 Main procedure of TKEH

In this subsection, we demonstrate the proposed algorithm TKEH which mines the top-k HUIs. We utilize several novel ideas that are explained previously. TKEH includes several strategies to raise the threshold. We also utilize array-based technique to calculate the utility values of items and upper-bounds.

The main procedure of TKEH is shown in Algorithm 4. This procedure takes transactional dataset $D$ and user-defined parameter $k$ as an input. Algorithm 4 returns the top-k HUIs of the dataset $D$. Line 1 sets itemset $\alpha$ as empty. Line 2 initially initializes the $min\_util$ 1. Line 3 scans the dataset $D$, calculates $TWU$ for all the items and creates a priority queue (named $kPatterns$) of $k$ size. Line 4 calculates the $RIU$ values for all the items and store these values into hashMap as describe in Section 4.4.1. RIU_strategy (Algorithm 1) is executed in line 5 that raise the $min\_util$ threshold. Line 6 finds the $Secondary$ items for the itemset $\alpha$. Line 7 sorts the items of Secondary set in nondecreasing order of $TWU$ on $Secondary$ items. Line 8 removes the items that are not in $Secondary$ and also removes the empty transactions from the dataset $D$. Line 9 sorts all the transactions to $\succ_T$. Line 10 builds $CUDM$ and $COVL$ structure. Line 11 and line 12 call $CUD\_strategy$ (Algorithm 2) and $COV\_strategy$ (Algorithm 3) respectively. Line 13 calculates the sub-tree utility for each item of $Secondary$ set. Line 14 finds the $Primary$ items. Line 15 calls Algorithm 5 to extend the itemset $\alpha$ by performing the depth-first search.

Algorithm 5 takes as input the current itemset $\alpha$, projected dataset, Primary, Secondary items, internal $min\_util$ threshold and priority queue $kPatterns$. This procedure extends $\alpha$ with single items during each call. Line 1 finds the extension of $\alpha$ with items of $Primary$ set. Line 2 initializes $\beta$ as $\alpha \cup \{x\}$. Line 3 calculates the utility of itemset $\beta$ and create the projected dataset for $\beta - D$. Line 4 checks the itemset $\beta$ is HUIs or not, if $\beta$ fulfill the $min\_util$ threshold then add in priority queue $kPatterns$ (line 5). Line 6 calculates $su$ and $TWU$ for itemset $\beta$. Line 7 and line 8 find the $Primary$ and $Secondary$ set for itemset $\beta$ respectively for $\beta$. Lastly, line 9 calls Algorithm 5 recursively to extend $\beta$ using depth-first search.

**Table 9** Transaction utility

| $T_{ID}$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ |
|---|---|---|---|---|---|---|---|
| TU | 17 | 15 | 21 | 9 | 7 | 20 | 22 |

**Table 10** Transaction weighted utility

| Item | *A* | *B* | *C* | *D* | *E* | *F* |
|---|---|---|---|---|---|---|
| TWU | 44 | 95 | 87 | 67 | 104 | 44 |

## 5 An example

In this section, a simple example is given to show how the proposed algorithm can find top-k HUIs from transactional dataset. We assume there are seven transactions in the dataset as shown in Table 2. Each item has their purchase quantity or internal utility in the transaction. For example, $IU(A, T_1) = 2$ in Table 2. There are six items in the transactions, denoted as $A$ to $F$. We also assume external utility for the six items. For example, $EU(A) = 2$ as shown in Table 3. The utility of item is calculated by multiplication of internal utility and external utility as described in Definition 3.4. For example utility of an item $A$ is calculated as $U(A, T_1) = IU(A, T_1) \times EU(A) = 2 \times 2 = 4$. The utility of an itemset in the dataset is calculated as defined by Definition 3.6. For example, utility

**Table 11** Top-k HUIs for $k = 10$

| Itemsets | Utility | Itemsets | Utility |
|---|---|---|---|
| {D, E} | 37 | {F, E, C, B} | 37 |
| {D, B} | 39 | {D, C, B, E} | 37 |
| {D, B, E} | 45 | {D, C, B} | 34 |
| {D, C, E} | 37 | {B, E} | 36 |
| {C, B, E} | 39 | {F, C, B} | 34 |

**Table 12** Statistical information about datasets

| Dataset | # of transactions | # of items | Avg. length | Max. Length | Type |
|---|---|---|---|---|---|
| Accidents | 340183 | 468 | 33.8 | 51 | Dense |
| Chess | 3196 | 75 | 37 | 37 | Dense |
| Mushroom | 8124 | 119 | 23 | 23 | Dense |
| Foodmart | 4141 | 1559 | 4.42 | 14 | Sparse |
| Retail | 88162 | 16470 | 10.3 | 76 | Sparse |

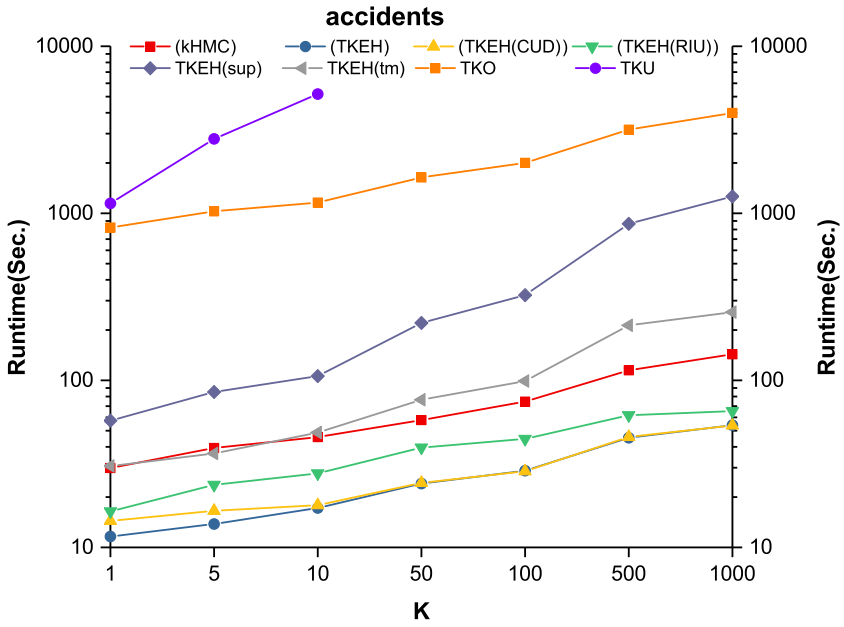


**Fig. 1** The runtime on accident

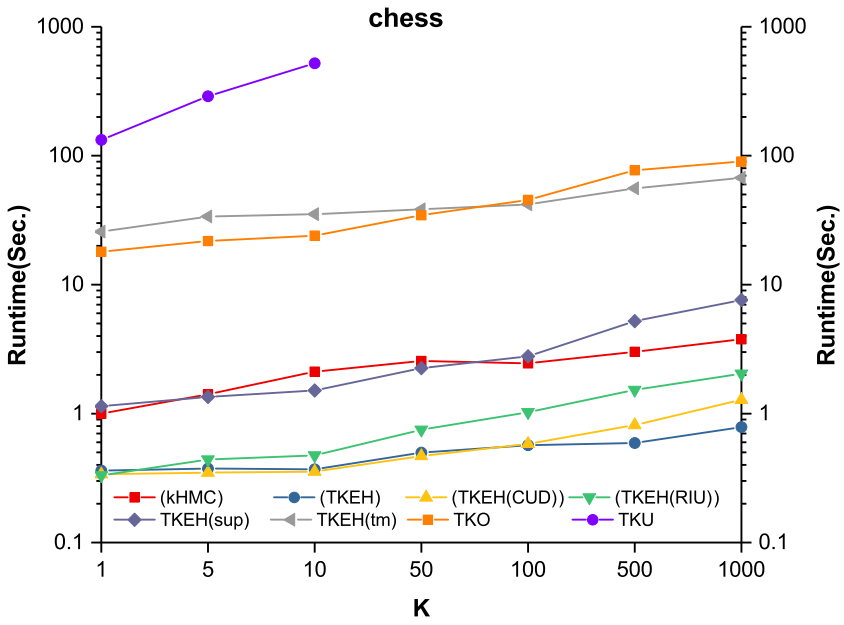


**Fig. 2** The runtime on chess

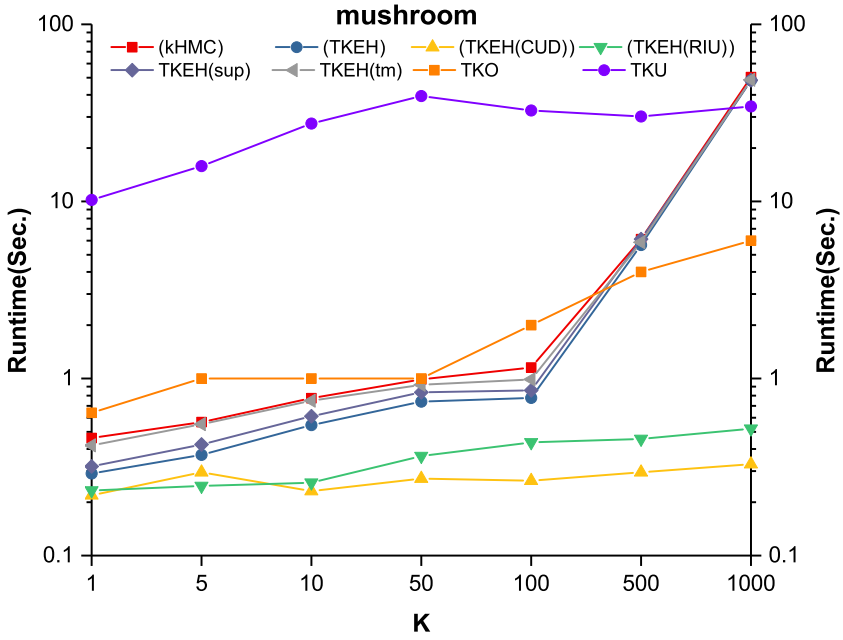


**Fig. 3** The runtime on mushroom

of an itemset $AE$ is calculated as $U(AE, T_1) = IU(A, T_1) \times EU(A) + IU(E, T_1) \times EU(E) = 7$.

The transaction utility of the running example is calculated as defined in Definition 3.7. $TU$ of $T_1$ is calculated as $TU(T_1) = U(A, T_1) + U(B, T_1) + U(D, T_1) + U(E, T_1) = 17$. The transaction utility values of all the transactions are shown in Table 9. The transaction weighted utility of each item is calculated by following Definition 3.8. Table 10 shows the $TWU$ of each item. The itemsets follow $min\_util$ threshold become the HUIs as describe by Definition 3.9. In the running example, we assume the $min\_util$ is 30. The HUIs for the running example is $\{ADBE : 33, DC : 31, DCB : 34, DCBE : 37, DCE : 37, DB : 39, DBE : 45, DE : 37, CB : 33, CBE : 39, BE : 36\}$ where the number beside each itemset indicates its utility value. It is not an easy task for the user to specify the appropriate $min\_util$ threshold. Therefore, top-k HUIs mining is introduced. In top-k HUIs

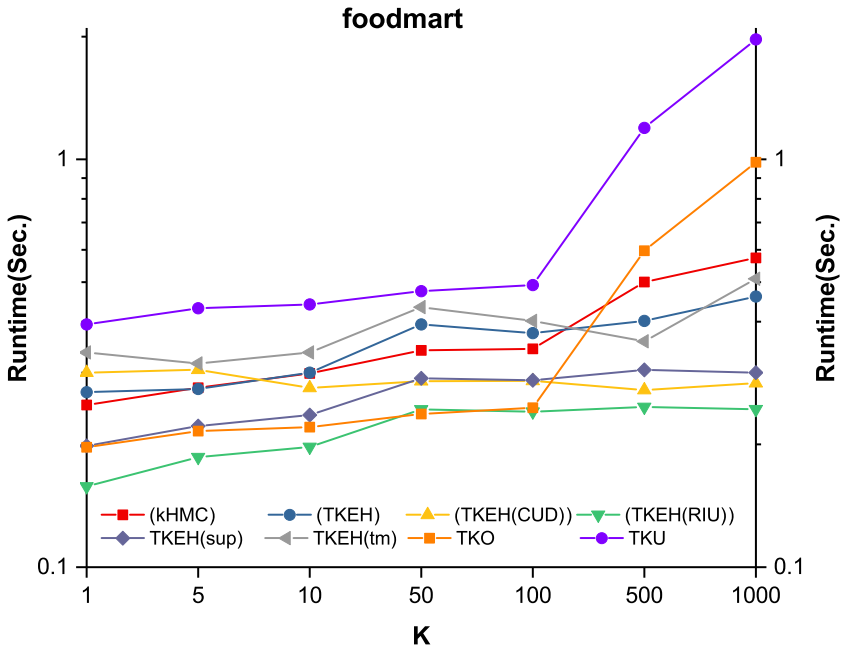


**Fig. 4** The runtime on foodmart

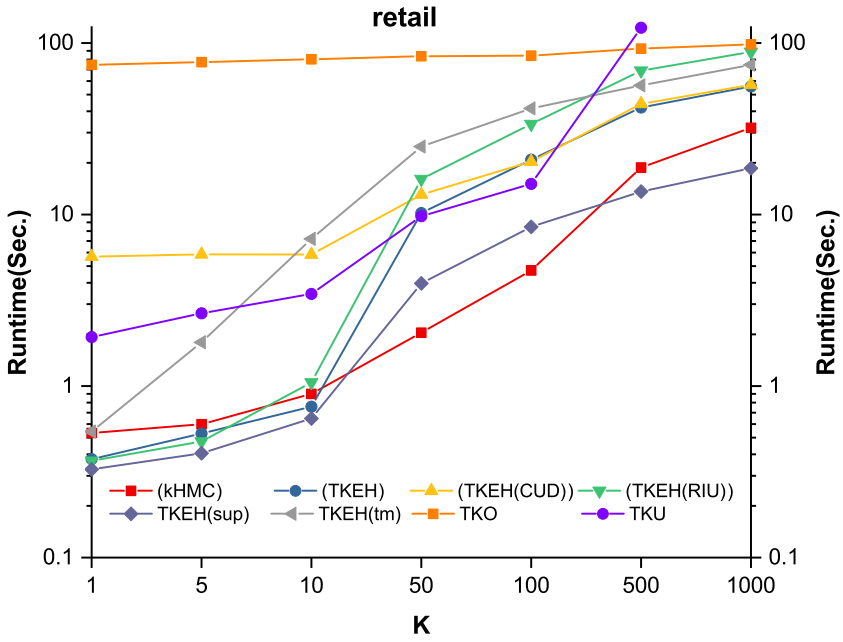


**Fig. 5** The runtime on retail

mining specifying the value of $k$ is easy. The final top-k HUIs for the running example is shown in Table 11 where $k = 10$.

## 6 Performance evaluations

All the experiments were conducted on an Intel Core-i7-6700 machine, 3.40 GHz CPU with 8 GB of memory, running Windows 10 Pro (64-bit OS). We compare the performance of TKEH with kHMC on the five real datasets available from spmf [6]. Moreover, to evaluate the influence of the design decisions in TKEH, we check the performance of four versions of TKEH named TKEH(CUD), TKEH(RIU), TKEH(sup) and TKEH(tm). TKEH utilizes all the threshold raising strategies (RIU, CUD and COV) and dataset reduction techniques (*tm* and *sup*). TKEH(CUD) utilizes only one threshold raising strategies named CUD and both dataset reduction techniques (*tm* and *sup*). Similarly, TKEH(RIU) utilizes only one threshold raising strategies named RIU and both dataset reduction techniques (*tm* and *sup*). TKEH(sup) and TKEH(tm) where *sup* and *tm*

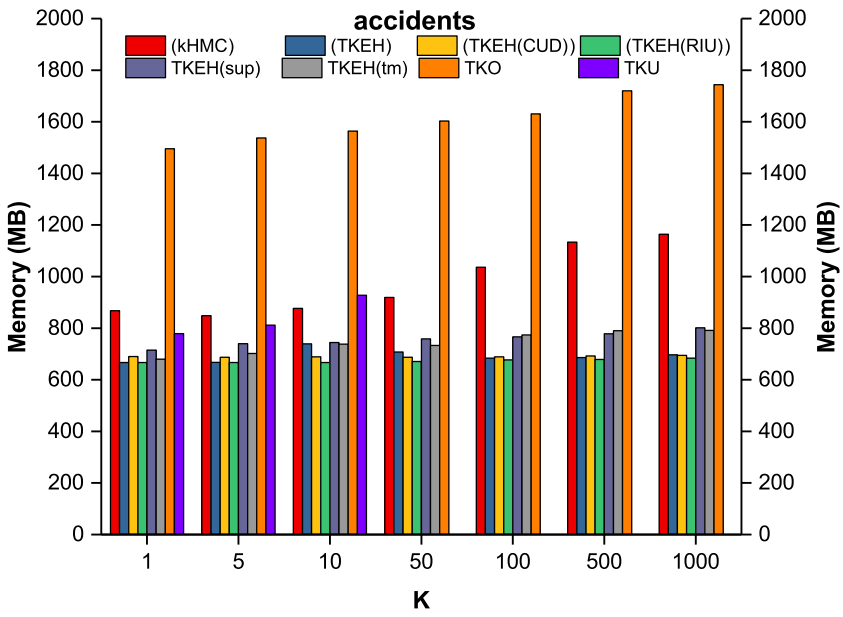


**Fig. 6** The memory usage on accident

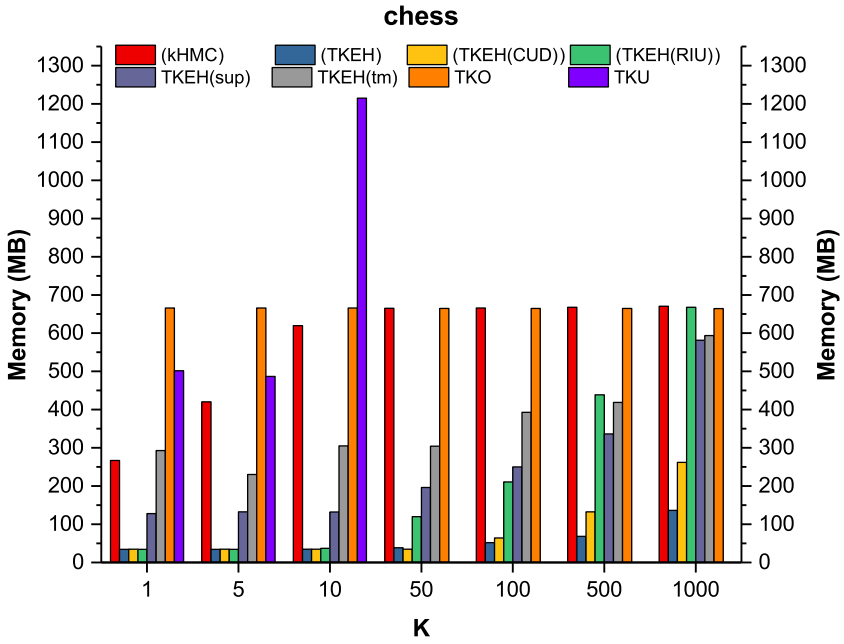


**Fig. 7** The memory usage on chess

are utilized respectively with all threshold raising strategies (CUD, RIU and COV).

The detailed characteristics of all the datasets are shown in Table 12 where Avg.length and Max.length denote the average transaction length and maximum transaction length respectively. The real size of accidents, chess, mushroom, foodmart and retail datasets are 63.1 MB, 641 KB, 1.03 MB, 175 KB and 6.42 MB respectively. The experimental results on dense and sparse datasets are separately shown and are discussed in Sections 6.1 and 6.2. The report of memory consumptions and scalability are shown respectively in Sections 6.3 and 6.4. To ensure robustness of the results, we ran all our experiments ten times to report the average results.

To compare the proposed algorithms with the state-of-the-art algorithms TKU, TKO and kHMC, we executed all the algorithms on all datasets by increasing $k$. Here the value of $k$ increases until all the algorithms take too much

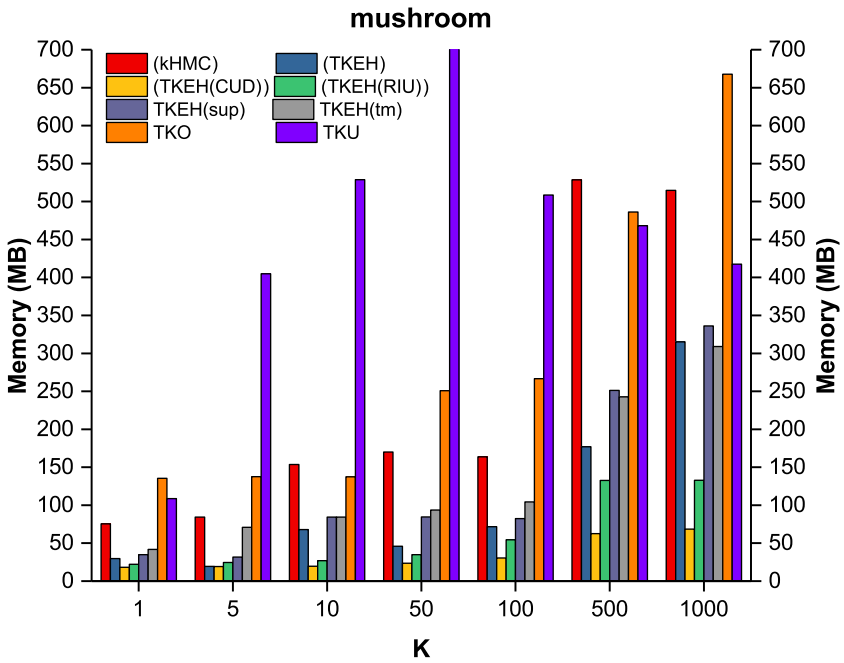


**Fig. 8** The memory usage on mushroom

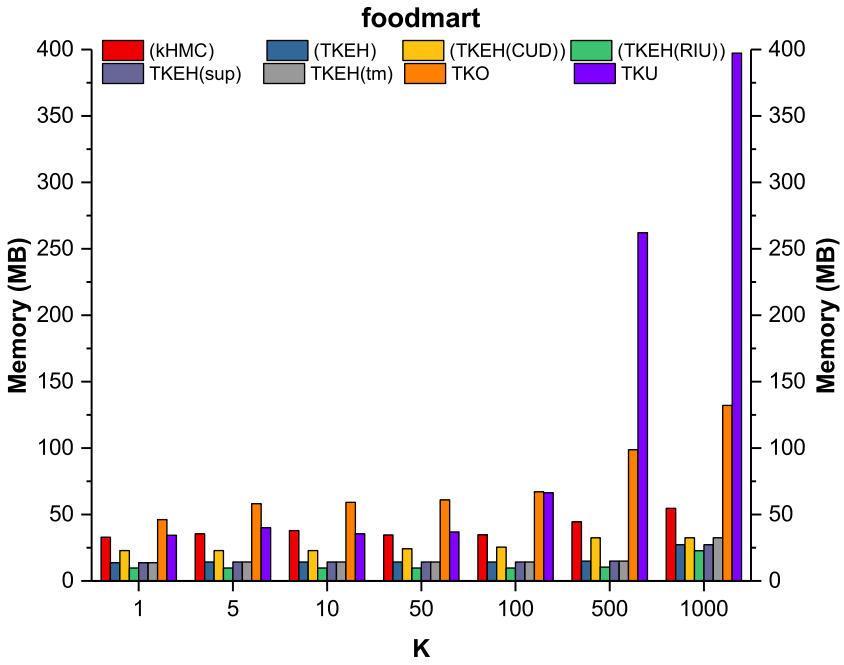


**Fig. 9** The memory usage on foodmart

time or out of memory or a clear winner is observed. The experimental results on all the datasets are separately shown in the next sections.

### 6.1 Dense datasets

For the dense datasets, all versions of TKEH performs consistently better than all the state-of-the algorithms. Figure 1 shows the running time of the all the algorithms on the accident dataset. We can see that both the *sup* and *tm* techniques are required to reduce the runtime and therefore, all the version of TKEH except TKEH(sup) and TKEH(tm) outperform kHMC algorithm for the very large $k$ values. For the chess and mushroom dataset, the *tm* technique merges the transactions widely. For the chess dataset, only the TKEH(sup) does not give better performance than kHMC algorithm. The other versions of TKEH give better performance as shown in the Fig. 2. For accidents and chess datasets, TKU executes up to $k = 10$. For 50 to 1000 value of $k$, TKO suffers from out of memory

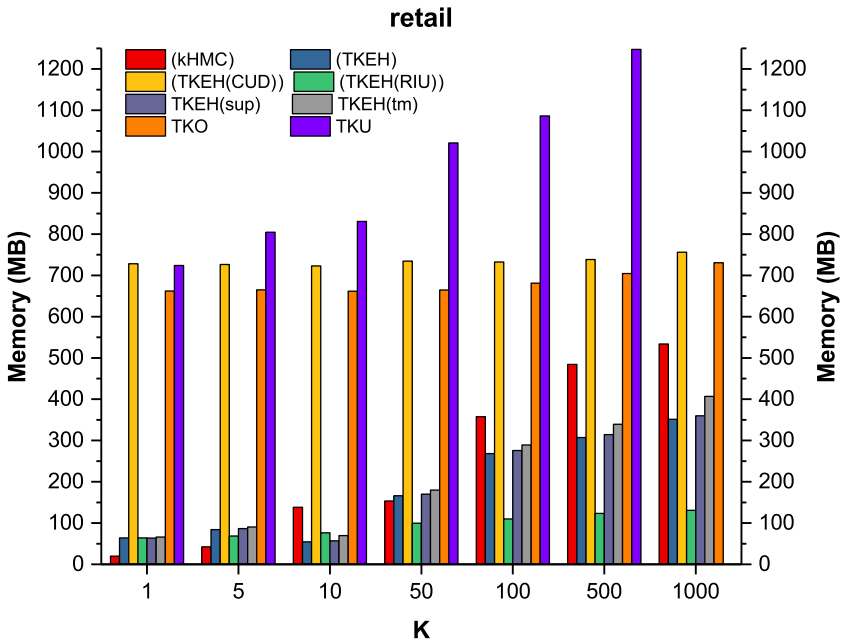


**Fig. 10** The memory usage on retail

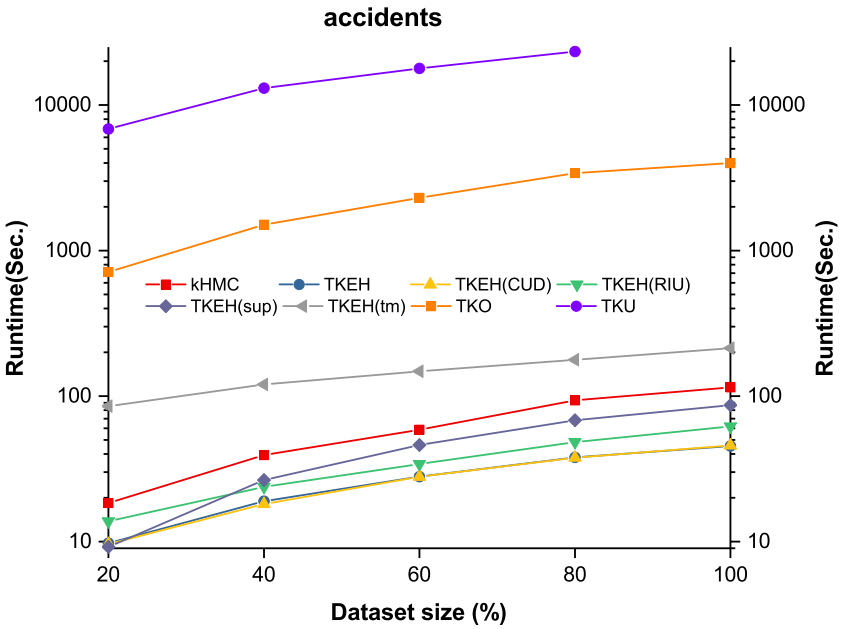


**Fig. 11** Runtime scalability of the algorithms on accidents

error. For the mushroom dataset, all the versions of TKEH outperform kHMC. TKEH, TKEH(CUD), TKEH(RIU) and TKEH(tm) give almost stable performance as shown in Fig. 3. TKEH(sup) and kHMC have their comparable runtime for mushroom dataset. TKEH, TKEH(CUD) and TKEH(RIU) outperform for dense datasets because *tm* technique merges the transactions widely. We observe that the state-of-the-art algorithm TKU and TKO perform worse than all other algorithms for all dense datasets.

## 6.2 Sparse datasets

For sparse datasets, the gaps between the runtime of the algorithms are smaller because most of the items do not

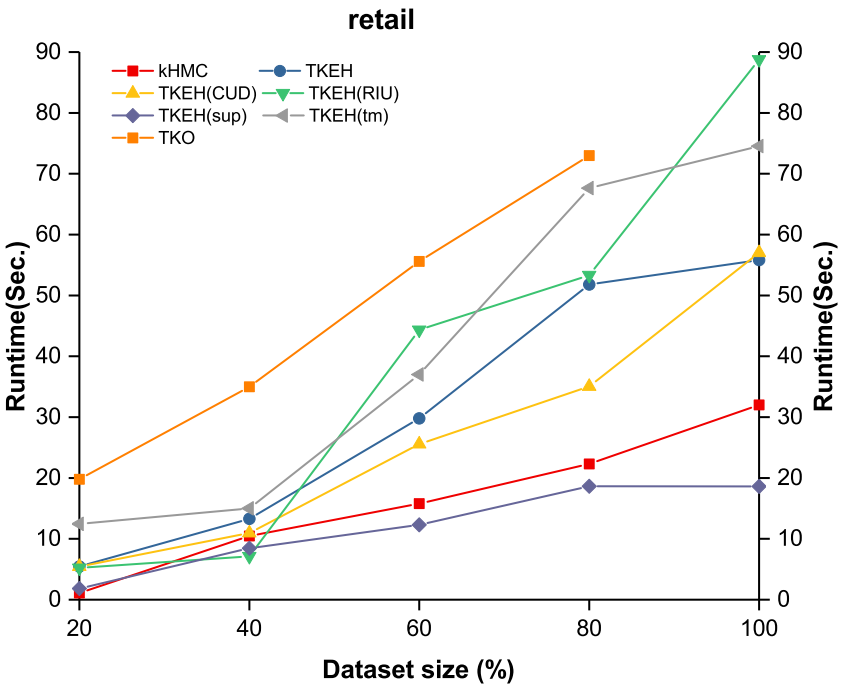


**Fig. 12** Runtime scalability of the algorithms on retail

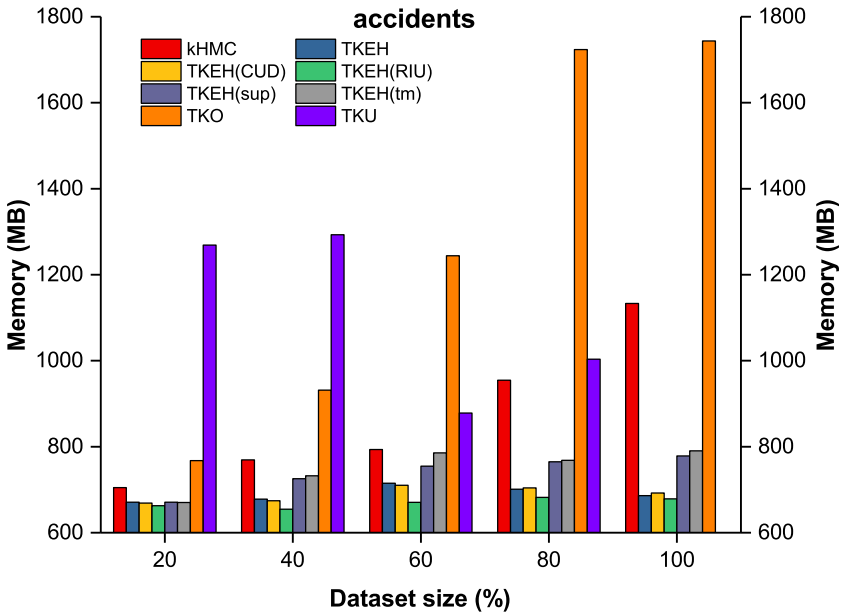


**Fig. 13** Memory usage scalability of the algorithms on accidents

;

appear in every transaction; thus *tm* technique is less effective and execution cost increases. For the foodmart dataset, kHMC performs better than TKEH and TKEH(tm) up to the $k = 100$. When the value of $k$ increases, kHMC gives the worst performance. Except for TKEH and TKEH(tm), all other algorithms continuously outperform kHMC as shown in Fig. 4. TKO executes successfully up to $k = 500$. On $k = 1000$, TKO suffers from out of memory error. We observe that TKO and TKU continuously perform worse on both the sparse datasets. In retail dataset, only TKEH(sup) outperforms kHMC algorithm as shown in the Fig. 5. The results show that retail dataset does not support *tm* technique. The retail dataset has a large number of distinct items and has wider maximum length than all the other datasets. Hence it does not support the *tm* technique.

We observe by the experimental results that when the datasets are highly sparse, we can drop the *tm* technique and then mine top-k HUIs efficiently.

### 6.3 Memory usage

In this section, we report the memory usage of the proposed algorithms and the state-of-the-art algorithms on three dense datasets (accidents, chess and mushroom) and two sparse

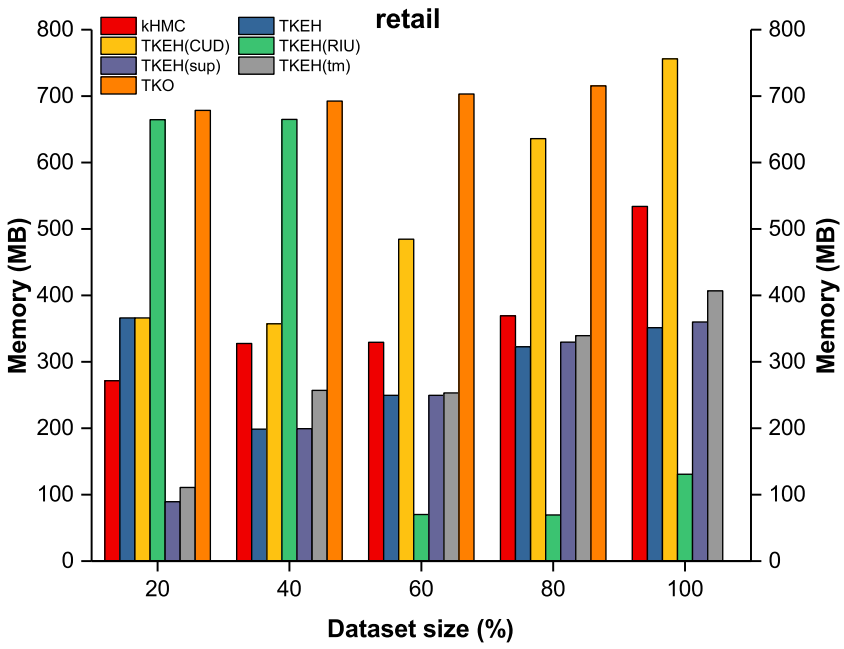


**Fig. 14** Memory usage scalability of the algorithms on retail

**Table 13** Relative runtime and memory usage improvements of TKEH over kHMC, TKO and TKU on accidents

| K | kHMC | TKO | TKU | kHMC | TKO | TKU |
|---|---|---|---|---|---|---|
| 1 | 2.57 | 70.83 | 98.66 | 1.30 | 2.24 | 1.17 |
| 5 | **2.85** | **74.43** | 202.04 | 1.27 | 2.30 | 1.22 |
| 10 | 2.66 | 67.44 | **300.69** | 1.19 | 2.12 | **1.25** |
| 50 | 2.40 | 67.98 | Crashed | 1.30 | 2.27 | Crashed |
| 100 | 2.59 | 69.68 | Crashed | 1.52 | 2.38 | Crashed |
| 500 | 2.54 | 69.88 | Crashed | 1.65 | **2.51** | Crashed |
| 1000 | 2.66 | 73.98 | Crashed | **1.67** | 2.50 | Crashed |

**Table 14** Relative runtime and memory usage improvements of TKEH over kHMC, TKO and TKU on chess

| K | kHMC | TKO | TKU | kHMC | TKO | TKU |
|---|---|---|---|---|---|---|
| 1 | 2.76 | 49.85 | 367.49 | 7.75 | **19.32** | 14.56 |
| 5 | 3.76 | 58.12 | 771.35 | 12.18 | 19.29 | 14.10 |
| 10 | **5.71** | 64.81 | **1409.28** | **17.91** | 19.25 | **35.11** |
| 50 | 5.14 | 69.39 | Crashed | 17.26 | 17.26 | Crashed |
| 100 | 4.31 | 79.82 | Crashed | 12.84 | 12.83 | Crashed |
| 500 | 5.09 | **130.36** | Crashed | 9.75 | 9.71 | Crashed |
| 1000 | 4.80 | 114.76 | Crashed | 4.91 | 4.87 | Crashed |

**Table 15** Relative runtime and memory usage improvements of TKEH over kHMC, TKO and TKU on mushroom

| K | kHMC | TKO | TKU | kHMC | TKO | TKU |
|---|---|---|---|---|---|---|
| 1 | **1.59** | 2.17 | 35.06 | 2.54 | 4.55 | 3.66 |
| 5 | 1.53 | **2.70** | 42.78 | **4.36** | **7.11** | **20.93** |
| 10 | 1.42 | 1.83 | 50.47 | 2.26 | 2.03 | 7.80 |
| 50 | 1.34 | 1.35 | **53.21** | 3.70 | 5.46 | 15.54 |
| 100 | 1.48 | 2.57 | 41.97 | 2.28 | 3.72 | 7.09 |
| 500 | 1.08 | 0.70 | 5.31 | 2.99 | 2.75 | 2.65 |
| 1000 | 1.05 | 0.12 | 0.71 | 1.63 | 2.12 | 1.32 |

**Table 16** Relative runtime and memory usage improvements of TKEH over kHMC, TKO and TKU on foodmart

| K | kHMC | TKO | TKU | kHMC | TKO | TKU |
|---|---|---|---|---|---|---|
| 1 | 0.93 | 0.74 | 1.47 | 2.41 | 3.38 | 2.52 |
| 5 | 1.01 | 0.79 | 1.58 | 2.48 | 4.07 | 2.80 |
| 10 | 1.00 | 0.75 | 1.47 | 2.65 | 4.14 | 2.48 |
| 50 | 0.86 | 0.70 | 1.21 | 2.42 | 4.26 | 2.58 |
| 100 | 0.92 | 0.92 | 1.31 | 2.42 | 4.69 | 4.64 |
| 500 | **1.25** | 1.54 | 2.97 | **2.97** | **6.61** | **17.53** |
| 1000 | 1.24 | **1.82** | **4.27** | 2.00 | 4.84 | 14.55 |

**Table 17** Relative runtime and memory usage improvements of TKEH over kHMC, TKO and TKU on retail

| K | kHMC | TKO | TKU | kHMC | TKO | TKU |
|---|---|---|---|---|---|---|
| 1 | **1.42** | **198.93** | **5.14** | 0.31 | 10.35 | 11.32 |
| 5 | 1.13 | 146.51 | 5.03 | 0.50 | 7.90 | 9.55 |
| 10 | 1.19 | 105.96 | 4.53 | **2.55** | **12.18** | **15.30** |
| 50 | 0.20 | 8.20 | 0.96 | 0.92 | 4.00 | 6.14 |
| 100 | 0.23 | 4.04 | 0.72 | 1.33 | 2.54 | 4.05 |
| 500 | 0.45 | 2.21 | 2.92 | 1.58 | 2.29 | 4.06 |
| 1000 | 0.57 | 1.76 | Crashed | 1.52 | 2.08 | Crashed |

datasets (foodmart and retail). For all the dense datasets, all the versions of TKEH consume less memory than kHMC as shown in the Figs. 6, 7, 8 and 9. In retail dataset, kHMC performs better than TKEH(CUD) and the other versions outperform kHMC as seen in the Fig. 10. kHMC consumes huge memory due to the construction of the utility-list during the mining process. TKO and TKU always consume more memory than all other algorithms, including the proposed algorithms. TKEH reuses some of it's data structures and reuses same $UAs$ to calculate utility of items and upper-bounds. Hence, all the versions of the proposed algorithm consume less memory than kHMC except for the retail dataset. The retail dataset is highly sparse and proposed algorithms are not efficient for highly sparse datasets. The transaction merging and dataset projection techniques utilized by TKEH are not suitable for highly sparse datasets such as retail.

### 6.4 Scalability

In order to test the scalability of all the algorithms, we varied the size of the dataset from 25% to 100% and studied the execution and memory consumption performance. We fix the value of $k$ by 1000 to check the scalability comparison of all the algorithm. In order to check the scalability, one dense (accidents) and one sparse (retail) dataset is used. All the versions of TKEH consume less runtime and less memory than the current best state-of-the-art algorithm kHMC. The Figs. 11 and 12 show that the running time of TKEH increases linearly with increased dataset size. We also observe that the memory usage of TKEH increases linearly when the number of transactions increases as shown in the Figs. 13 and 14. This indicates that TKEH scales well with the size of dataset.

## 7 Discussion

This paper utilizes three threshold raising strategies, RIU, CUD and COV. Transaction merging and utility array-based technique are utilized to reduce the memory requirement and speed up the execution process. The paper also uses $sup$ and EUCP strategy to prune the search space. The results show that TKEH has significant improvement in runtime for accident, chess, foodmart and mushroom datasets. The execution time on retail dataset is comparable. Moreover, the memory improvements of TKEH on all datasets is quite significant. The runtime performance improvements of TKEH are quite significant compared to kHMC, TKO and TKU on all the datasets except retail. TKU crashed for accidents, chess and retail dataset. In Tables 13, 14, 15, 16 and 17 the word **Crashed** indicates that TKU crashed due to running out of memory. Numbers in bold indicate that it to be the best performance of TKEH to that algorithm.

Table 13 provides a summary of relative runtime and memory usage of TKEH over kHMC, TKO and TKU on accidents. The column 2 in Table 13 provides that how many times TKEH is faster than kHMC. For example, for k=1000, TKEH is 2.66 times faster than kHMC. Similarly, columns 3 and 4 show that how many times TKEH is faster than TKO and TKU respectively. The columns 5 to 7 show that relative memory usage comparison. The column 5 shows how many times TKEH consumes less memory than kHMC. Similarly, columns 6 and 7 show the memory consumption improvement of TKEH over TKO and TKU respectively. Table 14, 15, 16 and 17 show the relative runtime and memory usage improvements of TKEH over kHMC, TKO and TKU on chess, mushroom, foodmart and retail dataset respectively. The relative improvement results show that the proposed algorithm significantly outperforms the state-of-the-art algorithms.

## 8 Conclusion and future directions

This paper proposed an efficient top-k high utility itemsets mining algorithm (named TKEH). TKEH utilized three strategies (RIU, CUD and COV) to raise the $min\_util$ threshold. TKEH utilized transaction merging and dataset projection techniques to reduce the dataset scanning cost. A utility-array also utilized to calculate utility of items and upper-bounds. To show the effects of all the techniques and

strategies, five versions of TKEH are proposed separately named as TKEH, TKEH(CUD), TKEH(RIU), TKEH(sup) and TKEH(tm). The experimental results showed that proposed algorithms outperform the state-of-the-art algorithms TKU, TKO and kHMC. The adopted techniques makes the proposed algorithm faster and more efficient. The scalability analysis showed that TKEH is highly scalable for dense and sparse datasets. The runtime improvement analysis showed that the proposed algorithm up-to three orders of magnitude faster than the state-of-the-art algorithms. Moreover, the proposed algorithms are always more memory efficient for dense datasets. Overall, this paper makes an useful contribution in field of top-k HUIs mining.

In future, the proposed idea can also be used in the field of on-shelf HUIs mining, sequential HUIs mining and data stream mining.

## Compliance with Ethical Standards

The article uses threshold raising and memory reduction techniques to mine the top-k high utility itemsets mining.

**Conflict of interests** The authors declare no conflicts of interest.

## References


1. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th international conference on very large data bases, VLDB '94. Morgan Kaufmann Publishers Inc, San Francisco, pp 487–499
2. Ahmed CF, Tanbeer SK, Jeong B-S, Choi H-J (2012) Interactive mining of high utility patterns over data streams. Expert Syst Appl 39(15):11979–11991
3. Chu C-J, Tseng VS, Liang T (2008) An efficient algorithm for mining temporal high utility itemsets from data streams. J Syst Softw 81(7):1105–1117
4. Dam T-L, Li K, Fournier-Viger P, Duong Q-H (2017) An efficient algorithm for mining top-k on-shelf high utility itemsets. Knowl Inf Syst 52(3):621–655
5. Duong Q-H, Liao B, Fournier-Viger P, Dam T-L (2016) An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies. Knowl-Based Syst 104:106–122
6. Fournier-Viger P, Gomariz A, Gueniche T, Soltani A, Wu C-W, Tseng VS (2014a) Spmf: a java open-source pattern mining library. J Mach Learn Res 15(1):3389–3393
7. Fournier-Viger P, Wu C-W, Zida S, Tseng VS (2014b) FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. Springer International Publishing, Cham, pp 83–92
8. Fournier-Viger P, Zida S (2015) Foshu: faster on-shelf high utility itemset mining – with or without negative unit profit. In: Proceedings of the 30th annual ACM symposium on applied computing, SAC '15. ACM, New York, pp 857–864
9. Krishnamoorthy S (2015) Pruning strategies for mining high utility itemsets. Expert Syst Appl 42(5):2371–2381
10. Krishnamoorthy S (2017) Hminer: efficiently mining high utility itemsets. Expert Syst Appl 90(Supplement C):168–183
11. Lee S, Park JS (2016) Top-k high utility itemset mining based on utility-list structures. In: 2016 International conference on big data and smart computing (BigComp), pp 101–108
12. Li HF, Huang HY, Chen YC, Liu YJ, Lee SY (2008) Fast and memory efficient mining of high utility itemsets in data streams. In: 2008 Eighth IEEE International conference on data mining, pp 881–886
13. Liu J, Wang K, Fung BCM (2012) Direct discovery of high utility itemsets without candidate generation. In: 2012 IEEE 12th International conference on data mining. Brussels, pp 984–989
14. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM international conference on information and knowledge management, CIKM '12. ACM, New York, pp 55–64
15. Liu Y, Liao W-k, Choudhary A (2005) A two-phase algorithm for fast discovery of high utility itemsets. In: Proceedings of the 9th Pacific-Asia conference on advances in knowledge discovery and data mining, PAKDD'05. Springer, Berlin, pp 689–695
16. Ryang H, Yun U (2015) Top-k high utility pattern mining with effective threshold raising strategies. Knowl-Based Syst 76:109–126
17. Shie B-E, Hsiao H-F, Tseng VS, Yu PS (2011) Mining high utility mobile sequential patterns in mobile commerce environments. Springer, Berlin, pp 224–238
18. Shie B-E, Yu PS, Tseng VS (2013) Mining interesting user behavior patterns in mobile commerce environments. Appl Intell 38(3):418–435
19. Tseng VS, Shie B-E, Wu C-W, Yu PS (2013) Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans on Knowl and Data Eng 25(8):1772–1786
20. Tseng VS, Wu CW, Fournier-Viger P, Yu PS (2016) Efficient algorithms for mining top-k high utility itemsets. IEEE Trans Knowl Data Eng 28(1):54–67
21. Tseng VS, Wu C-W, Shie B-E, Yu PS (2010) Up-growth: an efficient algorithm for high utility itemset mining. In: Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '10. ACM, New York, pp 253–262
22. Wu CW, Shie B-E, Tseng VS, Yu PS (2012) Mining top-k high utility itemsets. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '12. ACM, New York, pp 78–86
23. Yao H, Hamilton HJ (2006) Mining itemset utilities from transaction databases. Data Knowl Eng 59(3):603–626
24. Yen S-J, Lee Y-S (2007) Mining high utility quantitative association rules. Springer, Berlin, pp 283–292
25. Yin J, Zheng Z, Cao L, Song Y, Wei W (2013) Efficiently mining top-k high utility sequential patterns. In: 2013 IEEE 13th International conference on data mining, pp 1259–1264
26. Yun U, Ryang H, Ryu KH (2014) High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates. Expert Syst Appl 41(8):3861–3878
27. Zida S, Fournier-Viger P, Lin JC-W, Wu C-W, Tseng VS (2017) Efim: a fast and memory efficient algorithm for high-utility itemset mining. Knowl Inf Syst 51(2):595–625
28. Zihayat M, An A (2014) Mining top-k high utility patterns over data streams. Inf Sci 285:138–161. Processing and Mining Complex Data Streams

**Kuldeep Singh** is currently pursuing his Ph.D in Computer science & Engineering from Indian Institute of Technology (BHU) Varanasi. His research interest includes High utility itemset mining, itemset mining, social network analysis, and data mining. He received his M.Tech degree in Computer science and Engineering from Guru Jambheshwar University of Science and Technology, Hisar (Haryana). He has 8 years of teaching and research experience.

**Shashank Sheshar Singh** received M.Tech. degree in Computer Science and Engineering from Indian Institute of Technology, Roorkee (IITR). He received B.Tech. degree in Computer Science and Engineering from Kali Charan Nigam Institute of Technology (KCNIT), Banda affiliated to GBTU University. He is working toward the Ph.D. in Computer Science and Engineering from Indian Institute of Technology (BHU), Varanasi. His research interests include Data Mining, Influence Maximization, Link Prediction and Social Network Analysis.

**Ajay Kumar** received M.Tech in Computer Science & Engineering from Samrat Ashok Technological Institute Vidisha (M.P.). He received B.Tech. in Computer Science and Engineering from R.K.D.F Institute of Science & Technology Bhopal (M.P.), affiliated to R.G.P.V. Bhopal (M.P.). He is working toward the Ph.D. in Computer Science and Engineering from Indian Institute of Technology (BHU), Varanasi. His research interests include Data Mining, Link Prediction, Influence Maximization, and Social Network Analysis.

**Dr. Bhaskar Biswas** received Ph.D. in Computer Science and Engineering from Indian Institute of Technology (BHU), Varanasi. He received the B.Tech and M. Tech degree from Birla Institute of Technology, Mesra. He is working as Associate Professor at Indian Institute of Technology (BHU), Varanasi in the Computer Science and Engineering department. He has around 17 years of teaching and research experience. His research interest includes Data Mining, High utility itemset mining, Machine Learning, Social Network Analysis and Evolutionary Computation.