Chapter 4

Top-k High Utility Itemsets Mining

In Chapter 3, we developed EHIL, a length-based HUIs algorithm. Although EHIL is more efficient than FHM+ in many cases, it can discover all the itemsets satisfying a given minimum utility and length constraints, it is often difficult for users to set a proper minimum utility, minimum length and maximum length thresholds. 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 HUIs mining has been presented where k is the number of HUIs to be found. In this chapter, we present an efficient algorithm (named TKEH) for finding top-k HUIs.

4.1 Preliminaries and Problem Definition

Definition 4.1.1. (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.

Let us consider the sample transactional dataset D which is given in TABLE 4.1. This dataset contains seven transactions $(T_1, T_2, \ldots, T_7)$ where each transaction contains items

TABLE 4.1: A transaction 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)

for example transaction T_1 indicates that items A, B, D and E appear with quantity respectively $2, 2, 1$ and 3. TABLE 4.2 indicates the external utility of each item.

TABLE 4.2: External utility value

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

Definition 4.1.2. (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. for example, $IU(B,T_1)$ = 2.

Definition 4.1.3. (External Utility ($EU(x)$). Let x be an item. The external utility of item x is defined as $EU(x)$. For example, $EU(B) = 3$.

TABLE 4.2 shows the external utility value of all the items.

Definition 4.1.4. (Utility of an item in a transaction). Utility of item x is defined as $U(x,T_j)$ = $IU(x,T_j) \times EU(x)$. For example,$U(A,T_1) = IU(A,T_1) \times EU(A) = 2 \times 2 = 4$.

Definition 4.1.5. (Utility of an itemset in a transaction). Let X be an itemset. Utility of an Itemset X is defined as $U(X,T_j)$= $\sum_{x \in X \wedge X \subseteq T_j} U(x,T_j)$. For example, $U(AE,T_1)$ $= IU(A,T_1) \times EU(A) + IU(E,T_1) \times EU(E) = 7$.

Definition 4.1.6. (Utility of an itemset in a database 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.

Definition 4.1.7. (Transaction utility). Transaction utility T_j is defined as $TU(T_j) =$ $\sum_{x \in T_j} U(x,T_j)$. For example, $U(T_1) = U(A,T_1) + U(B,T_1) + U(D,T_1) + U(E,T_1) = 17$. TU value of each transaction is shown in TABLE 4.3

Definition 4.1.8. (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 $TWU(X) =$

TABLE 4.3: 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 4.4: Transaction Weighted Utility

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

TABLE 4.5: 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

$\sum_{X \in T_j \wedge T_j \in D} TU(T_j)$. For example, $TWU(AE) = TU(T_1) + TU(T_6) = 37$. TWU value of each item is shown in TABLE 4.4

Property 4.1.1. (TWU based overestimation). If TWU value of itemset X is greater than utility value of itemset X, that is $TWU(X) > U(X)$, then the itemset is assumed as overestimated.

Property 4.1.2. (TWU based pruning). If 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 4.1.9. (High utility itemset). An itemset X is called high utility itemset if the $U(X) \geq min\_util$ threshold. Otherwise, itemset X is low high utility itemsets. The HUIs for the running example is $\{ACDE : 33, CD : 35, CDE : 35, BC : 33, BCE : 37, C : 36, CE : 39\}$ where the number beside each itemset indicates its utility value and min_util is 30.

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 [24, 27, 25, 26, 28].

Definition 4.1.10. (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 $RU(X,T_j) = \sum_{i\in(X,T_j)} U(i,T)$ [24, 27, 25].

Definition 4.1.11. (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 4.1.3. (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 [24].

Definition 4.1.12. (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. In the running example, TABLE 4.5 shows HUIs with the k value is 10.

4.2 TKEH Algorithm

In this section, we give a step-by-step analysis of the proposed algorithm named TKEH. Section 4.2.1 describes the search space and shows the techniques to find larger itemsets from items. Section 4.2.2 describes the EUCS structure that raise min_util threshold. Section 4.2.3 describes the dataset cost reduction techniques to reduce the dataset scanning. Section 4.2.4 describes the threshold raising techniques. Section 4.2.5 describes the pruning strategies. Section 4.2.6 introduces array-based utility counting technique. Finally, Section 4.2.7 gives the pseudo-code of the proposed algorithm.

4.2.1 The Search Space

The search space of the top-k HUIs mining problem can be represented as a set-enumeration tree as in [24]. The items can be explored using depth-first-search in the set-enumeration tree starting from the root which is the empty set. We sort the items in increasing order of TWU values that reduce the search space [24, 27]. Some definitions related to exploration of itemsets in the set-enumeration tree are given below.

TABLE 4.6: TWU values of items as $\succ$ order.

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

Definition 4.2.1. (Extension of an item). Let α be an itemset. The set of items that are used to extend the itemset α is denoted by $E(\alpha)$ and is defined as $E(\alpha) = \{z\,|z \in I \wedge z \succ x, \forall x \in \alpha\}$.

Definition 4.2.2. (Extension of an itemset). For the itemset α, Z is an extension of α that appears in a sub-tree of α 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 α that is a child of α 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 $\{D,E\}$. And single-item extensions of α are $\{C,D\}$, $\{C,E\}$ and $\{D,E\}$. The itemsets extensions of α is $\{C,D,E\}$.

4.2.2 Concept of Co-occurrence Structure

In top-k-HUIs mining, a key challenge is to design efficient techniques to raise 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 min_util and prune the search space. This strategy is proposed by FHM algorithm [27] and after that is improved by kHMC algorithm [51]. The following paragraph first describes the $EUCS$ structure and then $EUCST$ strategy.

TABLE 4.7: $EUCS$ map for transaction T_1.

Item	A	F	D	C	B
F		▨	▨	▨	▨
D	17		▨	▨	▨
C				▨	▨
B	17		17		▨
E	17		17		17

TABLE 4.8: $EUCS$ 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

Definition 4.2.3. (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 $TWU(\{x, y\}) = z$.

TABLE 4.9: *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

TABLE 4.10: 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

Property 4.2.1. 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 [27]. We implemented the *EUCS* using hashmap instead of triangular matrix. The hashmap based implementation is more efficient. The *EUCS* only stores the values that $TWU \neq 0$. Hence, 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 [24]. This sorting order can be denoted by $\succ$ order. TABLE 4.6 shows the *TWU* value for the running example as $\succ$ order. The second scan constructs *EUCS*. Each cell in table represents *TWU* values for the itemsets. TABLE 4.10 represents *EUCS* implementation of the running example. The creation of *EUCS* structure for transaction T_1 is shown in TABLE 4.7 where T_1 is $\{(A, 4), (D, 4), (B, 6), (E, 3)\}$. 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 4.7. 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 *TWU* is already for the item then we simply add *TWU* values and update the *EUCS*. As T_2 is $\{(F, 6), (C, 5), (B, 3), (E, 1)\}$. 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 4.8. The same process is repeated for the next transactions. TABLE 4.9 shows all the item without pruning. The itemsets having less *TWU* value than *min_util* (colored in brown in TABLE 4.9) are removed from EUCS. TABLE 4.10 shows the final *EUCS* for the running example after eliminate the items using *Property 4.2.1* where *min_util* is 30. EUCS structure based pruning is called EUCP that is proposed in [27].

4.2.3 Dataset Scanning Techniques

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

4.2.3.1 Dataset Scanning using Projection

The proposed algorithm calculates the utility and upper-bounds of itemsets by scanning the dataset. At the same time, TKEH creates EUCST and CUDM structure and also calculates RIU. 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 α, 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 of these items. A dataset without these items is known as projected dataset.

Definition 4.2.4. (Projected dataset). The projection of a transaction T_j using an itemset α 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 α is denoted by $\alpha - D$ and defined as the multi-set $\alpha - D = \{\alpha - T \mid T \in D \wedge \alpha - T \neq \emptyset\}$.

The cost of dataset scans is greatly reduced using dataset projection techniques. The size of transactions gets smaller and smaller as the algorithm explores larger itemsets. Implementing dataset projection in the algorithm is quite a tough task and various inefficient approaches are present in literature to perform it. EFIM algorithm presents an efficient technique to perform dataset projection [28]. We utilize efficient dataset projection technique presented by EFIM algorithm.

4.2.3.2 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 4.2.5. (Transaction merging). Transaction merging technique identifies identical transactions and replaces them with a single transaction. Transactions $Ta_1, Ta_2, \ldots, Ta_m$ in a dataset D are merged 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 $IU(x, T_M)$ $=\sum_{j=1 \ldots m} IU(x, Ta_j)$.

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

Definition 4.2.6. (Projected transaction merging). Let the identical transaction as Ta_1, Ta_2, Ta_3, Ta_n in the a 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. But this 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 [28].

4.2.4 Threshold Raising Strategies

Algorithm 3: 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 $min\_util$;
```

4.2.4.1 RIU strategy

RIU (Real item utilities) strategy is proposed by REPT algorithm [49]. We utilize this strategy to raise the internal *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, RIU(A) is $U(A, T_1) + U(A, T_5) + U(A, T_6) = 4 + 4 + 4 = 12$. 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 *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, 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 4: 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 $min\_util$;
```

4.2.4.2 CUD strategy

We utilize CUD (Co-occurrence with Utility Descending order) strategy to increase the internal *min_util* threshold using the utilities of 2-itemsets stored in the EUCS structure. CUD strategy is utilized from kHMC algorithm [51]. 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 5: 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 $min\_util$;
```

4.2.4.3 COV strategy

We utilize COV (Coverage) strategy to raise the internal *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 item i with all subsets of its coverage $C(i)$ in the COVL where item $i \in I$. After all items are processed, the construction of COVL is completed. The **Algorithm 5** shows the COVL construction and *min_util* threshold raising process.

4.2.5 Pruning Strategies

In HUIs mining, a key problem is to design effective pruning strategies. For this purpose, we utilize *sup* pruning strategy proposed by EFIM algorithm [28]. This strategy is based on sub-tree utility which was also introduced in EFIM algorithm. We also utilize EUCS based pruning strategy name EUCP.

4.2.5.1 Prune search space using $EUCP$

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

Definition 4.2.7. (Pruning using *EUC* (*EUCP*)). Let X be an itemset, If $TWU(X) < min\_util$, then for any extension Y of $X, U(XY) < min\_util$.

4.2.5.2 Prune search space using Sub-tree Utility

Definition 4.2.8. (Sub-tree Utility). For an itemset α and an item $x \in E(\alpha)$ that can be extended α to follow the depth-first search to the sub-tree. The *su* of the item *x*, if α 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 4.2.2. (Pruning using *su* (*sup*)). For an itemset α 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 [27, 28].

In rest of the chapter, 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 α is a set defined as $Primary(\alpha) = \{x \mid x \in E(\alpha) \wedge su(\alpha, x) \geq min\_util\}$. The *Secondary* items of α is a set defined as $Secondary(\alpha) = \{x \mid x \in E(\alpha) \wedge TWU(\alpha, x) \geq min\_util\}$. Since $TWU(\alpha, x) \geq su(\alpha, x)$, $Primary(\alpha) \subseteq Secondary(\alpha)$.

4.2.6 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 4.2.9. (Utility Array). For the set of items I appear in a dataset D, 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 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 su(α): 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 6: 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).                              ▷ See Algorithm 3
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).                             ▷ See Algorithm 4
12 COV_strategy(hashmap EUCST,k).                             ▷ See Algorithm 5
13 Scan $D$, compute $su(\alpha,x)$ of each item $x \in Secondary(\alpha)$, using $UA[x]$;
14 $Primary(\alpha) = \{x \mid x \in Secondary(\alpha) \wedge su(\alpha,x) \geq min\_util\}$;
15 search($\alpha$, $D$, $Primary(\alpha)$, $Secondary(\alpha)$, $min\_util$, kpatterns);
16 return Top-k HUIs;              ▷ all itemset stored in priority queue, kPatterns
```

Algorithm 7: The *Search* procedure

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

Output: The set of top-k HUIs that are extension of α

```
1 foreach item $x \in Primary(\alpha)$ do
2     $\beta \leftarrow \alpha \cup \{x\}$
3     Scan $\alpha - D$, compute $U(\beta)$ and create $\beta - D$;          ▷ using transaction merging
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);
```

4.2.7 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 6**. This procedure takes transactional dataset D and user-defined parameter k as an input. **Algorithm 6** returns the top-k HUIs of the dataset D. Line 1 sets itemset α as empty. Line 2 initially initializes *min_util* by 1. Line 3 scans the dataset D, calculate 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 4.2.4.1. **RIU_strategy** (**Algorithm 3**) is executed in line 5 that raise *min_util* threshold. Line 6 finds the *Secondary* items for the itemset α. 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 remove 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 4**) and *COV_strategy* (**Algorithm 5**) respectively to raise the internal *min_util* threshold. Line 13 calculates sub-tree utility for each item of

Secondary set. Line 14 finds the *Primary* items. Line 15 calls **Algorithm 7** to extend the itemset α by performing the depth-first search.

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

TABLE 4.11: 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

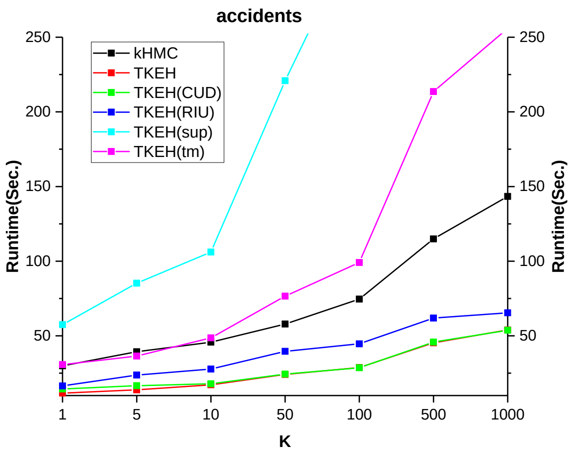

FIGURE 4.1: Runtime evaluation on accident dataset

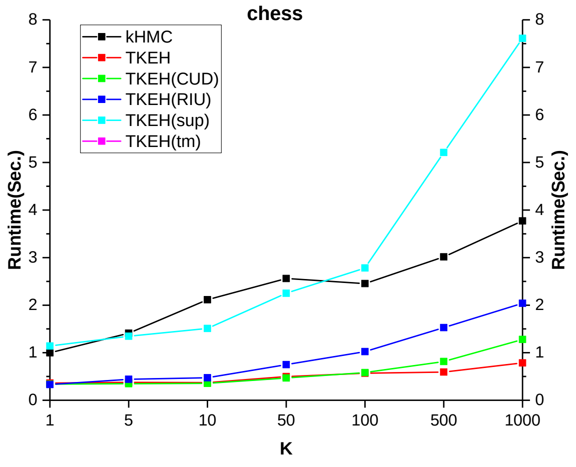

FIGURE 4.2: Runtime evaluation on chess dataset

4.3 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 the state-of-the-art algorithm kHMC on the five real datasets available from spmf [77]. Moreover, to evaluate the influence of the design strategies 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) utilize *sup* and *tm* technique respectively with all threshold raising strategies (CUD, RIU and COV).

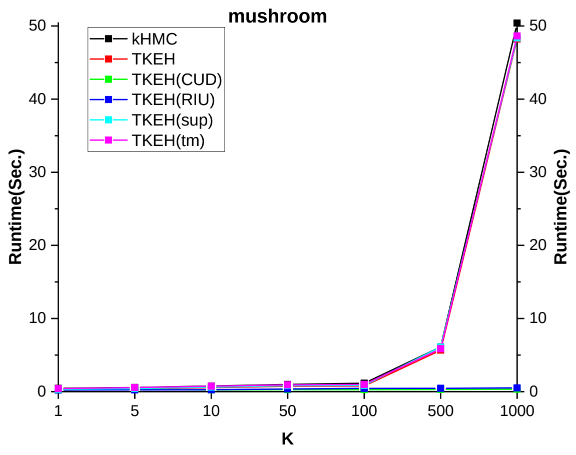

FIGURE 4.3: Runtime evaluation on mushroom dataset

The detailed characteristics of all the datasets are shown in TABLE 4.11 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 all these datasets are separately shown and are discussed in Section 4.3.1 and Section 4.3.2. The report of memory consumptions and scalability are shown respectively in Section 4.3.3 and Section 4.3.4. 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 algorithm, we executed both of the algorithms on all datasets by increasing k. Here the value of k increases until all the algorithms take too much 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.

4.3.1 Dense Datasets

For the dense datasets, all versions of TKEH performs consistently better than kHMC. FIGURE 4.1 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. Hence, all the version of TKEH except TKEH(sup) and TKEH(tm) outperform kHMC algorithm for the very large values of k. For the chess and mushroom dataset, the tm technique merges the transactions widely. For the chess dataset, only TKEH(sup) does not give good performance than kHMC algorithm, rest all the other versions of TKEH give better performance as shown in FIGURE 4.2. 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 FIGURE 4.3. 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.

4.3.2 Sparse Datasets

For sparse datasets, the gap between the runtime of the algorithms is smaller because the items do not appear in every transaction; thus tm technique is less effective and execution cost increases. For the foodmart dataset, kHMC performs well than TKEH and TKEH(tm) up to the $k = 100$. When the value of k increases, kHMC gives worst performance. Except TKEH and TKEH(tm), all other algorithms continuously outperform kHMC as shown in FIGURE 4.4. In retail dataset, only TKEH(sup) outperforms kHMC algorithm as shown

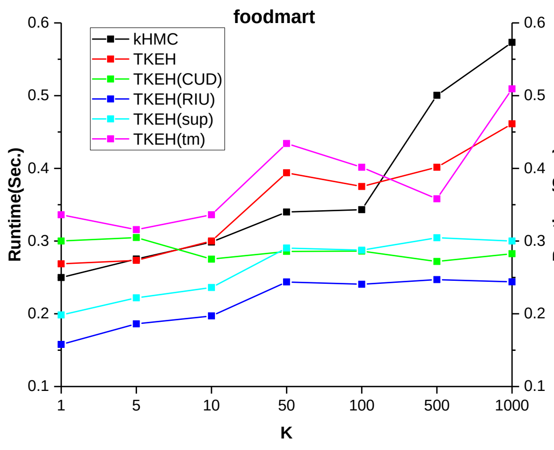

FIGURE 4.4: Runtime evaluation on foodmart dataset

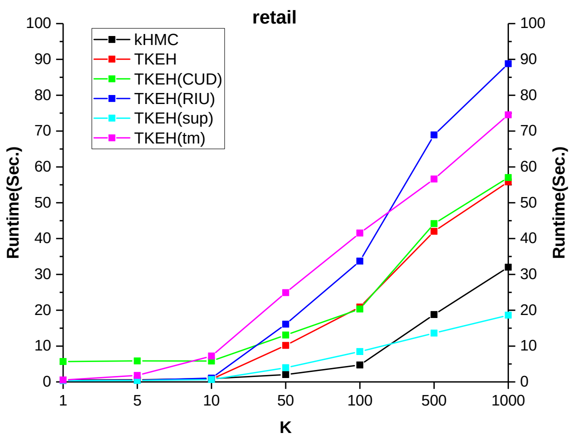

FIGURE 4.5: Runtime evaluation on retail dataset

in FIGURE 4.5. 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 transaction length than all the other datasets. Hence, it does not support *tm* technique.

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

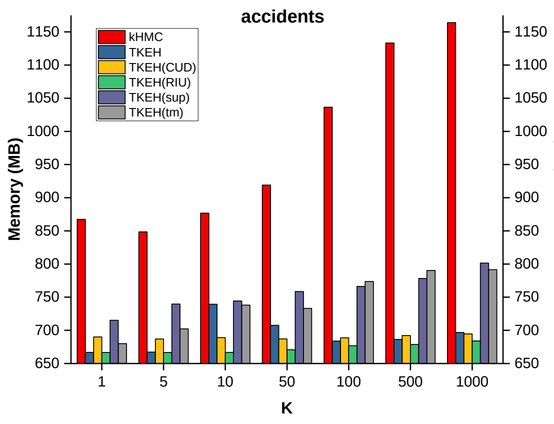

FIGURE 4.6: Memory consumption on accident dataset

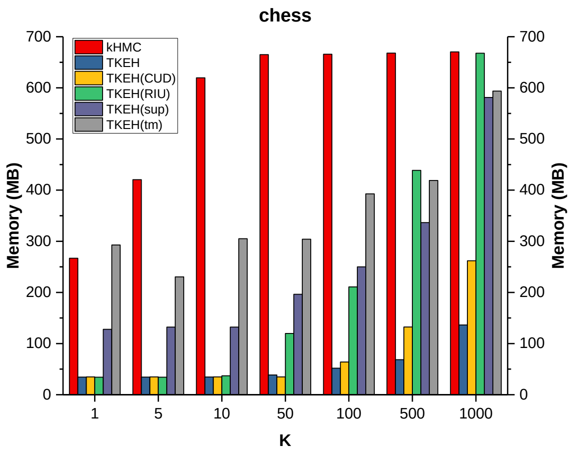

FIGURE 4.7: Memory consumption on chess dataset

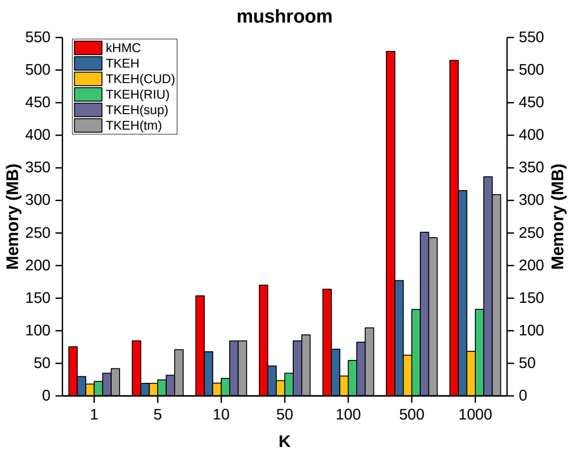

FIGURE 4.8: Memory consumption on mushroom dataset

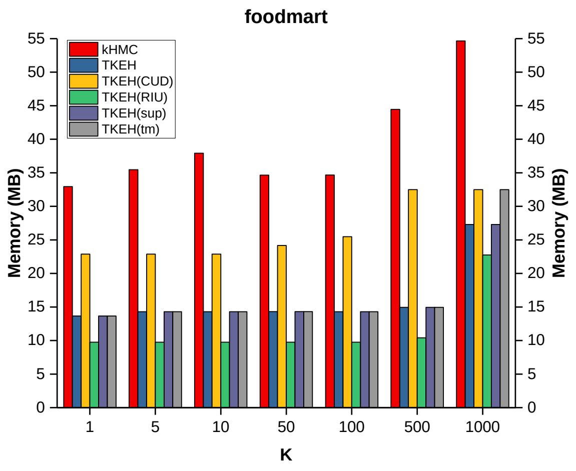

FIGURE 4.9: Memory consumption on foodmart dataset

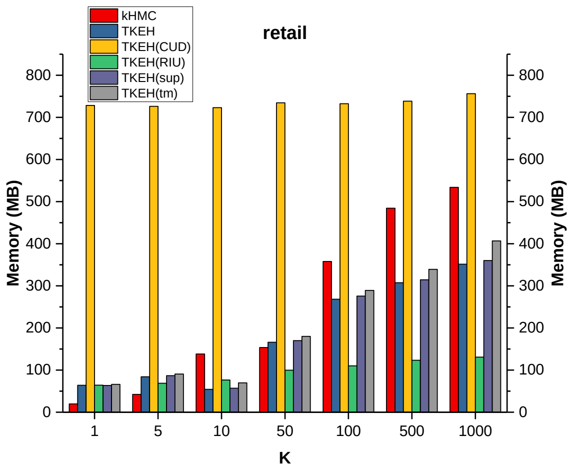

FIGURE 4.10: Memory consumption on retail dataset

4.3.3 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 datasets (foodmart and retail). For all the dense datasets, all the versions of TKEH consume less memory than kHMC as shown in Figures 4.6 to 4.9. In retail dataset, kHMC performs better than TKEH(CUD) and rest all the versions outperforms kHMC as seen in FIGURE 4.10. kHMC consumes huge memory due to the construction of the utility-list during the mining process. TKEH reuses some of it's data structures and reuses same *UAs* to calculate utility of items and upper-bounds. Hence, all the proposed versions of the proposed algorithm consume less memory than kHMC except

TABLE 4.12: Improvements of TKEH over kHMC at the highest value of k ($k = 1000$)

	Runtime					Memory				
Dataset	TKEH	TKEH (CUD)	TKEH (RIU)	TKEH (sup)	TKEH (tm)	TKEH	TKEH (CUD)	TKEH (RIU)	TKEH (sup)	TKEH (tm)
accident	2.657	2.668	2.191	0.114	0.560	1.671	1.676	1.702	1.452	1.471
chess	4.797	2.946	1.849	0.495	0.056	4.914	2.559	1.004	1.153	1.129
foodmart	1.243	2.028	2.350	1.911	1.125	2.002	1.682	2.402	2.002	1.682
mushroom	1.046	153.556	96.861	1.042	1.036	1.633	7.523	3.877	1.531	1.666
retail	0.574	0.561	0.361	1.720	0.430	1.520	0.706	4.081	1.483	1.313

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 as retail.

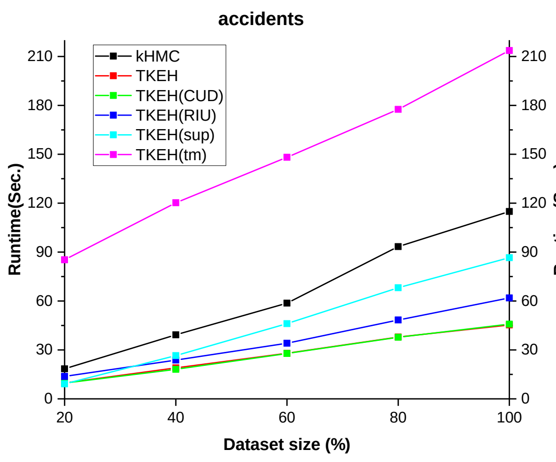

FIGURE 4.11: Runtime scalability of the algorithms on accidents dataset

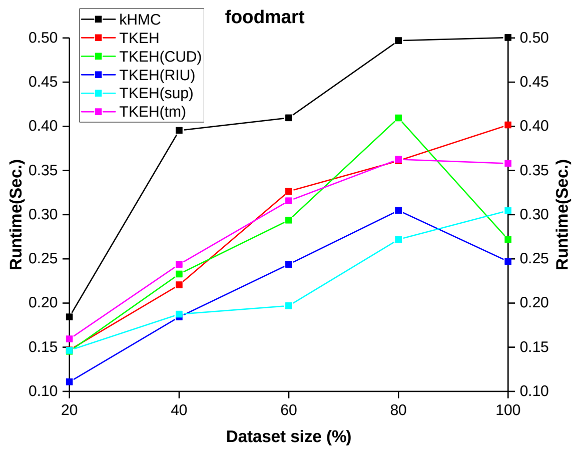

FIGURE 4.12: Runtime scalability of the algorithms on foodmart dataset

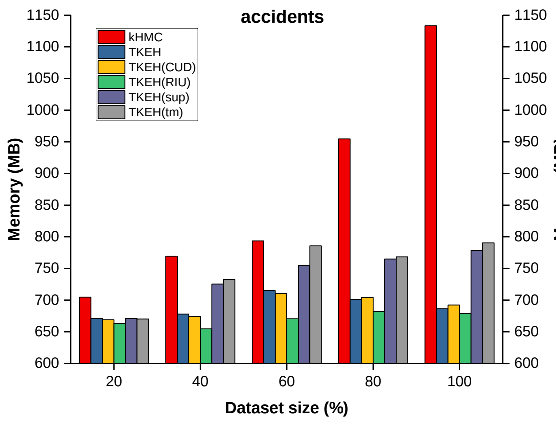

FIGURE 4.13: Memory scalability of the algorithms on accidents dataset

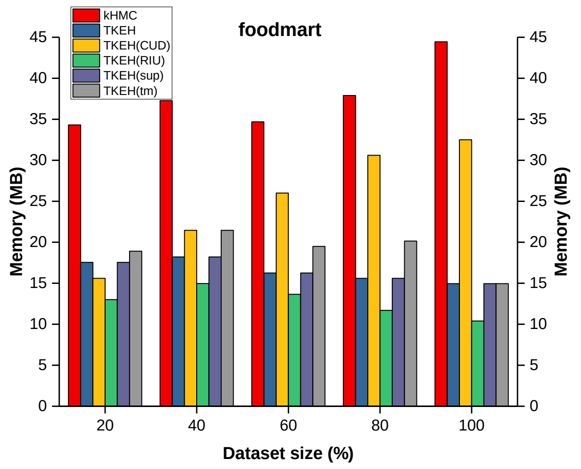

FIGURE 4.14: Memory scalability of the algorithms on foodmart dataset

4.3.4 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 500 to check the scalability comparison of all the algorithm. In order to check the scalability, one dense (accidents) and one sparse (foodmart) dataset are used. All the versions of TKEH consume less runtime and less memory than kHMC. FIGURE 4.11 and 4.12 show that the running time of TKEH increases linearly with increased dataset size. We also observe that the memory usages of TKEH increases linearly when the number of transactions increases as shown in FIGURE 4.13 and 4.14. This indicates that TKEH scales well with the size of dataset.

4.4 Discussion

This chapter utilizes three threshold raising strategies such as RIU, CUD and COV. Transaction merging and utility counting techniques are utilized to reduce the memory requirement and speed up the execution process. The chapter also employs *sup* and EUCP technique to prune the search space. A summary of the performance improvements of all the versions of TKEH over kHMC at the highest *k* values is shown in TABLE 4.12. TABLE 4.12 shows that the proposed algorithm up to 153.556 times faster in execution time and consume up to 4.914 times less memory than kHMC. The results show that TKEH significant improvement in runtime for accident, chess, foodmart and mushroom datasets. kHMC algorithm performs better on retail dataset in runtime. However, the memory improvements of TKEH on all datasets are quite significant. The runtime performance improvements of TKEH, TKEH(CUD), TKEH(RIU) were quite significant compared to kHMC on all the datasets except retail. TKEH(sup) and TKEH(tm) do not give as good results as the other versions of TKEH algorithm. It shows that both the *sup* and *tm* techniques do not perform good individually but they performs better together.

4.5 Summary

In this chapter, we addressed the mining top-k HUIs by utilizing three threshold raising strategies named RIU, CUD and COV. Transaction merging and array-based utility counting techniques are utilized to reduce the memory requirement and speed up the execution process. The proposed algorithm also utilized *sup* and EUCP technique to prune the search space. A summary of the performance improvements of all the versions of TKEH over kHMC at the highest *k* values is shown in TABLE 4.12. In order to understand the working of proposed algorithm, we presented a detail example with dummy transactional dataset. The results show that TKEH significant improvement in runtime for accident, chess, foodmart and mushroom datasets. TKEH algorithm not performs even better than kHMC on retail dataset in runtime. However, the memory improvements of TKEH on all datasets are quite significant. The runtime performance improvements of TKEH, TKEH(CUD), TKEH(RIU) were quite significant compared to kHMC on all the datasets except retail. TKEH(sup) and TKEH(tm) do not give as good results as the other versions of TKEH algorithm. It shows that both the *sup* and *tm* techniques do not perform good individually but they performs better together.