# Chapter 6

# High Utility Itemsets Mining with Negative Utility Value and Length Constraints

In previous chapter, we have developed efficient algorithm named EHIN for mining HUIs with negative utility. Although EHIN is more efficient than FHN. It incurs the problem of generating a large number of candidate itemsets and most of the generated itemsets are very small in size which degrade mining performance and action-ability. In order to overcome these issues, we propose an algorithm named EHNL (**E**fficient **H**igh utility itemsets mining algorithm with **N**egative utility and **L**ength constraints). Although negative utility and constraint-based mining are commonly seen in real-world applications. Mining HUIs with negative utility and length constraints have not yet been proposed in literature. For illustrative example, we utilize datasets presented in TABLE 2.8 and TABLE 2.9 in Chapter 2. The preliminary definitions and properties related to the proposed algorithm are presented below.

*Definition* 6.0.1. (Redefined transaction utility) The redefined transaction utility is denoted by $RTU(T_j)$ for transaction $T_j$ and is computed as $RTU(T_j) = \sum_i^m U(x_i, T_j)$ in which $m$ is the number of items in $T_j$ transaction. If $m \leq max\_length$ then there is no change in $m$, otherwise, $m = max\_length$. To calculate $RTU$, items must be sorted according to descending order to their utility values.

For the running example, $min\_length$ and $max\_length$ constraints are 1 and 4 respectively. $RTU$ of $T_1$ is as $RTU(T_1)$= $U(A,T_1)$ + $U(D,T_1)$ + $U(E,T_1)$ + $U(B,T_1)$ $= 4+4+3+(-6) = 11$ because we calculate $RTU$ by only adding the positive items. $RTU(X) \geq TU(X)$, because $TU$ is the summation of all items in a transaction without any deletion and summation of negative items where $RTU$ is the summation of remaining items after applying $min\_length$ constraints and summation with only positive items. TABLE 6.1 shows the $TU$ and $RTU$ value of each transaction.

TABLE 6.1: Redefined Transaction Utility

| TID | Transaction | TU | Redefined TU |
|---|---|---|---|
| $T_1$ | $A,B,D,E$ | 5 | 11 |
| $T_2$ | $B,C,E$ | 3 | 6 |
| $T_3$ | $B,C,D,E$ | 9 | 15 |
| $T_4$ | $C,D,E$ | 9 | 9 |
| $T_5$ | $A$ | 4 | 4 |
| $T_6$ | $A,B,C,D,E$ | 14 | 16 |
| $T_7$ | $B,C,E$ | -5 | 4 |

The number of items in transaction $T_6$ is more than $max\_length$ threshold. Hence, we calculate $RTU$ by summation of only utility item equal to $max\_length$ threshold. Now $RTU$ of $T_6$ is $RTU(T_6) = U(A)+ U(B)+ U(C)+ U(D) = 4+(-3)+4+8 = 16$. If $max\_length$ threshold is considered as 5 than the $RTU$ is 17 because length of items in $T_6$ meets the $max\_length$ threshold and no need to drop any item.

*Definition* 6.0.2. (Redefined transaction weighted utility). The redefined transaction weighted utility ($RTWU$) of an itemset $X$ is defined as $RTWU(X)$ =$\sum_{X \subseteq T_j \in D} RTU(T_j)$

For example, $RTWU$ value of item $A$ is calculated as, $RTWU(A) = RTU(A,T_1)+ RTU(A,T_5)+ RTU(A,T_6) = 11+4+16$ =31. $RTWU$ cannot be less than $TWU$ and actual utility value of itemsets, hence can be used as an upper bound.

*Definition* 6.0.3. (Ordering of items). The items in the transactions are sorted according to the $\succ$ total order as $RTWU$ ascending order. For the running example, the sorted items are as $A \succ C \succ D \succ E \succ B$. For the sample dataset, the ordered set of items are provided in TABLE 6.2.

*Definition* 6.0.4. (Extension of an item). Let the itemset be $\alpha$ and set of items that used to extend the itemset $\alpha$ is denoted as $E(\alpha)$ and is defined as $E(\alpha) = \{x \mid x \in I \wedge x \succ i, \forall i \in \alpha\}$.

*Definition* 6.0.5. (Extension of an itemset). Let the itemset be $\alpha$ and $Y$ is an extension of $\alpha$ that appears in a sub-tree of $\alpha$ in the set-enumeration tree. If $Y = \alpha \cup X$ for an itemset $X \in 2^{E(\alpha)}$. The maximum length of an itemset after extension must not be greater than $max\_length$ threshold.

Here, $Y$ is a single-item extension of $\alpha$ that is a child of $\alpha$ in the set-enumeration tree. If $Y = \alpha \cup \{X\}$ for an item $X \in E(\alpha)$.

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

*Definition* 6.0.6. (Extension of negative item). Let the itemset be $\alpha$ which can be extended to itemset $X$, where $Y = \alpha \cup \{X\}$ and $X$ is a set of items with negative utility.

**Rationale**. $\alpha \cup \{X\}$ only occurs in less or equal than the number of transactions containing itemset $\alpha$. Extensions of itemset $\alpha$ with positive utility item may be less or equal or greater than the utility of itemsets $\alpha$. But extensions of itemset $X$ with negative utility item always decrease the utility of itemsets as proposed by the *Property 2.5.1*. Hence, from the above properties, if an itemsets $U(\alpha) > min\_util$ then the negative utility itemsets $X$ is added to itemsets $\alpha$. The number of extended itemset must not be greater than $max\_length$ threshold according to length constraints. The utility of extended itemset is still greater or equal to $min\_util$ then the itemset is HUIs.

*Definition* 6.0.7. (High utility itemsets). An itemset $X$ is called a high utility itemsets if and only if $U(X) \geq min\_util$ where $min\_util$ is a user-defined minimum utility threshold.

For the running example, TABLE 6.3 shows HUIs where $min\_length$, $max\_length$ and $min\_util$ are 2, 3 and 15 respectively.

Two-phase based algorithms suffer from multiple dataset scans and generate lots of candidates. In order 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 data structure to store the information of items and the remaining utility based pruning strategy to prune the search space [24, 25, 26, 27, 28]. The proposed algorithm EHNL do not utilize utility-list and utility-list based pruning strategy. We introduce here utility-list for comparison with the our proposed strategies to prune the search space.

*Definition* 6.0.8. (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, that is $RU(X,T_j)$

*Definition* 6.0.9. (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, 25, 27].

*Property* 6.0.1. (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 it's supersets are low utility itemsets. Otherwise, the itemset is HUIs. The detail and proof of the remaining utility upper bound ($REU$) based upper bound are given in [24, 30].

*Definition* 6.0.10. (Largest length in a transaction). For the itemset $X$ and transaction $T_j$, if $V(T_j,X) = \{v_1, v_2, \ldots, v_k\}$ is the set of items occurring in $T_j$ then itemsets $X$ can be extended, i.e. $V(T_j,X) = \{v \in T_j | v \succ x, \forall x \in X\}$. The $max\_length$ constraint set has the maximum length of the itemset $X$. The $max\_length$ also describes the length of the item that can be added to an itemset $X$ as $maxExtend(X) = max\_length - |X|$ where $|X|$ defines the number of items in $X$. $maxExtend$ is used while extending the itemsets. The largest utility value with $X$ for transaction $T_j$ is denoted as $L(T_j,X)$ [47].

**Problem statement**. We determine all the itemsets whose utility is not less than user-defined $min\_util$ threshold and length of itemsets are not less than $min\_length$ and not greater than $max\_length$ thresholds.

## 6.1 The Proposed Algorithm

In this section, we give a step-by-step analysis of the proposed algorithm named EHNL. Section 6.1.1 describes the dataset cost reduction techniques to reduce the dataset scanning. Section 6.1.2 describes the pruning strategies. Section 6.1.3 introduces array-based utility counting technique. Finally, Section 6.1.4 gives the pseudo-code and explanation of the proposed algorithm.

### 6.1.1 Efficient Dataset Scanning Techniques

The proposed algorithm reduces the dataset scanning costs by reducing the dataset size using dataset projection and transaction merging techniques. We check *min_length* constraint before apply dataset scanning techniques.

*Definition* 6.1.1. (Transaction merging). Let the identical transaction be $T_1$, $T_2$, $T_3$, $T_n$ replaced by a new transactions $T_M$= $T_1$= $T_2$ = $T_3$ = $T_n$ (Identical transactions may not contain the same quantity values of each item). And quantity of each item in these identical transactions is $x \in T_M$ is defined as $IU(x,T_M) = \sum_{i=1,...,n} IU(x,T_i)$.

In the running example, transaction $T_2$ and $T_7$ are identical and after transaction merging we get a new transaction $T_{2\_7}$. where $IU(C,T_M) = 7$, $IU(E,T_M) = 3$ and $IU(B,T_M) = 4$.

We need to merge the transaction in projected datasets. Projected transactions merging produces higher dataset reduction than original transaction merging because projected transactions are smaller than original transactions. Therefore, the projected transactions could be more likely to be identical.

*Definition* 6.1.2. (Projected dataset). The projection of a transaction $T_j$ using an itemset $\alpha$ is denoted as $\alpha - T$ and is 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 as $\alpha - D$ and is defined as the multi-set $\alpha - D = \{\alpha - T \mid T \in D \wedge \alpha - T \neq \emptyset\}$.

*Definition* 6.1.3. (Projected transaction merging). Let the identical transactions be $T_1$, $T_2$, $T_3$, $T_n$ in the projected dataset $\alpha - D$ replaced by a new transaction $T_M$= $T_1$= $T_2$ = $T_3$ = $T_n$. And quantity of these identical transactions $x \in T_M$ is defined as $IU(x,T_M) = \sum_{i=1,...,n} IU(x,T_i)$.

The projected dataset $\alpha - D$ for $\alpha = \{D\}$ contains the identical projected transactions as $\alpha - T_1 = \{E,B\}$, $\alpha - T_3 = \{E,B\}$, $\alpha - T_4 = \{E\}$ and $\alpha - T_6 = \{E,B\}$ is further merged by a new projected transaction using offset pointers in the original dataset. Transactions $\alpha - T_1$, $\alpha - T_3$ and $\alpha - T_6$ can be replaced by a new transaction $T_{1\_3\_6} = \{E,B\}$ where

$IU(E,T_{1\_3\_6}) = 6$ and $IU(B,T_{1\_3\_6}) = 5$. The example also shows that when as we mine long itemsets the $\alpha - D$ become shorter and shorter.

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 each other. But this technique to compare all the transactions to each other is not an efficient technique. To overcome this problem, we follow the same transaction sorted technique $\succ_T$ proposed in [28]. This sorting technique is not computationally expensive and performs only once.

*Definition* 6.1.4. (Total order on the transactions). For the dataset *D*, the total order $\succ_T$ is defined as lexicographical order when the transactions are being read backward. For more details and complexity of total order $\succ_T$ on the transactions, we can see in [28].

We sort the original dataset according to a new total order $\succ_T$ on the transactions. Sorting of transactions has been performed using *RTWU* before merging. This sorting is done in linear time and performs only once, so the cost of the sorting is negligible. This sorted dataset puts up the following property. The identical transactions always appear consecutively in the projected dataset $\alpha - D$. This property binds because we read the transaction from backward. The projection also plays an important role to remove the smallest items of the transaction to the $\succ_T$ order.

### 6.1.2 Pruning Strategies

We have so far introduced one novel tighter upper-bound *RTWU* which considers negative utility and length constraints for reducing the search space. Now we are going to introduce two new strategies to prune the search space. These strategies are more efficient and much tighter upper-bound on the utility of itemsets. These strategies are calculated when the tree is traversed in dept-first search manner.

#### 6.1.2.1 Prune search space using Redefined Sub-tree Utility

*Definition* 6.1.5. (Redefined 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 into the sub-tree. To get the *RSU* of the item $x$ w.r.t.[1] $\alpha$ is $RSU(\alpha, x) = \sum_{T_j \in (\alpha \cup \{x\})}[U(\alpha, T_j) + U(x, T_j) + \sum_{i \in T_j \wedge i \in E(\alpha \cup \{x\})} U(i, T)]$.

Moreover in the running example $\alpha = \{C\}$, we have $RSU(\alpha, D) = (1+12+2) + (2+4+3) + (4+8+1) = 37$ and $RSU(\alpha, E)$ $(5+1+(-3)) = +(1+2+(-6)) + (2+3+0) + (4+1+(-3)) + (2+2+(-9)) = 23$. We added only positive utility value according to the *Property 2.5.2*.

*Property* 6.1.1. (*RSU* Overestimation). For an itemset $\alpha$ and an item $x \in E(\alpha)$, the utility value of $RSU(\alpha, x) \geq U(\alpha \cup \{x\})$ and accordingly. Here $RSU(\alpha, x) \geq U(x)$ keeps the extension $x$ of $\alpha \cup \{x\}$. The proof of sub-tree utility is presented in [28].

The relationships between the redefined upper-bounds ($RTWU$ and $RSU$) and the state-of-the-art upper bound $REU$ are given as follows.

$REU$ upper-bound is presented in [24, 27] and is calculated with utility-list data structure. The proposed $RSU$ is a redefined upper-bound of $SU$. $SU$ is proposed in [28] which works only for positive utility items. $SU$ calculates the utility of itemset by dept-first search in tree. Similarly, redefined upper-bound $RSU$ calculates the utility at itemset $\alpha$ by dept-first search rather than at the child itemset of $\alpha$. FIGURE 5.2 shows the difference between the proposed $RSU$ upper-bound and $REU$. The figure shows that if an itemset $\alpha$ with an item $x$ have less utility than $min\_util$ then the itemset with their child is pruned for the $RSU$ upper-bound. And in $REU$ upper-bound, if the itemset $\alpha$ with an item $x$ has less utility than $min\_util$ then the only child nodes are pruned as shown in FIGURE 5.2. The relationship between the $RWTU$ and $TWU$ is already explained by the *Property 2.5.4*.

In remaining chapter, we refer to items having $RSU$ and $RTWU$ as *Primary* and *Secondary* respectively.

*Definition* 6.1.6. (*Primary* and *Secondary* items). For an itemset $\alpha$, the *Primary* items of itemset $\alpha$ are the set of items, $Primary(\alpha) = \{x \mid x \in E(\alpha) \wedge RSU(\alpha, x) \geq min\_util\}$. The itemset $\alpha$ is the set of items $Secondary(\alpha) = \{x \mid x \in E(\alpha) \wedge RTWU(\alpha, x) \geq min\_util\}$. The $RTWU(\alpha, x) \geq RSU(\alpha, x)$, so $Primary(\alpha) \subseteq Secondary(\alpha)$.

[1]with respect to

| | $UA[A]$ | $UA[B]$ | $UA[C]$ | $UA[D]$ | $UA[E]$ |
|---|---|---|---|---|---|
| **(Step 1)** Initialization | 0 | 0 | 0 | 0 | 0 |
| **(Step 2)** After reading $T_1$ | 11 | 11 | 0 | 11 | 11 |
| **(Step 3)** After reading $T_2$ | 11 | 17 | 6 | 11 | 17 |
| **(Step 4)** After reading $T_3$ | 11 | 32 | 21 | 26 | 32 |
| **(Step 5)** After reading $T_4$ | 11 | 32 | 30 | 35 | 41 |
| **(Step 6)** After reading $T_5$ | 15 | 32 | 30 | 35 | 41 |
| **(Step 7)** After reading $T_6$ | 32 | 49 | 47 | 52 | 58 |
| **(Step 8)** After reading $T_7$ | 32 | 53 | 51 | 52 | 62 |

FIGURE 6.1: Calculate $RTWU$ using utility-array

#### 6.1.2.2 Pruning using Length Constraints

*Definition* 6.1.7. (Length Constraints) If $|T_j|< min\_length$ then the transaction $T_j$ is removed from the dataset $D$. And if $|T_j|\geq max\_length$ then the maximum items consider calculating $TU$ where transaction $T_j$ is equal to $max\_length$. To remove very small items, $min\_length$ constraint plays an important role.

The proposed technique prunes candidates which do not fulfill the length constraints. We initially remove the transactions which do not fulfill the minimum length constrains. We compare the length of candidates HUIs with $max\_length$ constraint. If the length of any candidate HUIs is equal to upper-length constraint then recursion of adding extended itemsets are stopped. Otherwise, new items are added to enhance the length of items using recursive functions explained in next subsection. The proposed algorithm is inspired by a similar length-based HUIs mining algorithm FHM+.

### 6.1.3 Calculate Upper Bounds using Utility Array

Previously, we presented redefined upper-bounds to prune the search space. Now we present an array-based technique to calculate the upper-bounds in the linear time.

*Definition* 6.1.8. (Utility Array). For the set of items $I$ appeared in a dataset $D$, the $UA$ is an array of length $|I|$ that have an entry denoted as $UA[x]$ for each item $x \in I$. Each entry is called $UA$ that is used to store a utility value.

Initially, we delete the transactions based on $min\_length$. After that, the remaining length of transactions is $|I|$. So the $UA$ calculates upper-bounds, as follows.

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

For example, $RTWU$ of the sample transactional dataset is shown in FIGURE 6.1. The length of $UA$ is set equal to the number of items in the transactional dataset. The $RTWU$ calculation process initially sets the $UA$ with the zeros as shown in the Step 1 in FIGURE 6.1. Step 2 reads the transaction $T_1$ and updates the $UA$ with $RTU$. Transaction $T_1$ has the items $A, B, D$ and $E$, hence, the only respective position of $UA$ is updated with $RTU$ value 11. Step 3 reads the transaction $T_2$ and updates the $UA$ with $RTU$ value 6. Transaction $T_2$ has the items $B, C$ and $E$ and hence the respective positives in $UA$ is updated with the $RTU$ value 6 as shown the step 3 in FIGURE 6.1. Similarly all the transactions are read and the $UA$ are updated. Finally we find the $RTWU$ value for each item.

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

**Calculating Length($\alpha$):** $UA$ is initialized to 0. Then the $UA[x]$ for each item $x \in T_j$ $\cap E(\alpha)$ is calculated as $UA[x] = UA[x] + 1$ for each transaction $T_j$ in the dataset $D$. After the dataset scanning, the $UA[x]$ contains $Length(\alpha, x)$ $\forall x \in E(\alpha)$ where each item $x \in E(\alpha)$.

### 6.1.4 EHNL Algorithm

This section presents a novel algorithm for mining HUIs with negative utility and length constraints named EHNL. It utilizes several novel ideas explained in the previous sections. **Algorithm 11** takes a transactional dataset $D$ and three user-defined thresholds as parameters. It outputs the set of HUIs. Line 1 takes the itemset $\alpha$ as an empty set. Line 2 initializes the $\eta$ a sets with set of negative utility items. In line 3-13, **foreach** loop is used to delete the transactions which do not fulfill the $min\_length$ threshold and calculate the $RTU$ for the dataset $D$. Line 4 initializes $\iota$ with the length of transaction $T_j$.

```
Algorithm 11: EHNL algorithm
Input: $D$: a tranasctional dataset, user-specified threshold: $min\_util$, $min\_length$ and
       $max\_length$
Output: High utility itemsets (HUIs) in $D$
1  $\alpha \leftarrow \emptyset$;
2  $\eta \leftarrow$ set of negative utility items in $D$;
3  foreach $T_j \in D$ do
4      $\imath \leftarrow |T_j|$                                   ▷ no. of items in $T_j$
5      if $\imath < min\_length$ then
6          Remove $T_j$ from $D$;
7      else if $\imath > max\_length$ then
8          sort items of $T_j$ in decreasing order;
9          $\beta \leftarrow 0$
10         $count \leftarrow 0$
11         for $count \leq max\_length$ do
12             $\beta \leftarrow \beta + U(count)$;
13         $U(T_j) \leftarrow \beta$;                          ▷ update $RTU(T_j)$ with $\beta$
14 Scan dataset $D$ and compute $RTWU(\alpha,x)$ for item $x \in I$, using $UA[x]$.   ▷ SeeDefinition
     6.0.2
15 $Secondary(\alpha) = \{x \mid x \in I \wedge RTWU(\alpha,x) \geq min\_util\}$.
16 Let $\succ$ be the total order of $RTWU$ increasing values on $Secondary(\alpha)$
17 Scan $D$, remove item $x \notin Secondary(\alpha)$ from the transactions $T_j$ and delete empty
     transactions $T_j$;
18 Sort all the remaining transactions in $D$ according to $\succ_T$ with positive utility items
     followed by negative utility items
19 Assign offsets to negative items in each transaction in $D$.
20 Scan dataset $D$ and compute the $RSU(\alpha,x)$ of each item $x \in Secondary(\alpha)$, using $UA[x]$;
21 $Primary(\alpha) = \{x \mid x \in Secondary(\alpha) \wedge RSU(\alpha,x) \geq min\_util\}$;
22 $Search\_pos(\alpha, D, Primary(\alpha), Secondary(\alpha), min\_util, max\_length)$;
23 return HUIs;
```

Line 5 checks the length of transaction $T_j$ with *min_length* threshold. If transaction $T_j$ has less items than *min_length* then the transaction $T_j$ is removed from the dataset $D$ (line 6). Line 7 checks the length of $T_j$ with *max_length*, if the length is higher than *max_length* threshold then line 8 sorts the items of $T_j$ in decreasing order to find the $RTU$. Line 9 and line 10 initialize the variables $\beta$ and *count* with zero. Line 11 calls the **for** loop and calculates the utility of the transaction $T_j$ (in line 12). Line 13 updates the utility of transaction with the value of $\beta$ where $\beta$ represents the utility value of transaction $T_j$ where maximum items in $T_j$ are equal to the *max_length*. Line 14 scans the dataset $D$ and calculates $RTWU$ for all positive items in $D$. Line 15 determines the *Secondary* items of $\alpha$ using *Definition 6.1.6*. Line 16 sorts the items of *Secondary* set by using *Definition 6.0.3*. Line 17 scans the dataset $D$ and deletes the items which are not the member of the *Secondary* set. After removal of these items some transactions become empty, so we drop these empty transactions. Line 18 sorts the remaining transactions according to $\succ_T$ order where positive utility items are followed by negative utility items. Line 19 connects negative utility items in each transaction with the help of offset pointer. The offset pointer keeps the address of first negative item. Line 20 scans the dataset $D$ and calculates the $RSU$ value of each Secondary item with itemset $\alpha$. Line 21 determines the *Primary* items of $\alpha$ using *Definition 6.1.6*. Line 22 calls the recursive algorithm *Search_pos* (**Algorithm 12**) to extend the itemset $\alpha$ with positive items by performing dept-first search.

**Algorithm 12:** The *Search_pos* procedure

```
Input: $\alpha$ : an itemset, $\alpha - D$ : a projected dataset, $Primary(\alpha)$ : the primary items of $\alpha$,
       $Secondary(\alpha)$: the secondary items of $\alpha$, $min\_util$, $max\_length$.
Output: The set itemsets which are extensions of $\alpha$ with positive items and satisfies
        $max\_length$ threshold
1  if $| \alpha | < max\_length$ then
2      foreach item $x \in Primary(\alpha)$ do
3          $\beta \leftarrow \alpha \cup \{x\}$                              ▷ See Definition 6.0.5
4          $\iota \leftarrow | \beta |$                                        ▷ no. of items in $\alpha$
5          Scan $\alpha - D$, compute $U(\beta)$ and create $\beta - D$;
6          if $U(\beta) \geq min\_util$ then
7              Output $\beta$
8          if $U(\beta) > min\_util$ and $\iota < max\_length$ then
9              Search_neg($\eta$, $\beta$ , $\beta - D$ , $min\_util$, $max\_length$);
10         Scan $\beta - D$, compute $RSU(\beta,x)$ and $RTWU(\beta,x)$ where item $x \in$
             $Secondary(\alpha)$, using two UAs
11         $Primary(\beta) = \{x \in Secondary(\alpha) \mid RSU(\beta,x) \geq min\_util\}$
12         $Secondary(\beta) = \{x \in Secondary(\alpha) \mid RTWU(\beta,x) \geq min\_util\}$
13         if $\iota < max\_length$ then
14             Search_pos($\beta$, $\beta - D$, $Primary(\beta)$, $Secondary(\beta)$, $min\_util$, $max\_length$);
```

**Algorithm 12** takes current itemset $\alpha$ to be extended with positive utility items. Line 1 initializes the variable $\iota$ with the length of itemset $\alpha$. Line 2 checks the length of itemset $\alpha$, if the length of itemset $\alpha$ is larger than *max_length* threshold, the process returns back

**Algorithm 13:** The *Search_neg* procedure

**Input:** $\eta$: set of promising negative items, $\beta$: an itemset, $\beta - D$: the projected dataset, $min\_util$ and $max\_length$

**Output:** The set itemsets which are extensions of $\beta$ with negative items.

```
1  $\iota \leftarrow |\beta|$                                          ▷ no. of items in $\beta$
2  if $\iota < max\_length$ then
3      foreach each item $x \in \eta$ do
4          $\gamma = \beta \cup \{x\}$;                    ▷ Extension of $\beta$ with negative item.
5          Scan $\beta - D$, compute $U(\gamma)$ and create $\gamma - D$.
6          if $U(\gamma) \geq min\_util$ then
7              Output $\gamma$
8          Calculate $RSU(\gamma,x)$ for all item $x \in \eta$ by scanning $\gamma - D$ once, using the $UA$.
9          $Primary(\gamma) = \{x \in \eta \mid RSU(\gamma,x) \geq min\_util\}$.
10         $Search\_neg(Primary(\gamma), \gamma, \gamma - D, min\_util, max\_length)$
```

to the calling algorithm. Otherwise, itemset $\alpha$ needs to extend more. Line 3-13 extend the itemset $\alpha$ with the items which are member of the *Primary* set. Line 4 adds an item $x$ into itemset $\alpha$ where item $x$ is a member of *Primary* set. Line 5 scans the projected dataset $\alpha - D$ and calculates the utility of extended itemset $\beta$, Thereafter, projected dataset $\beta - D$ is created. Line 6 checks the utility of itemset $\beta$ with $min\_util$ threshold. If the utility of itemset $\beta$ is not less than $min\_util$ threshold then the itemset $\beta$ is HUIs (line 7). Line 8 checks the condition, if the utility of itemset $\beta$ is greater than $min\_util$ threshold then *Search_neg* (**Algorithm 13**) is called to extend the itemset $\beta$ with negative utility items (line 9). Line 10 scans the projected dataset $\beta - D$ and calculates $RTWU$ and $RSU$ value of items with the itemset $\beta$. Line 11 and line 12 determine the *Primary* and *Secondary* items of $\beta$ respectively using *Definition 6.1.6*. Line 13 calls the recursive algorithm *Search_neg* to again extend the itemset $\beta$ by performing dept-first search.

**Algorithm 13** takes current itemset $\beta$ to be extended with negative utility items. Line 1 initializes the variable $\iota$ with the length of itemset $\beta$. Line 2 checks the length of itemset $\alpha$, if the length of itemset $\alpha$ is not less than $max\_length$ threshold, the process returns back to the calling algorithm. Otherwise, itemset $\beta$ needs to extend with negative utility item. Line 3-10 extend the itemset $\beta$ with the negative utility items which are member of set $\eta$. Line 4 adds an item $x$ into itemset $\beta$ where item $x$ is a member of set $\eta$. Line 5 scans the projected dataset $\beta - D$ and calculates the utility of extended itemset $\gamma$. Thereafter, projected dataset $\gamma - D$ is created. Line 6 checks the utility of itemset $\gamma$ with $min\_util$

threshold. If the utility of itemset $\gamma$ is not less than $min\_util$ threshold then the itemset $\gamma$ is HUIs (line 7). Line 8 calculates the *RSU* of each the member items of set $\eta$ with itemset $\gamma$. Line 9 determines the *Primary* set for the itemset $\gamma$. Finally, recursive procedure *Search_neg* is called to further extend itemset with the negative utility items.

### 6.1.5 An Illustrative Example

In this section, a simple illustrative example is given to explore HUIs mining process. Let us assume that there are seven transactions in an example dataset as shown in TABLE 2.8 and there are five items which appear with their internal quantity. We further assume that the external utility or profit value of every single item is predefined as in TABLE 2.9. In order to understand more about the length constraints and proposed algorithm, here we assume $min\_util$ $min\_length$ and $max\_length$ threshold as 15, 2 and 3 respectively.

TABLE 6.2: Sorted items according to $\succ$ total order with their $RTWU$ value

| $T_id$ | Transaction | Utility ($U$) | $RTU$ |
|---|---|---|---:|
| $T_1$ | $A,D,E,B$ | 4, 4, 3, -6 | 11 |
| $T_2$ | $C,E,B$ | 5, 1, -3 | 6 |
| $T_3$ | $C,D,E,B$ | 1, 12, 2, -6 | 15 |
| $T_4$ | $C,D,E$ | 2, 4, 3 | 9 |
| $T_6$ | $A,C,D,E,B$ | 4, 4, 8, 1, -3 | 16 |
| $T_7$ | $C,E,B$ | 2, 2, -9 | 4 |

TABLE 6.3: Final HUIs of the running example

| Itemset | Utility | Itemset | Utility |
|---|---:|---|---:|
| $\{A,C,D\}$ | 16 | $\{C,D,E\}$ | 37 |
| $\{A,D\}$ | 20 | $\{C,E\}$ | 23 |
| $\{A,D,E\}$ | 24 | $\{D,E\}$ | 37 |
| $\{C,D\}$ | 31 | $\{D,E,B\}$ | 15 |
| $\{C,D,B\}$ | 16 | $--$ | $--$ |

The proposed algorithm uses *Definition 6.0.1* to overestimate the transaction utility. TABLE 6.2 shows $RTU$ values for this example. The transaction $T_5$ is removed because

TABLE 6.4: Dataset characteristics.

| **Dataset** | **# of transactions** | **# of distinct 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 |
| T10I4D100K | 100000 | 870 | 10.1 | 29 | Sparse |
| T40I10D100K | 100000 | 942 | 39.6 | 77 | Dense |

this transaction has less items than the $min\_length$ constraint. And the $RTU$ value of the transaction $T_6$ now is 16 instead of 17 because $max\_length$ constraint is assumed here 3. $RTU$ value is calculated by summation by only three items instead to four items for this example. Then the proposed algorithm calculates $RTWU$. For this example, $RTWU$ values are $A = 27$, $B = 45$, $C = 43$, $D = 44$ and $E = 54$.

$RTWU$ values of items are not less than *min_util* then the items are considered as *Secondary* itemset. The items in *Secondary*$(\alpha)$ = $\{A,B,C,D,E\}$. After this, all items are sorted according to the $\succ$ total order. The second column of TABLE 6.2 shows the sorted items according to $\succ$ total order. And the third column shows the utility values of the sorted items. Thereafter, the items are removed which are not the elements of *Secondary* set. At the same time, empty transactions are removed from the dataset. After that, the proposed algorithm scans dataset again and calculates *RSU* of all itemsets. The items of *RSU* which are having utility not less than *min_util* are in *Primary* set. Only the items of the *Primary* set are used to explore by depth-first search. **Algorithm 12** finds descendant nodes in sub-tree using dept-first search. The *Search_neg* algorithm is recursively called to extend all items with the positive items as well as with the negative items. After executing all the algorithms, we found the final HUIs of this example. TABLE 6.3 shows the final HUIs with their utility values while assuming *min_util* $min\_length$ and $max\_length$ threshold as 15, 2 and 3 respectively.

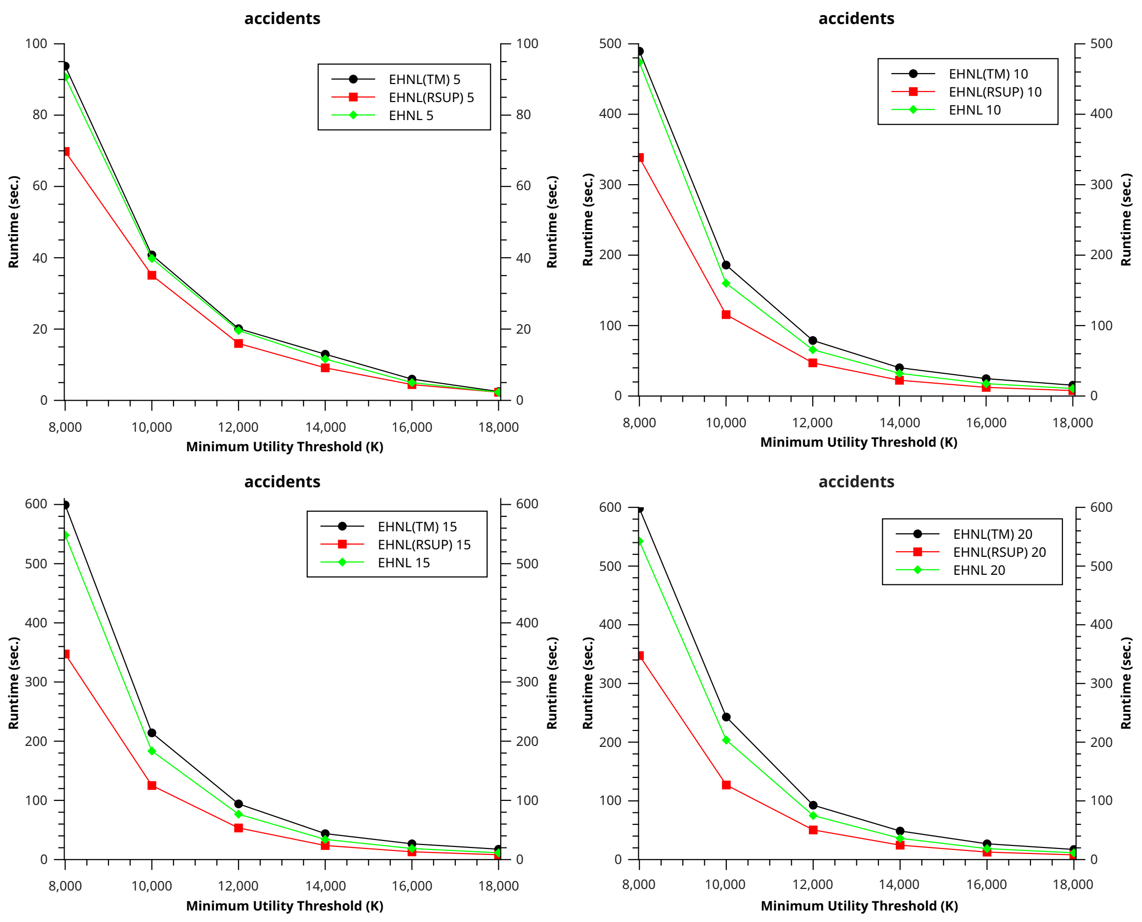

FIGURE 6.2: Runtime performance on accidents dataset

TABLE 6.5: Number of candidates on accidents dataset

| | EHNL | | | | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $min\_util(k)$ | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 8000 | 594 | 1689 | 1827 | 1827 | 594 | 1689 | 1827 | 1827 | 919 | 4239 | 5147 | 5147 |
| 10000 | 243 | 919 | 986 | 1002 | 243 | 919 | 986 | 1002 | 341 | 2483 | 2995 | 3110 |
| 12000 | 174 | 514 | 580 | 580 | 174 | 514 | 580 | 580 | 218 | 1240 | 1720 | 1720 |
| 14000 | 141 | 322 | 338 | 368 | 141 | 322 | 338 | 368 | 209 | 746 | 854 | 1033 |
| 16000 | 47 | 216 | 226 | 226 | 47 | 216 | 226 | 226 | 99 | 528 | 591 | 591 |
| 18000 | 6 | 135 | 144 | 144 | 6 | 135 | 144 | 144 | 12 | 318 | 373 | 373 |

TABLE 6.6: Number of candidates on chess dataset

| | EHNL | | | | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $min\_util(k)$ | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 60 | 14409 | 52489 | 52584 | 52584 | 14409 | 52489 | 52584 | 52584 | 41142 | 232595 | 234229 | 234229 |
| 80 | 8549 | 20513 | 20513 | 20513 | 8549 | 20513 | 20513 | 20513 | 30046 | 120564 | 120721 | 120721 |
| 100 | 5246 | 9233 | 9416 | 9416 | 5246 | 9233 | 9416 | 9416 | 23304 | 65845 | 68487 | 68487 |
| 120 | 2549 | 4727 | 4727 | 4727 | 2549 | 4727 | 4727 | 4727 | 12213 | 39832 | 39832 | 39832 |
| 140 | 1487 | 2471 | 2476 | 2629 | 1487 | 2471 | 2476 | 2629 | 8392 | 21754 | 21214 | 24896 |
| 160 | 912 | 1132 | 1441 | 1595 | 912 | 1132 | 1441 | 1595 | 5431 | 8735 | 13055 | 16247 |

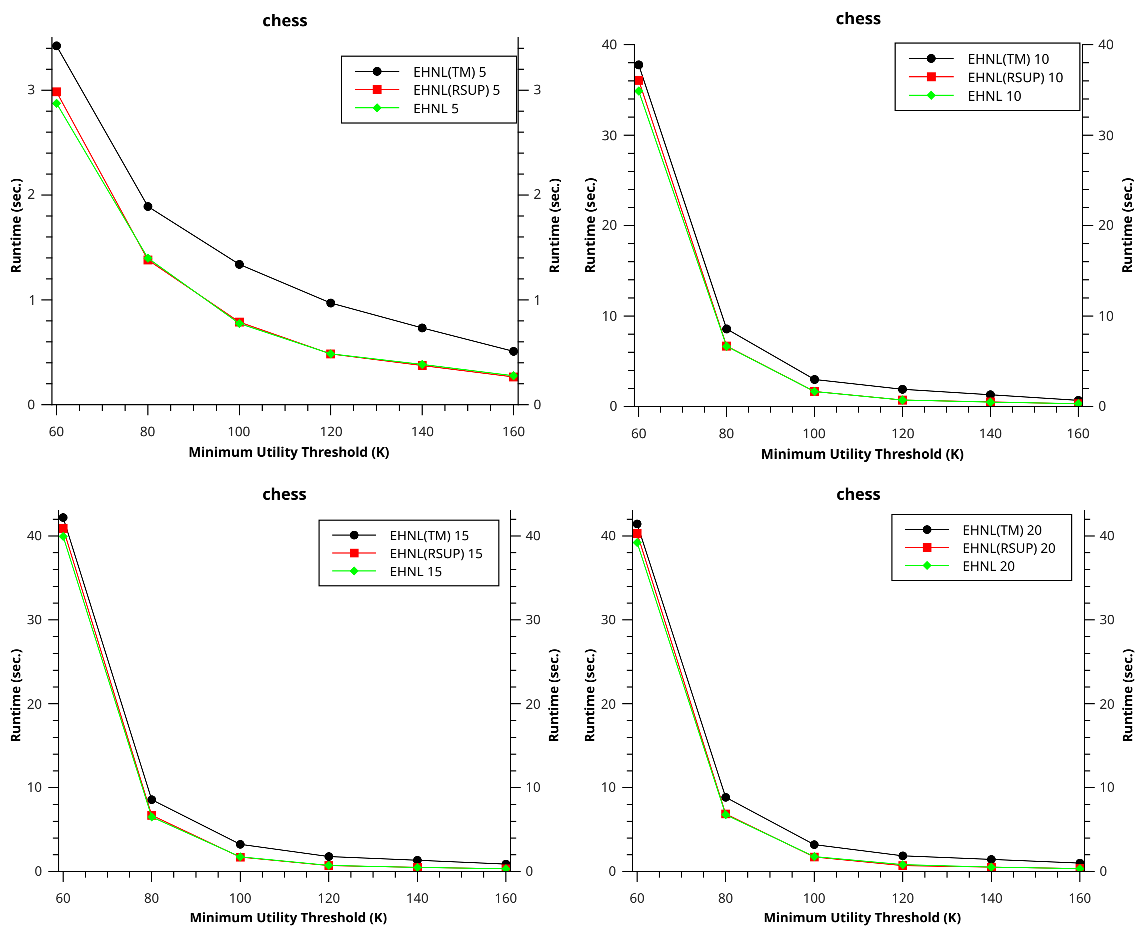

FIGURE 6.3: Runtime performance on chess dataset

TABLE 6.7: Number of candidates on mushroom dataset

| | EHNL | | | | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *min_util*$(k)$ | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 250 | 193 | 579 | 599 | 599 | 193 | 579 | 599 | 599 | 914 | 3277 | 3933 | 3828 |
| 300 | 43 | 237 | 253 | 253 | 43 | 237 | 253 | 253 | 190 | 1759 | 2464 | 2464 |
| 350 | 17 | 78 | 84 | 84 | 17 | 78 | 84 | 84 | 37 | 1197 | 1382 | 1382 |
| 400 | 8 | 28 | 34 | 34 | 8 | 28 | 34 | 34 | 23 | 229 | 878 | 878 |
| 450 | 3 | 19 | 23 | 23 | 3 | 19 | 23 | 23 | 7 | 150 | 315 | 315 |
| 500 | 3 | 9 | 14 | 14 | 3 | 9 | 14 | 14 | 6 | 68 | 113 | 103 |

## 6.2 Experimental Results

In this section, we check the performance of our proposed algorithm (EHNL), We implemented the proposed algorithm by extending the open-source java library [77]. Experiments were performed on a PC with an Intel Core-i7-6700 machine, 3.40 GHz CPU with 8 GB of memory, running on a Windows 10 Pro (64-bit Operating System).

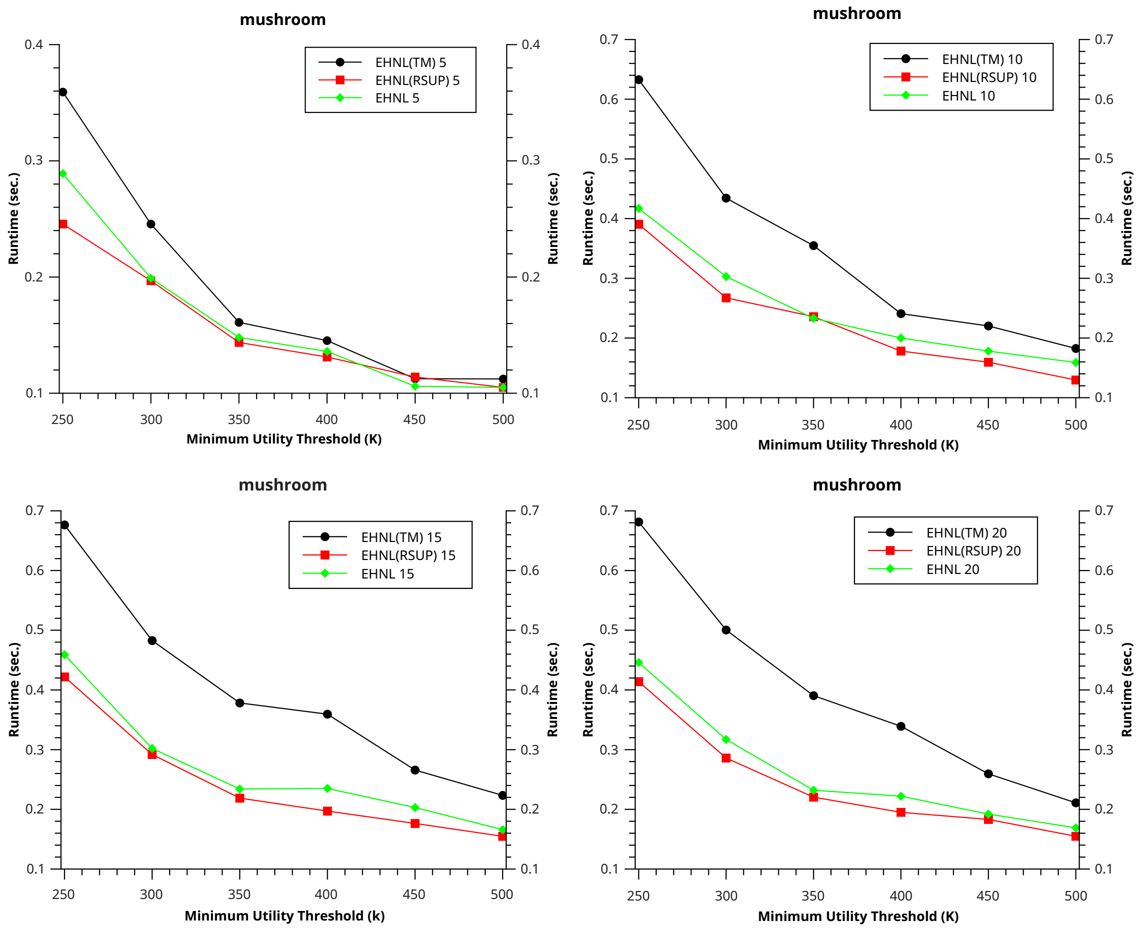

FIGURE 6.4: Runtime performance on mushroom dataset

TABLE 6.8: Number of candidates on T10I4D100K dataset

| | EHNL | | | | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *min_util*$(k)$ | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 20 | 864 | 885 | 887 | 888 | 864 | 885 | 887 | 888 | 2108 | 2196 | 2219 | 2225 |
| 25 | 593 | 616 | 621 | 621 | 593 | 616 | 621 | 621 | 1430 | 1596 | 1624 | 1641 |
| 30 | 428 | 450 | 462 | 465 | 428 | 450 | 462 | 465 | 1035 | 1149 | 1191 | 1212 |
| 35 | 338 | 355 | 361 | 363 | 338 | 355 | 361 | 363 | 774 | 857 | 884 | 900 |
| 40 | 266 | 284 | 292 | 296 | 266 | 284 | 292 | 296 | 572 | 664 | 714 | 728 |
| 45 | 223 | 237 | 242 | 244 | 223 | 237 | 242 | 244 | 471 | 552 | 588 | 606 |

For the performance test, the following benchmark databases in SPMF [77] were chosen: accidents, chess, mushroom, T10I4D100K and T40I10D100K. The accidents, chess and mushroom are real and dense datasets. The others are synthetic datasets. TABLE 6.4 shows the detailed characteristics of all the datasets. To ensure robustness of the results, we ran all our experiments ten times to report the average results.

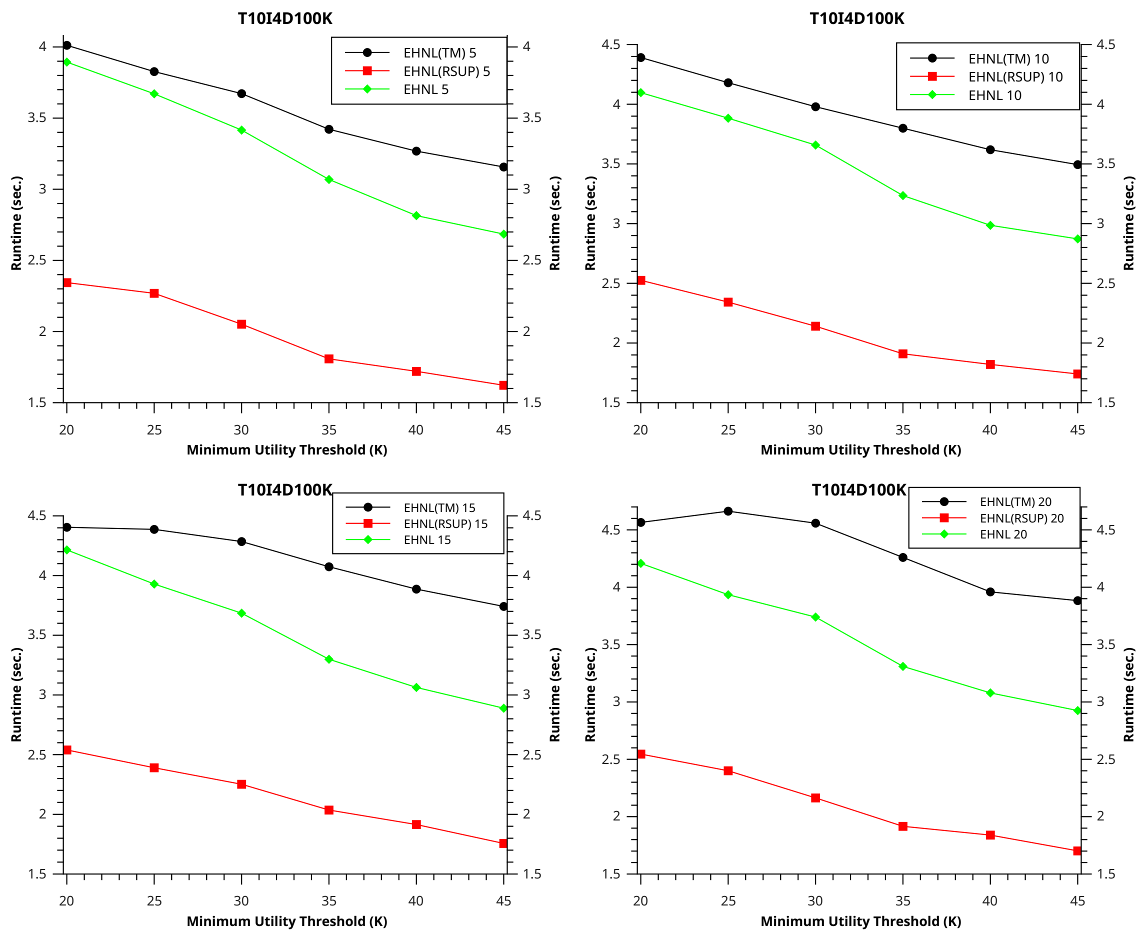

FIGURE 6.5: Runtime performance on T10I4D100K dataset

TABLE 6.9: Number of candidates on T40I10D100K dataset

| | EHNL | | | | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *min_util*$(k)$ | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 100 | 7813 | 8816 | 8915 | 8947 | 7813 | 8816 | 8915 | 8947 | 52326 | 72114 | 73430 | 73884 |
| 120 | 5305 | 5930 | 5995 | 6029 | 5305 | 5930 | 5995 | 6029 | 34608 | 45518 | 47106 | 47601 |
| 140 | 3626 | 4292 | 4366 | 4398 | 3626 | 4292 | 4366 | 4398 | 23047 | 32765 | 33979 | 34405 |
| 160 | 2596 | 3222 | 3277 | 3302 | 2596 | 3222 | 3277 | 3302 | 17302 | 25191 | 26987 | 27209 |
| 180 | 1873 | 2458 | 2549 | 2569 | 1873 | 2458 | 2549 | 2569 | 11719 | 19103 | 20617 | 21160 |
| 200 | 1364 | 1900 | 2008 | 2035 | 1364 | 1900 | 2008 | 2035 | 8293 | 14205 | 15523 | 16003 |

In order to evaluate the influence of the design techniques in EHNL, we check the performance of two versions of EHNL named EHNL(RSUP) and EHNL(TM). EHNL utilizes both the dataset reduction techniques $TM$ and pruning strategy $RSU$. EHNL(RSUP) utilizes only pruning strategy $RSU$ where $TM$ is disabled for this version. Similarly, EHNL(TM) utilizes only one dataset reduction techniques $TM$ where pruning strategy $RSU$ is disabled for this version.

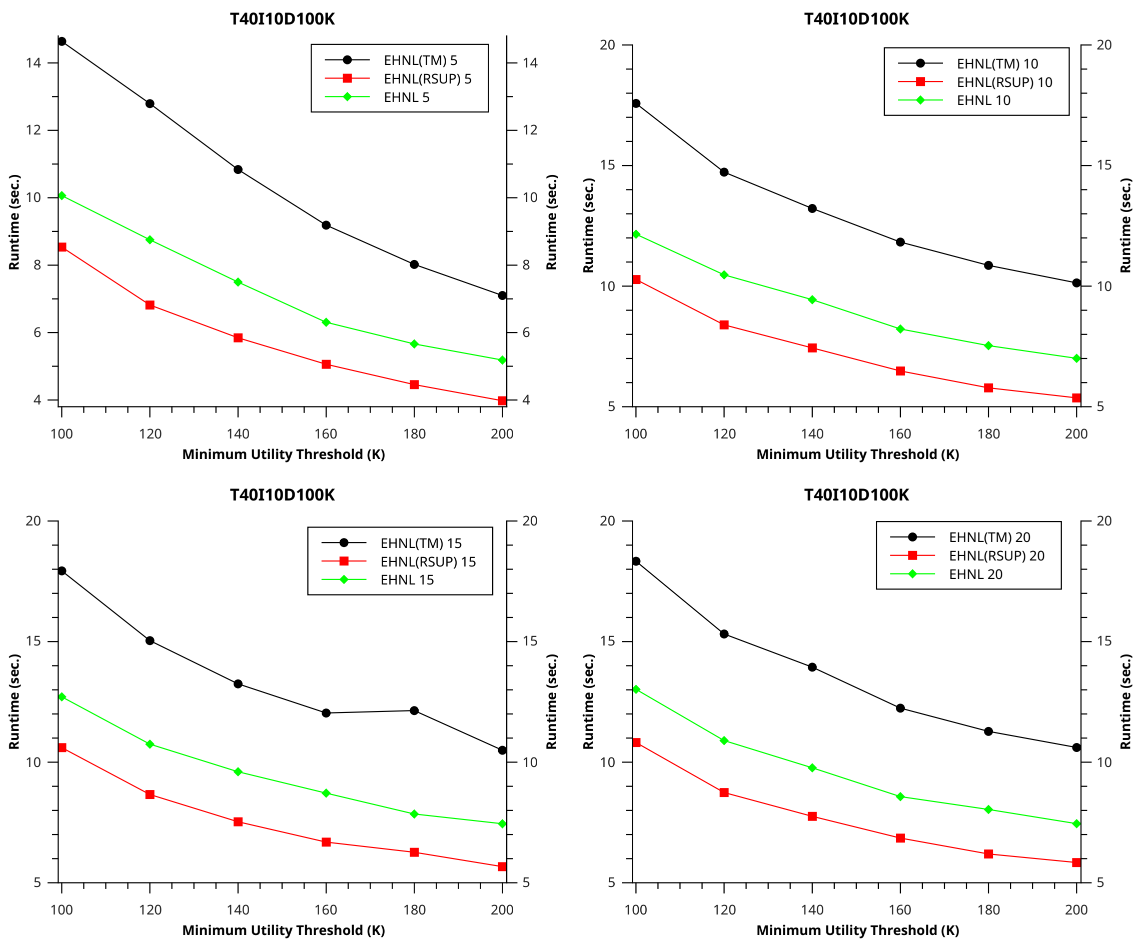

FIGURE 6.6: Runtime performance on T40I10D100K dataset

TABLE 6.10: Number of HUIs mined on accidents and chess dataset

| accidents | | | | | chess | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *min_util(k)* | **L(5)** | **L(10)** | **L(15)** | **L(20)** | *min_util(k)* | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 8000 | 2230 | 13211 | 13280 | 13280 | 60 | 20091 | 561713 | 675765 | 675765 |
| 10000 | 928 | 3929 | 3929 | 3929 | 80 | 5429 | 67976 | 70922 | 70922 |
| 12000 | 448 | 1311 | 1311 | 1311 | 100 | 1252 | 7896 | 7917 | 7917 |
| 14000 | 211 | 470 | 470 | 470 | 120 | 236 | 825 | 825 | 825 |
| 16000 | 63 | 167 | 167 | 167 | 140 | 28 | 57 | 57 | 57 |
| 18000 | 5 | 60 | 60 | 60 | 160 | 2 | 2 | 2 | 2 |

To evaluate the performance, we executed all the version on all the benchmark datasets by decreasing *min_util* threshold. For the experiment, we set the four different *max_length* threshold as 5, 10, 15 and 20, where the *min_length* is set 2. We decrease *min_util* threshold until all the versions take too much time or out of memory. The experimental results on all the datasets with all the versions of the proposed algorithm

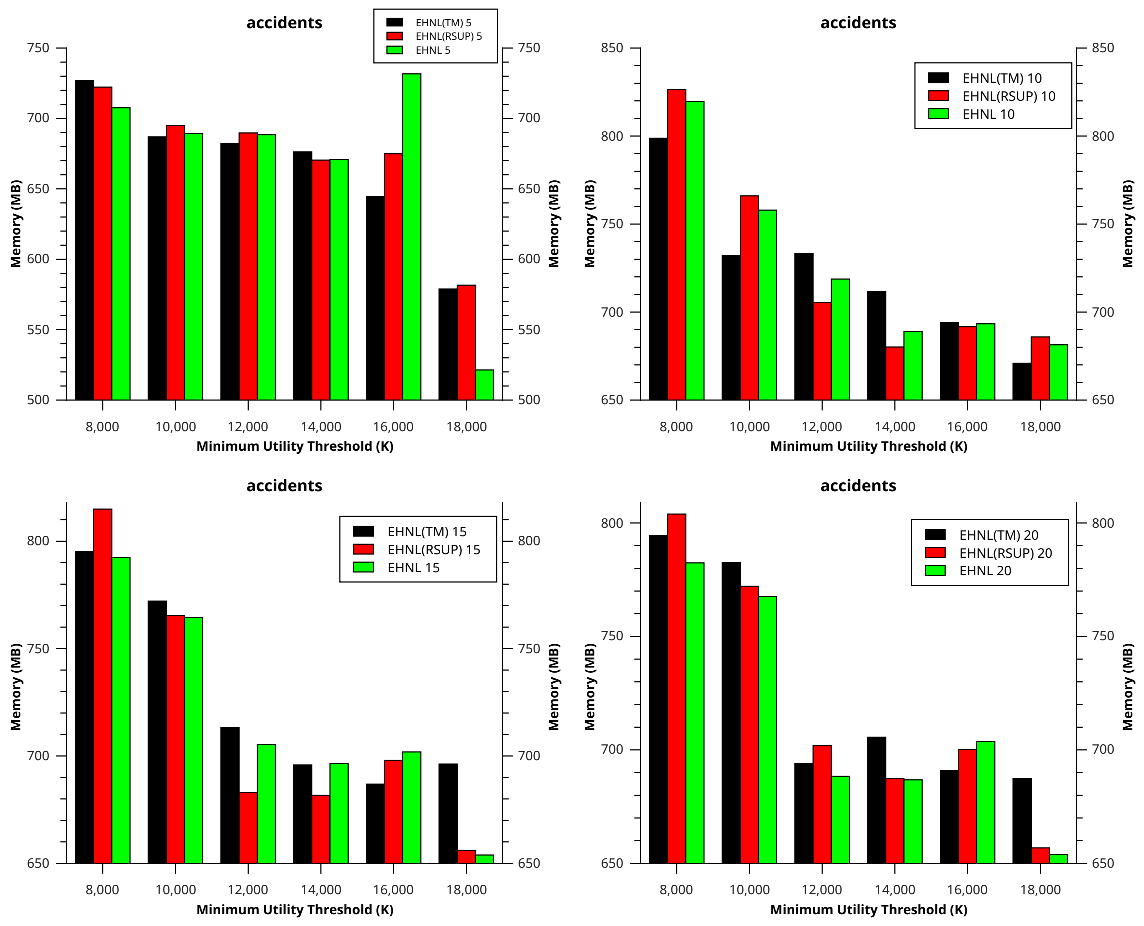

FIGURE 6.7: Memory consumption on accidents dataset

TABLE 6.11: Number of HUIs mined on mushroom dataset

| accidents | | | | |
|---|---|---|---|---|
| *min_util*($k$) | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 250 | 107 | 791 | 868 | 868 |
| 300 | 42 | 139 | 151 | 151 |
| 350 | 14 | 22 | 23 | 23 |
| 400 | 6 | 7 | 7 | 7 |
| 450 | 3 | 3 | 3 | 3 |
| 500 | 1 | 1 | 1 | 1 |

are shown in the following.

On the dataset accidents, EHNL and EHNL(TM) give similar performance for all *max_length* thresholds. When *max_length* threshold is 20, all versions of EHNL take almost same amount of runtime as shown in FIGURE 6.2. For chess and mushroom datasets, runtime of EHNL and EHNL(RSUP) almost same as shown in FIGURE 6.3

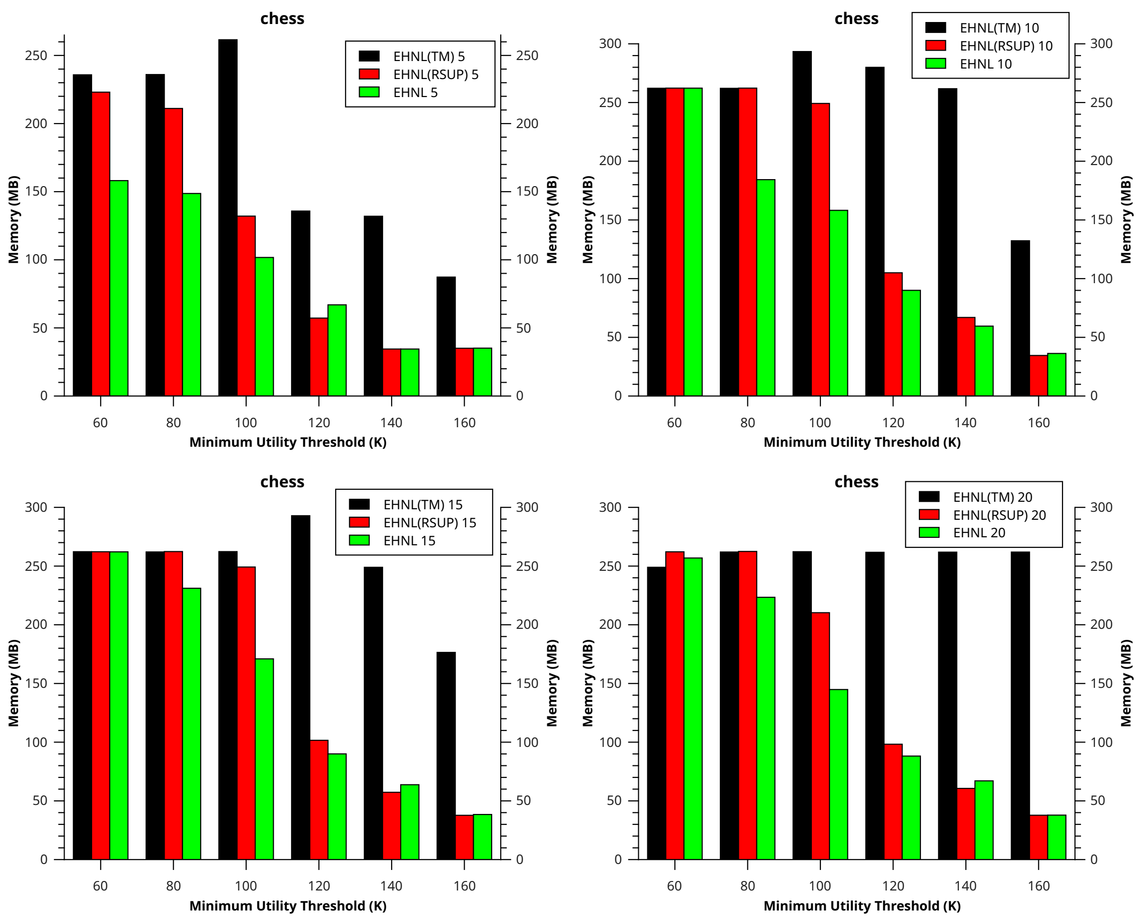

FIGURE 6.8: Memory consumption on chess dataset

TABLE 6.12: Number of HUIs mined on T10I4D100K and T40I10D100K dataset

| T10I4D100K | | | | | T40I10D100K | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *min_util*(*k*) | **L(5)** | **L(10)** | **L(15)** | **L(20)** | *min_util*(*k*) | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| 20 | 279 | 291 | 291 | 291 | 100 | 679 | 10908 | 13773 | 13775 |
| 25 | 136 | 141 | 141 | 141 | 120 | 233 | 4116 | 5125 | 5125 |
| 30 | 56 | 58 | 58 | 58 | 140 | 69 | 1161 | 1406 | 1406 |
| 35 | 27 | 28 | 28 | 28 | 160 | 24 | 196 | 219 | 219 |
| 40 | 11 | 12 | 12 | 12 | 180 | 10 | 14 | 14 | 14 |
| 45 | 3 | 4 | 4 | 4 | 200 | 6 | 6 | 6 | 6 |

and FIGURE 6.4 which means *TM* technique not give much effect on these type of dataset with length constraints.

For T10I4D100K and T40I10D100K datasets, EHNL(RSUP) outperforms EHNL and EHNL(TM) for all *max_length* thresholds because the number of distinct items in these datasets are higher. Hence, EHNL and EHNL(TM) take more time while merging and

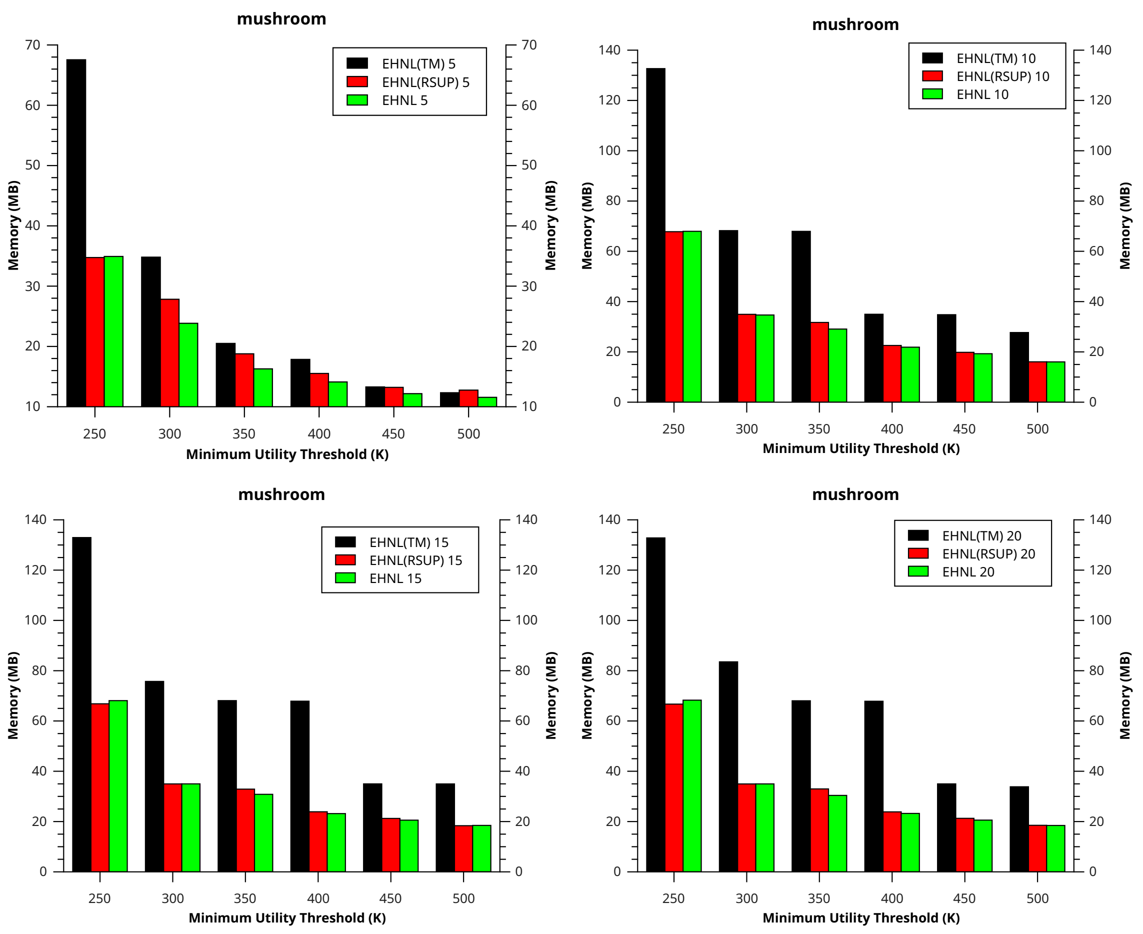

FIGURE 6.9: Memory consumption on mushroom dataset

merge less number of items. The runtime comparison of all the versions on T10I4D100K and T40I10D100K are shown in FIGURE 6.5 and FIGURE 6.6 respectively.

## 6.2.1 Effect of Techniques

The proposed algorithm uses transaction merging technique to reduce the dataset scanning cost. The proposed algorithm also uses sub-tree based pruning technique to prune the search space. In order to assess the effectiveness of these techniques, we present two versions of the proposed algorithm named EHNL(RSUP) and EHNL(TM) which utilize only sub-tree and transaction merging techniques respectively. EHNL(TM) take more runtime on almost all the dataset, because it does not utilize *RSUP* pruning strategy. EHNL(TM) uses only transaction merging and *RTWU* pruning technique.

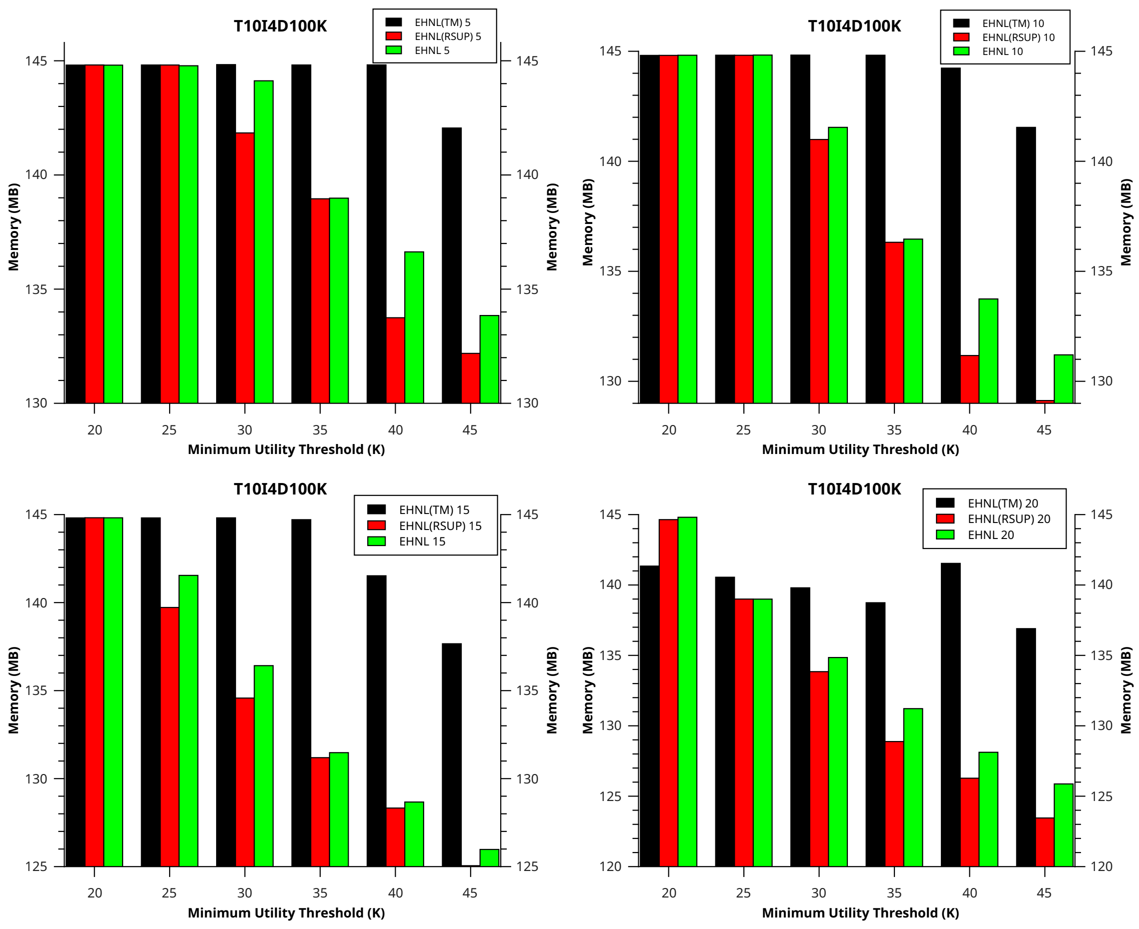

FIGURE 6.10: Memory consumption on T10I4D100K dataset

*RTWU* based technique prune only limited number of non-HUIs itemset compared to sub-tree based technique. The comparison and relationship between RSUP and *RTWU* show that *RSU* is far better than basic *TWU*. However, transaction merging technique merges lot of identical transactions. EHNL(TM) uses transaction merging technique and shows the effectiveness of this merging technique. For sparse datasets which have many distinct items, the transaction merging technique is not much effective. Hence, EHNL(RSUP) consumes less runtime for T10I4D100K and T40I10D100K datasets. Therefore, for the space and a large number of distinct items datasets, we can use EHNL(RSUP) instead of EHNL and EHNL(TM) algorithms.

The number of candidate itemsets reductions for each version on all the datasets are depicted. TABLE 6.5 shows the candidate itemsets generated on accidents dataset. TABLE 6.6 shows the candidate itemsets generated on chess dataset. TABLE 6.7 shows the candidate itemsets generated on mushroom dataset. TABLE 6.8 shows the candidate

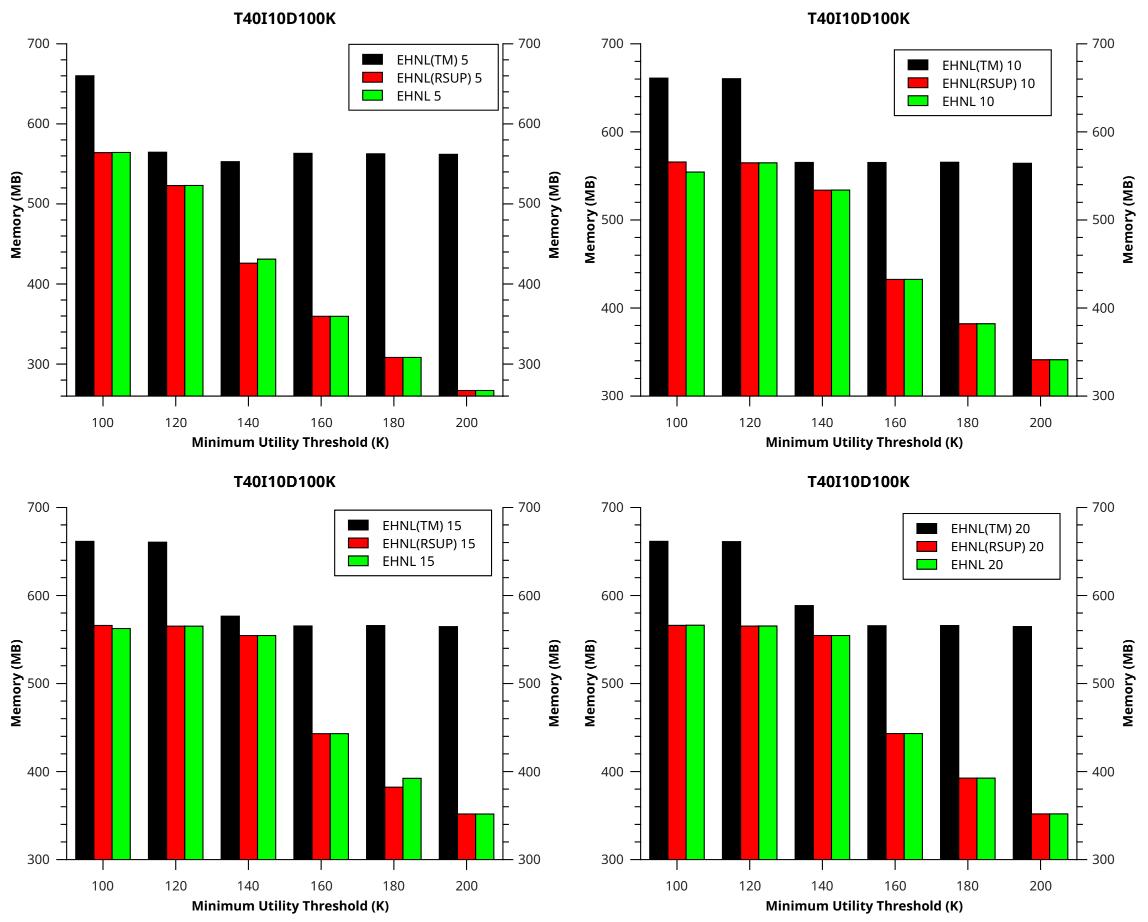

FIGURE 6.11: Memory consumption on T40I10D100K dataset

itemsets generated on T10I4D100K dataset. TABLE 6.9 shows the candidate itemsets generated on T40I10D100K dataset. The generated candidate itemsets results show as *max_length* increases the number of candidate itemsets is also increases. When *min_util* threshold increases the candidate itemsets decreases. As the candidate itemsets decrease the memory usages and runtime are also decreases. The candidate itemsets results show the effects and effectiveness of proposed techniques.

The effects of proposed techniques on the number of HUIs are shown on all the datasets. TABLE 6.10 shows the number of HUIs mined on accidents and chess dataset. TABLE 6.11 shows the number of HUIs mined on mushroom dataset. TABLE 6.12 shows the number of HUIs mined on T10I4D100K and T40I10D100K dataset. The HUIs mined results shows as *max_length* increases the number of HUIs is also increases. As well as *min_util* threshold increases HUIs decreases.

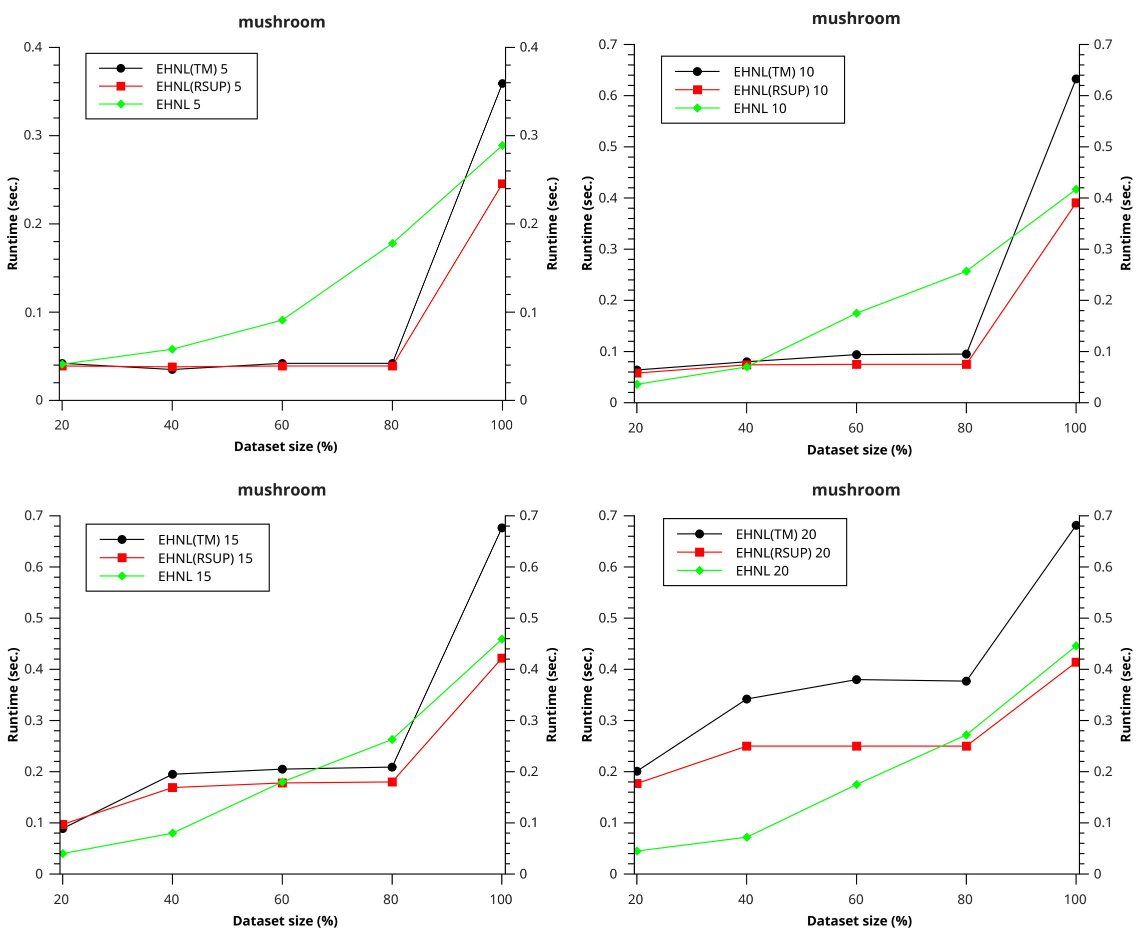

FIGURE 6.12: Runtime scalability on mushroom dataset

## 6.2.2 Memory Usage

In this section, we report the memory usage of the all the versions of the proposed algorithm on all the benchmark datasets. For the accident dataset, when *max_length* threshold increases, EHNL consumes less memory than EHNL(RSUP) and EHNL(TM). As shown in FIGURE 6.7, when *max_length* is set higher (20), EHNL consumes less memory than other two versions.

For chess and mushroom datasets, EHNL consumes less memory than other two versions for all the *max_length* and all *min_util* thresholds. It means transaction merging technique highly merges the transactions for chess and mushroom datasets.

For T10I4D100K dataset, EHNL(RSUP) always consumes less memory than EHNL and EHNL(TM). T10I4D100K is sparse dataset and has very less average length of items in a

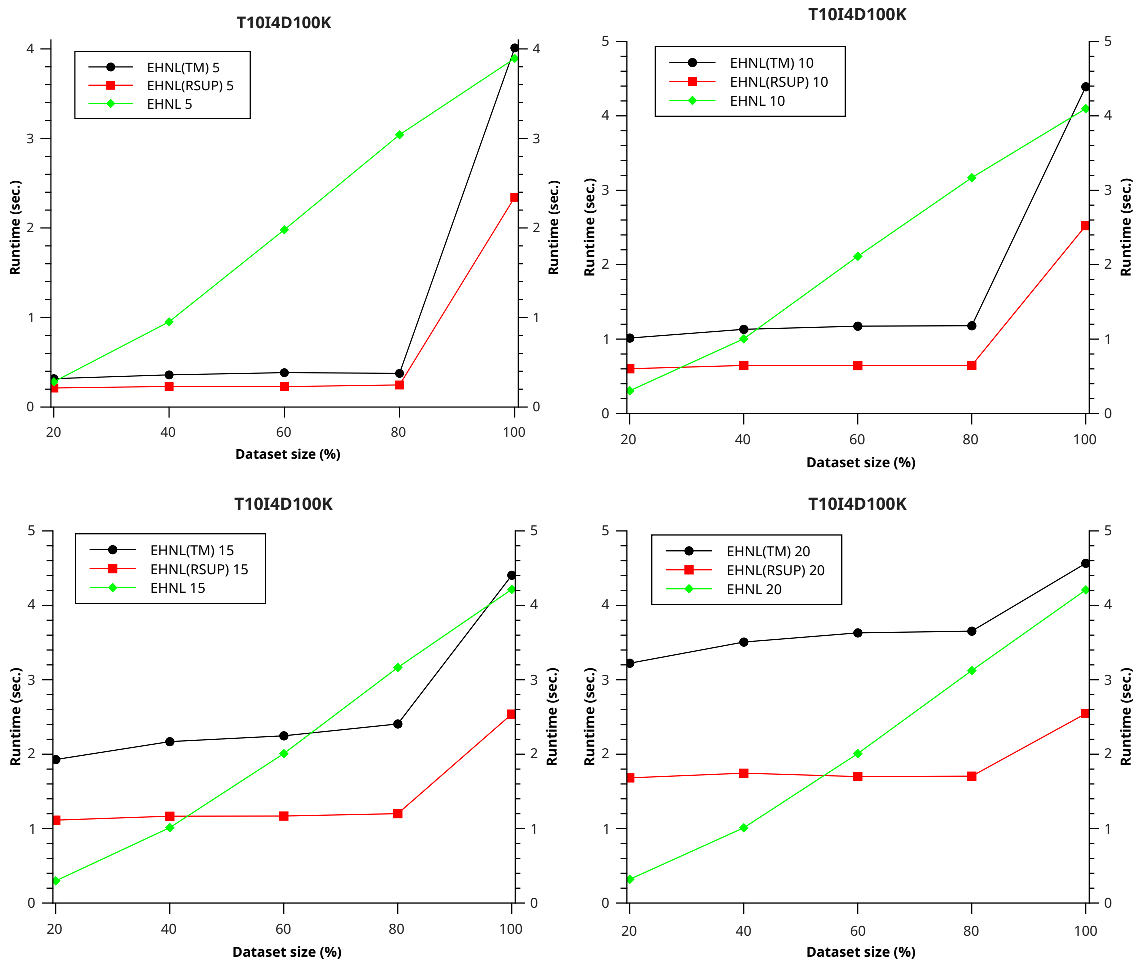

FIGURE 6.13: Runtime scalability on T10I4D100K dataset

transaction. Hence, transaction merging technique does not perform well and take much time to check identical transactions. Therefore, EHNL and EHNL(TM) consume more memory because these versions use transaction merging technique. For T40I10D100K dataset, EHNL and EHNL(RSUP) almost consume same amount of memory.

EHNL(TM) consumes high memory because it utilized transaction merging technique only with *RTWU* based pruning. *RTWU* pruning technique is not as efficient as sub-tree pruning. EHNL performs well because it takes advantage of both the transaction merging and sup-tree pruning techniques. EHNL(RSUP) seldom outperforms where the number of distinct items are high and datasets are spaces. This shows that mining without transaction merging technique or EHNL(RSUP) is good for the datasets which have a large number of distinct items or sparse datasets.

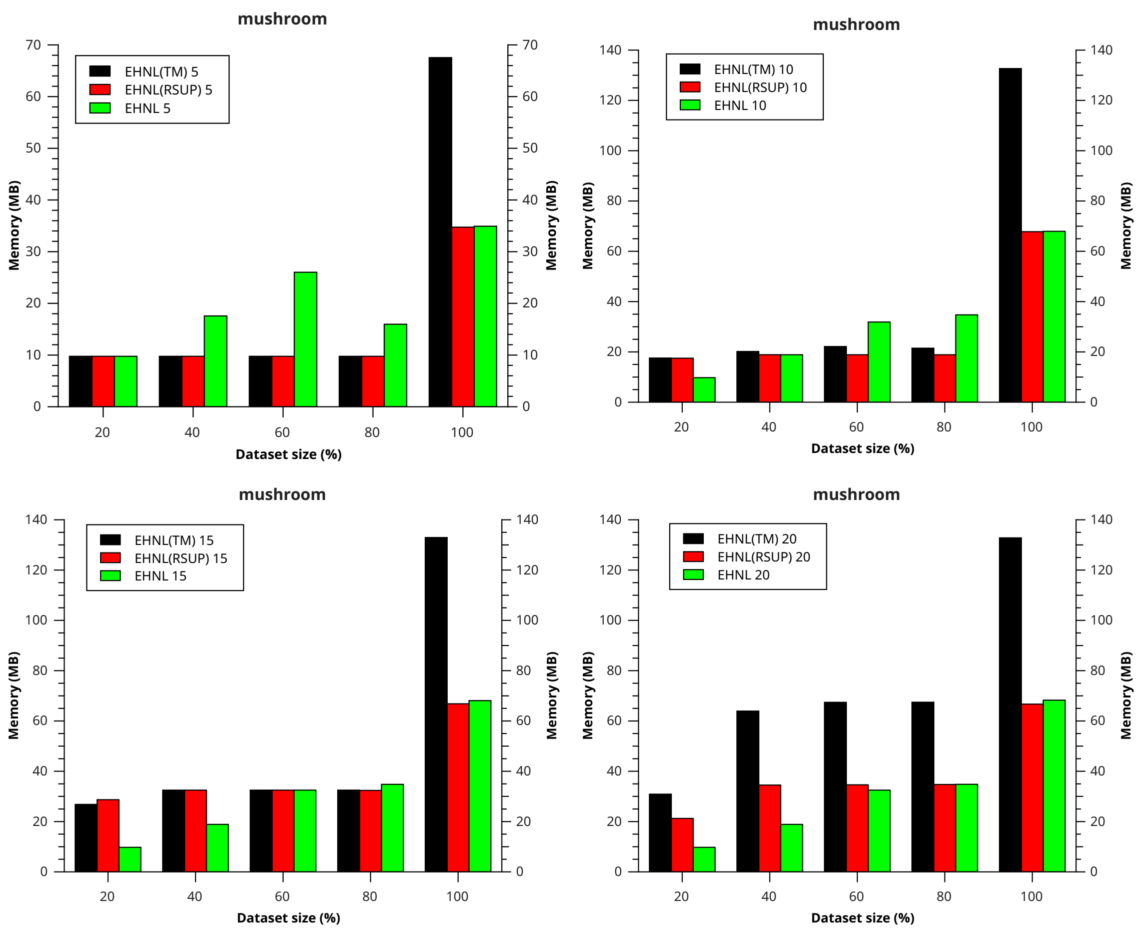

FIGURE 6.14: Memory scalability on mushroom dataset

### 6.2.3 Scalability Experiments

In this section, we test the scalability of the all the versions of the proposed algorithm. We fix *min_util* threshold to the lowest minimum threshold that is used in each dataset for runtime and memory evolution. The length constraints are set same as set for runtime and memory evolution. In order to evaluate the scalability of proposed algorithm, the size of datasets is varied from 20% to 100%. To check the scalability, we choose one real (mushroom) and one synthetic (T10I4D100K) dataset. FIGURE 6.12 shows the runtime of the algorithms on mushroom. FIGURE 6.13 shows the runtime of the algorithms on T10I4D100K. We also observe that the memory usage of EHNL increases linearly when the number of transactions increases as shown in FIGURE 6.14 and FIGURE 6.15 for the mushroom and T10I4D100K dataset respectively. As shown in FIGURE 6.12 and

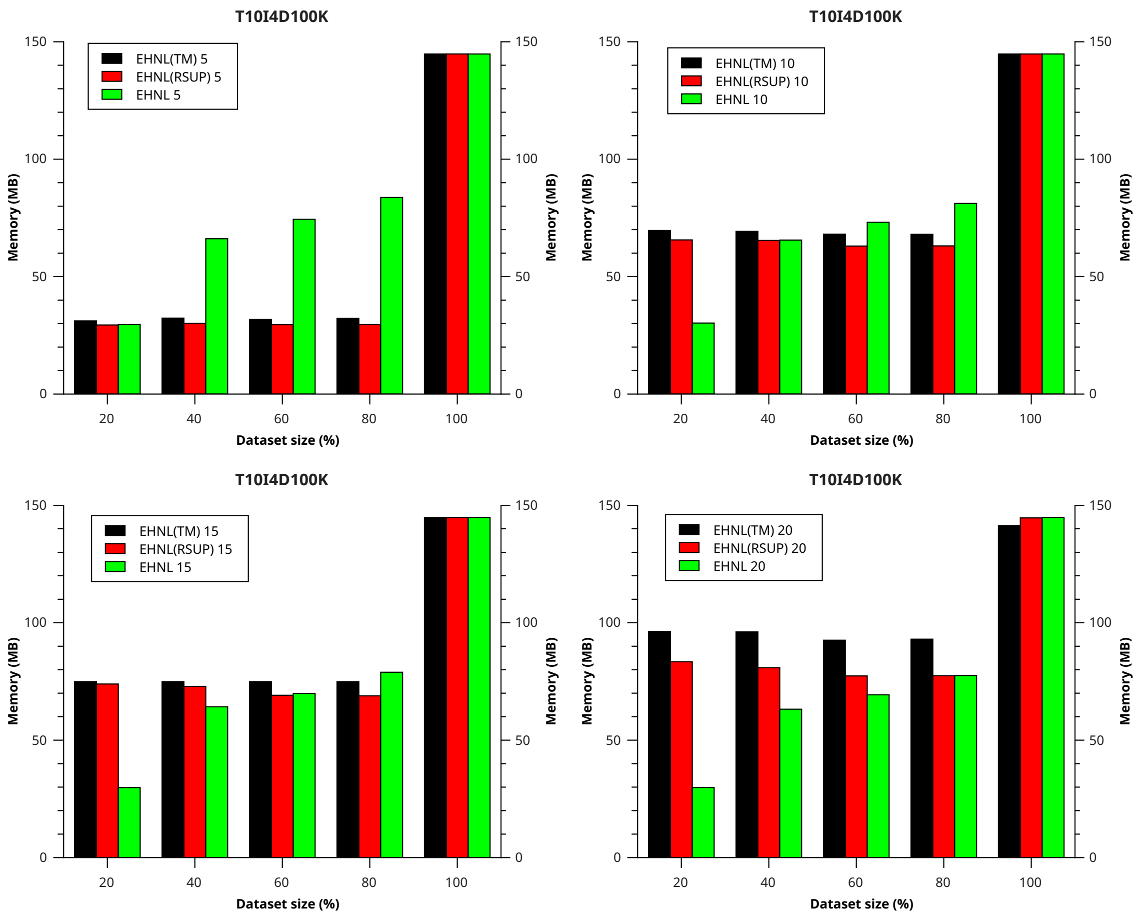

FIGURE 6.15: Memory scalability on T10I4D100K dataset

FIGURE 6.13, the proposed algorithms have good scalability under different length constraint.

TABLE 6.13: Runtime improvements of EHNL over EHNL(RSUP) and EHNL(TM)

| | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|
| **Dataset** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| accidents | 0.770 | 0.715 | 0.634 | 0.641 | 1.034 | 1.033 | 1.093 | 1.103 |
| chess | 1.038 | 1.034 | 1.024 | 1.028 | 1.190 | 1.083 | 1.056 | 1.057 |
| mushroom | 0.850 | 0.937 | 0.919 | 0.928 | 1.243 | 1.518 | 1.474 | 1.528 |
| T10I4D100K | 0.602 | 0.616 | 0.602 | 0.605 | 1.030 | 1.072 | 1.045 | 1.085 |
| T40I10D100K | 0.848 | 0.845 | 0.834 | 0.831 | 1.455 | 1.446 | 1.411 | 1.408 |

TABLE 6.14: Memory improvements of EHNL over EHNL(RSUP) and EHNL(TM)

| | EHNL(RSUP) | | | | EHNL(TM) | | | |
|---|---|---|---|---|---|---|---|---|
| **Dataset** | **L(5)** | **L(10)** | **L(15)** | **L(20)** | **L(5)** | **L(10)** | **L(15)** | **L(20)** |
| accidents | 1.021 | 1.008 | 1.028 | 1.027 | 1.027 | 0.974 | 1.003 | 1.015 |
| chess | 1.411 | 1.000 | 1.000 | 1.021 | 1.491 | 1.000 | 1.001 | 0.969 |
| mushroom | 0.995 | 0.998 | 0.982 | 0.977 | 1.934 | 1.952 | 1.952 | 1.945 |
| T10I4D100K | 1.000 | 1.000 | 1.000 | 0.999 | 1.000 | 1.000 | 1.000 | 0.976 |
| T40I10D100K | 1.000 | 1.021 | 1.006 | 1.000 | 1.170 | 1.192 | 1.176 | 1.168 |

### 6.2.4 Discussion

This chapter presents a new negative utility based HUIs mining algorithm. This chapter also presents a sub-tree based strategy to prune the search space. The presented pruning strategy calculates the utility of itemsets using array-based technique. The proposed algorithm uses length based constraints to discard very small itemsets. In order to overcome the dataset scanning cost, this chapter utilizes dataset projection and transaction merging techniques. The presented ideas are evaluated on five benchmark datasets. The presented results are quite useful and more actionable.

The runtime improvement performance of EHNL over EHNL(RSUP) and EHNL(TM) at the lowest *min_util* are shown in TABLE 6.13. The results reveal that EHNL is not runtime faster than EHNL(RSUP) on accidents, T10I4D100K and T40I10D100K datasets but there is modest improvement on chess and mushroom datasets. EHNL is always runtime faster than EHNL(TM). However, memory improvement performance of EHNL is always quite significant all the datasets as shown in TABLE 6.14. Moreover, EHNL consumes comparatively less memory than EHNL(RSUP) for sparse dataset also.

## 6.3 Summary

In this chapter, we addressed the problem of mining HUIs with negative utility value and length constraints. Most of the traditional HUIs mining algorithms mine the rules from the datasets that have only positive utility. But in real life negative utility is very important. In literature, only HUINIV-Mine [29] and FHN [30] algorithms are proposed to solve the negative utility itemsets mining. But the rules mined by both of these algorithms include very large number of very small itemsets. We overcome this problem by incorporating the concept of length-based constraints in negative HUIs mining. Only one algorithm named FHM+ [47] is proposed in literature with length constraints but this algorithm not mined the rules with negative utility value. Furthermore, the state-of-the-art algorithms are consumed lots of memory and running time to solve the negative utility mining. In order to achieve the efficiency, we utilize sub-tree pruning strategy to reduce the search space. But the main problems we faced are how to incorporate the sub-tree pruning strategy and length constraints with negative utility itemsets mining. In order to solve these problems, we sort the items in the transactions non-increasing order according to their utility values and keep the negative items at the end of sorted items with the help of offset pointers. Furthermore, in order to achieve efficiency, we utilize dataset projection and transaction merging techniques as preprocessing techniques to reduce the dataset scanning cost. Another problem we faced is how to calculate the utility of itemsets efficiently. To solve this problem, we use array-based utility counting technique to calculate utility value in linear time and negligible memory space. We explain the working of the proposed algorithm with detail example. The experimental results show that the proposed algorithms mine HUIs from the negative utility values with length constraints. The experimental results show that the proposed algorithms mine HUIs efficiently. Moreover, in order to show the effect of sub-tree based pruning strategy and transaction mering technique, we propose two versions of the proposed algorithm. The experimental results show how much the strategies used to achieve the efficiency are effective with real life or benchmark datasets. Moreover, the proposed algorithm mines HUIs efficiently for real datasets with low memory consumption.