Chapter 3

High Utility Itemsets Mining Considering Length Constraints

Traditional HUIs mining algorithms discover large amount of HUIs with various very small itemsets. A large number of HUIs are not good because it consumes large memory and not actionable. Very few items in HUIs are also less useful to take decisions. Traditional algorithms take a lot of time to generate candidate itemsets and find HUIs. Hence concise HUIs mining plays an important role. Therefore length constraints based approach to discover HUIs is a good way to reduce search space and remove irrelevant items (very small itemsets) and discover more relevant HUIs. The problem of mining HUIs with length constraints is to find all itemsets having a utility no less than min_util and containing at least min_length items and at most max_length items. To enforce the length constraints, it is needed to check min_length and max_length threshold as early as possible. To reduce the search space, we introduce dataset projection and transaction merging. The proposed algorithm essentially consists a set of techniques for reducing upper-bounds with length constraints. These length constraints introduce tighter and revised upper-bounds such as revised TWU ($RTWU$) to prune the search space. We take the example transactional dataset from TABLE 2.1 and external utility from TABLE 2.2. The revised utilities are as follows.

Definition 3.0.1. (Revised transaction utility). The revised transaction utility denoted by $RTU(T_j)$ for transaction T_j 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.

TABLE 3.1: Revised TU values in the example

T_{id}	TU	RTU
T_1	17	14
T_2	9	9
T_3	21	20
T_4	9	9
T_5	4	4
T_6	20	16
T_7	13	13

For example, RTU of T_1 can be as $RTU(T_1)$= $U(A,T_1)$ + $U(B,T_1)$ + $U(D,T_1)$ = 4 + 6 + 4 = 14. $RTU(X) \leq TU(X)$, because TU is the summation of all items in a transaction without any deletion. TABLE 3.1 shows TU values of the running dataset. RTU is the summation of remaining items after deletion by apply min_length and max_length constraints. TABLE 3.1 shows RTU value of each transaction.

TABLE 3.2: TWU values of items

Item	A	B	C	D	E
TWU	41	80	72	67	89

TABLE 3.3: $RTWU$ values of items

Item	A	B	C	D	E
RTWU	34	72	67	59	81

Definition 3.0.2. (Revised transaction weighted utility). The revised 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, the RTU of transactions T_1, T_2, T_3, T_4, T_5, T_6 and T_7 is 14, 9, 20, 9, 4, 16 and 13 respectively. Hence, $RTWU(A)$= $RTU(T_1)$ + $RTU(T_5)$ + $RTU(T_6)$ = 14 + 4 + 16 = 34, which is a tighter upper-bound on the utility of $\{A\}$ and its supersets than the original TWU which is calculated as 41. TABLE 3.2 shows the TWU and TABLE 3.3 shows the $RTWU$ value of each item. The proposed $RTWU$ has the two following important properties.

Property 3.0.1. The revised TWU is a more tighter upper-bound than the original TWU. For an itemset X, The relationship between these upper-bounds is $RTWU(X) \leq TWU(X)$.

Proof: For the itemset X, we can observe that $RTU(X) \leq TU(X)$. Hence, $RTWU(X) \leq TWU(X)$ where $RTWU(X)$ = $\sum_{X \subseteq T_j \in D} RTU(T_j)$ and $TWU(X)$ =$\sum_{X \subseteq T_j \in D} TU(T_j)$.

Property 3.0.2. (Pruning using $RTWU$). For the itemset X, if $RTWU(X) < min\_util$, then X and its supersets are not HUIs.

Proof: The $RTU(T_j)$ shows the utility for a transaction T_j and an itemset observing the length constraints (min_length and max_length). Hence, $RTWU(X)$ can be considered as an upper-bound. It is also observed that the utility of supersets for itemset X cannot appear in more transactions than itemset X.

All the two-phase algorithms (Two-Phase, BAHUI, UP-Growth and UP-Growth+) in the literature use the ordinary TWU utility to prune the search space. UP-Growth+ is one of the fastest among these two-phase algorithms. In recent years, one-phase algorithms such as d2HUP and HUI-Miner were proposed to avoid a huge number of candidate itemsets generation. These algorithms are faster than UP-Growth+. The improved versions of HUI-Miner algorithm (FHM, HUP-Miner and EFIM) are proposed to reduce the execution time and also reduce the number of join operation. One-phase algorithms used the concept of remaining utility and utility-list. The proposed algorithm is a variation of the EFIM algorithm. Now we present the revised utility-list (RUL) which is proposed in FHM+ algorithm [47]. We presents RUL to compare with proposed revised sub-tree and local utility upper bounds.

Definition 3.0.3. (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 extend, i.e. $V(T_j,X) = \{v \in T_j | v \succ x, \forall x \in X\}$. The max_length constraint sets 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 $maxExtend(X) = max\_length - \mid X \mid$, where $\mid X \mid$ defines the number of items in X. The largest utility value with X for transaction T_j is denoted as $L(T_j,X)$.

Definition 3.0.4. (Revised remaining utility). For the itemset X and transaction T_j the RRU is defined as $RRU(X,T_j) = \sum L(T_j,X)$. The RRU of the X in the dataset is defined as $RRU(X) = \sum_{X \subseteq T_j \in D} RRU(X,T_j)$.

In the running example, the RRU of itemset $\{A\}$ is 22, while the RU of $\{A\}$ is 29. Thus the RRU can be a much tighter upper-bound than the original RU upper-bound. The RRU has the two following important properties.

Property 3.0.3. (The *RRU* is a tighter upper-bound than the *RU*). For the itemset X, the relationship between *RRU* and the original *RU* is $RRU(X) \leq RU(X)$. This shows that *RRU* upper-bound is much tighter [47].

Proof: For an itemset X, transaction T_j and $RRU(X) \leq RU(X)$ shows that $RRU(X)$ $=\sum_{X \subseteq T_j \in D} RRU(X, T_j) \leq RU(X) = \sum_{X \subseteq T_j \in D} RU(X, T_j)$.

Property 3.0.4. (Pruning using *RRU*). For the itemset X, if the sum of $U(X) + RRU(X) <$ min_util then itemset X and its supersets are not HUIs observing with length constraints.

Proof: Since $U(X)$ denotes the utility of X and $RREU(X)$ denotes the highest utilities of items that could be appended to X respect to the max_length constraint.

Definition 3.0.5. (Revised utility-list). For the itemset X in a dataset D, $RUL(X)$ is a set of tuples. The tuple consists of the fields (tid, $iutil$, $llist$) for itemset X in each transaction T_j. The utility-lists used in the previous utility-based algorithms is different from revised utility-list which is proposed in FHM+. The RUL replaces $rutil$ with $llist$ element, $llist$ stores the set $L(T_j, X)$.

Let us consider the running example with $max\_length = 3$. The RUL of $\{A\}$ is $\{(T_1, 4,$ $\{6,4\}), (T_5, 4, \{0\}), (T_6, 4, \{8,4\})\}$. The proposed RUL stores the necessary information for pruning an itemset X and its extensions using *Property 3.0.4*.

Property 3.0.5. (Pruning search space using the revised utility-list). For the itemset X, if the summation value of the $iutil$ and $llist$ in $RUL(X)$ is less than min_util then itemset X and its extensions are not HUIs concerning the length constraints.

RUL based pruning property is very useful as used in FHM+ algorithm. But we cannot utilize this property because proposed algorithm is horizontal (tree-based) algorithm. The proposed algorithm uses tree based pruning strategies (instead of utility-list based strategy) as proposed in Section 3.1.3.

3.1 EHIL Algorithm

The proposed algorithm introduces several new ideas to reduce the execution time and memory requirement for discovering HUIs. We utilize array-based utility counting

technique to calculate upper-bounds efficiently. The proposed algorithm uses a strict designed constraint for each itemset in the search space.

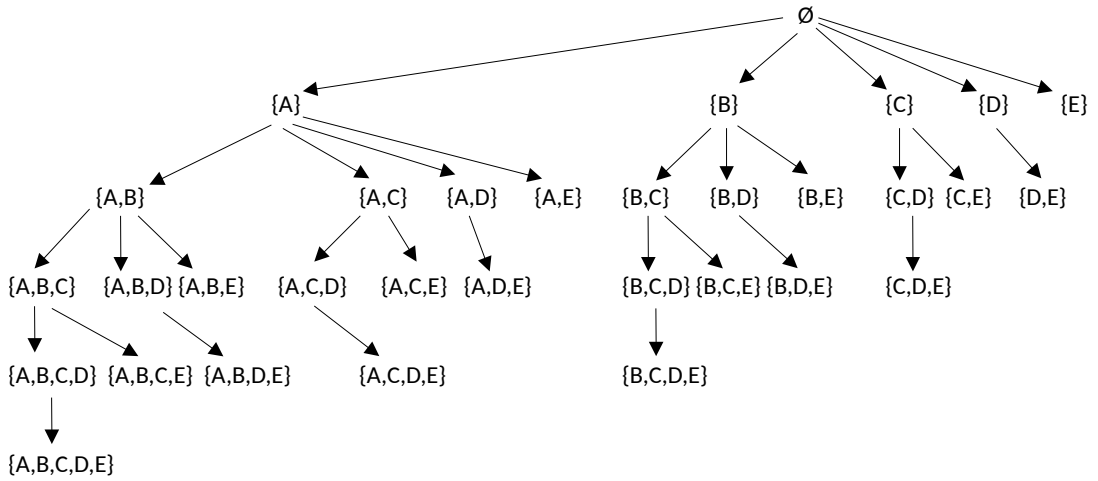

FIGURE 3.1: Set-enumeration tree for $I = \{A, B, C, D, E\}$

3.1.1 The Search Space

The search space of HUIs mining problem has been represented as a set-enumeration tree. The representation of itemsets has been shown in set-enumeration tree in FIGURE 3.1 and this representation would become search space for proposed algorithm. The technique used to explore the search space is same as used in UP-Growth [12]. The difference is that during the depth-first search, the proposed algorithm recursively appends one item at a time to itemset α by following $\succ$ order for an itemset α while generating larger itemsets as shown in FIGURE 3.1. During the implementation, ordering of itemsets is done according to increasing $RTWU$ as this reduces the search space for HUIs.

Definition 3.1.1. (Extension of an item). Let the α be itemset and $E(\alpha)$ is a set of items that is used to extend the itemset α and is defined as $E(\alpha) = \{z \mid z \in I \wedge z \succ x, \forall x \in \alpha\}$.

Definition 3.1.2. (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 where $Z = \alpha \cup W$ for an item $W \in 2^{E(\alpha)}$.

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 $\{C, D, E\}$.

3.1.2 Efficient Dataset Scanning Techniques

The proposed algorithm reduces the dataset scanning costs by reducing the dataset size using dataset projection and transaction merging techniques.

3.1.2.1 Dataset Scanning using Projection

Dataset projection techniques are employed to reduce the memory requirement and speed up the execution in the mining process. Dataset Projection technique relies on the observation when an itemset α is taken into account and all items not belonging to $E(\alpha)$ can be ignored when dataset is scanned to calculate the utility of itemsets within the sub-tree of α. Such dataset where items do not belong to extended set is called a projected dataset.

Definition 3.1.3. (Projected dataset). The projection of a transaction T_j using an itemset α 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 α is denoted as $\alpha - D$ and is defined as the multi-set $\alpha - D = \{\alpha - T \mid T \in D \wedge \alpha - T \neq \emptyset\}$.

In the running example, let us consider dataset D and $\alpha = \{c\}$. The projected dataset $\alpha - D$ contains transactions $\alpha - T_2 = \{B, E\}$, $\alpha - T_3 = \{B, E\}$, $\alpha - T_4 = \{E\}$, $\alpha - T_6 = \{B, E\}$ and $\alpha - T_7 = \{B, E\}$. The projection technique highly reduces the dataset scanning cost and therefore, larger transactions become smaller as seen in the example. In order to implement projection technique, we sort each original transaction to the $\succ$ total order. Each projected transaction is represented by an offset pointer on the corresponding original transaction. The major issue is how to efficiently implement dataset projection technique. To solve this problem, we follow the dataset projection technique proposed in EFIM algorithm [28].

3.1.2.2 Dataset Scanning using Merging

To reduce the dataset scanning, we utilize a transaction merging technique. This is a relay on the observation that the dataset contains the identical transactions. Identical transactions contain the exactly same items but may not have same quantity values

(internal utility). The merging technique identifies these identical transactions and replaces them with a single transaction.

TABLE 3.4: Transaction Merging

T_{id}	Transaction	Purchase quantity (IU)	Utility (U)	RTU
T_1	A, D, B, E	2, 1, 2, 3	4, 4, 6, 3	14
$T_{M\_2\_7}$	**C, B, E**	**7, 4, 3**	**7, 12, 3**	**22**
T_3	D, C, B, E,	3, 1, 2, 2	12, 1, 6, 2	20
T_4	D, C, E	1, 2, 3	4, 2, 3	9
T_5	A	2	4	4
T_6	A, D, C, B, E	2, 2, 4, 1, 1	4, 8, 4, 3, 1	16

Definition 3.1.4. (Transaction merging). Let the identical transactions Ta_1, Ta_2, Ta_3, Ta_n are replaced by a new transaction T_M= Ta_1= Ta_2 = Ta_3 = Ta_n (Identical transactions may not contain the same quantity values of each item). And quantity of these identical transactions is $k \in T_M$ and is defined as $IU(k,T_M) = \sum_{i=1,\ldots,n} IU(k,Ta_i)$. 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 transaction could be more likely to be identical.

Definition 3.1.5. (Projected transaction merging). Let the identical transactions 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 $k \in T_M$ is defined as $IU(k,T_M) = \sum_{i=1,\ldots,n} IU(k,Ta_i)$.

For example, the identical transactions $\alpha - T_2$ and $\alpha - T_7$ can be replaced by a new transaction $T_{M\_2\_7} = \{C,B,E\}$ where $IU(C,T_M) = 7$, $IU(B,T_M) = 4$ and $IU(E,T_M) = 3$. TABLE 3.4 shows the merging of identical transactions. The projected dataset $\alpha - D$ for $\alpha = \{C\}$ contains the identical projected transactions as $\alpha - T_{M\_2\_7} = \{C,B,E\}$, $\alpha - T_3 = \{C,B,E\}$ and $\alpha - T_6 = \{C,B,E\}$ is further merged by a new projected transaction using offset pointers in the original dataset.

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. In order to overcome this problem, we need to compare all transactions with each other. But this

technique to compare all the transactions with one another is not an efficient technique. To overcome this problem following approach is presented here.

Definition 3.1.6. (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 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. Accordingly, we can find the identical transactions in the projected dataset as shown in TABLE 3.4. 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.

The projected dataset follows the above property to get all the identical transactions by comparing each transaction with the next transaction. Therefore, the transaction merging technique can be done easily by scanning the dataset only once. All One-phase HUIs mining algorithms do not perform transaction merging technique except for the EFIM. The utility-list based algorithms like FHM, HUP-Miner and hyper-link based algorithms, d2HUP do not perform transaction merging due to their vertical dataset representation.

3.1.3 Pruning Strategies

We have so far introduced one novel tighter upper-bound, $RTWU$ with length constraints for reducing the search space. Now we introduce three new pruning strategies to prune the search space. These strategies are more efficient and much tighter upper-bound on the utility of itemsets.

3.1.3.1 Prune search space using Revised Local Utility

Definition 3.1.7. (Revised Local Utility). For an itemset α and an item $Z \in E(\alpha)$, RLU value of α is $RLU(\alpha, z) = \sum_{T \in (\alpha \cup Z)})[U(\alpha, T) + RRE(\alpha, T)]$.

In the running example $\alpha = \{B\}$. We have that $RLU(\alpha, C)$ = (9 + 20 + 16 + 13) = 58, $RLU(\alpha, D) =$ (14 + 20 + 16) = 50.

Definition 3.1.8. (*RLU* overestimation). For an itemset α and an item $Z \in E(\alpha)$, let ZE be an extension of α such that $Z \in ZE$. Therefore, $RLU(\alpha, Z) \geq U(ZE)$.

Theorem 3.1. *(Pruning using RLU). For the itemset α and an item $Z \in E(\alpha)$, if $RLU(\alpha, Z) < min\_util$ then the single item Z and all extensions of α containing item Z are low-utility in a sub-tree. Furthermore, item Z is ignored for exploring all sub-trees of α.*

3.1.3.2 Prune search space using Revised Sub-tree Utility

Definition 3.1.9. (Revised Sub-tree Utility). Let us consider an itemset α and an item $Z \in E(\alpha)$ that can extend α according to the depth-first search $(Z \in E(\alpha))$. The *RSU* of the item Z w.r.t.[1] α is $RSU(\alpha, Z) = \sum_{T \in (\alpha \cup \{Z\})} [U(\alpha, T) + U(Z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{Z\})} U(i, T)]$.

Moreover in the running example, $\alpha = \{C\}$. We have that $RSU(\alpha, B)$ = (7 + 12 + 3) + (1 + 6 + 12) + (4 + 3 + 9) = 57.

Property 3.1.1. (*RSU* Overestimation). For an itemset α and an item $Z \in E(\alpha)$, the utility value of $RSU(\alpha, Z) \geq U(\alpha \cup \{Z\})$ and accordingly, $RSU(\alpha, Z) \geq U(Z)$ which keeps the extension Z of $\alpha \cup \{Z\}$.

Theorem 3.2. *(Pruning using RSU). For an itemset α and an item $Z \in E(\alpha)$, if $RSU(\alpha, Z) < min\_util$ then the single item extension $\alpha \cup \{Z\}$ and its extensions are low-utility in sub-tree. Furthermore, the sub-tree of $\alpha \cup \{Z\}$ is pruned in the set-enumeration tree. Using this Theorem, some sub-tree can be pruned of an itemset α. Therefore, the number of sub-trees is reduced. And hence the search space is reduced.*

The relationships between the revised upper-bounds (*RTWU*, *RREU*, *RLU* and *RSU*) and the state-of-the-art upper bounds (*RTWU*, *RREU*) [47] are following.

Property 3.1.2. For an itemset α, an item z and an itemset $Y = \alpha \cup \{Z\}$, the relationship $TWU \geq RTWU(Y) \geq RLU(\alpha, z) \geq RREU(Y) \geq REU(Y) = RSU(\alpha, z)$ holds.

[1] with respect to

According to the above relationship, it is seen that the revised upper-bounds are more tighter upper-bound than the state-of-the-art upper-bounds on the utility of itemset Y. Length constraints upper-bound $RTWU$ is tighter than original TWU as shown by the *Property 3.0.1*. $RTWU$ is equal to RLU, the difference is that RLU is calculated by performing dept-fist search the tree. Therefore, RLU can be more tighter and effective for pruning the search space. The relationship and tightness of $RREU$ and REU are also shown by the *Property 3.1.2*. The main advantage of the proposed pruning strategies RLU and RSU is that these are calculated when depth first search for itemset α rather than for the child itemset Y. Therefore, if $RSU(\alpha, z)$ is less than min_util then the proposed algorithm prunes the whole sub-tree of item z including the itemset Y instead of pruning only the descendant nodes of Y. Hence, the RSU is more effective for pruning the search space than other Upper-bounds.

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

Definition 3.1.10. (*Primary* and *Secondary* items). For an itemset α, the *Primary* items of itemset α is the set of items, $Primary(\alpha) = \{Z \mid Z \in E(\alpha) \wedge RSU(\alpha, Z) \geq min\_util\}$. The itemset α is the set of items $Secondary(\alpha) = \{Z \mid Z \in E(\alpha) \wedge RLU(\alpha, Z) \geq min\_util\}$. The $RLU(\alpha, Z) \geq RSU(\alpha, Z)$, so $Primary(\alpha) \subseteq Secondary(\alpha)$.

Definition 3.1.11. (Length Constraints). Mining HUIs with length constraints is to find all itemsets having a utility not less than min_util and containing length at least min_length items and at most max_length items. Where min_util, min_length and max_length are parameters given by the user.

For example, if $min\_util = 30$, $min\_length = 2$ and $max\_length = 3$, the set of HUIs is: $\{\{D,C\}, \{D,C,B\}, \{D,C,E\}, \{D,B\}, \{D,B,E\}, \{D,E\}, \{C,B\}, \{C,B,E\}, \{B,E\}\}$.

3.1.3.3 Pruning using Length Constraints

The proposed technique prunes candidates which do not fulfill the length constraints. We initially remove the transaction which does not follow min_length constrains. We compare the length of candidates HUIs with max_length constraint. If the length of any candidate HUIs is equals to upper-length constraint then recursion of adding extended itemsets are

stopped. Otherwise, a new item is added to enhance the length of itemset using recursive function explained in next subsection. The proposed algorithm is inspired by a similar length-based HUIs mining algorithm FHM+[47].

3.1.4 Calculate Upper Bounds using Utility Array

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

Definition 3.1.12. (Utility Array) For the set of items I which appears in a dataset D. The UA is an array of length $|I|$ that have an entry denoted as $UA[Z]$ for each item $Z \in I$. 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 UA calculates upper-bounds as follows.

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

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

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

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

Algorithm 1: EHIL algorithm

Input: D: a transaction dataset, min_util: user-specified threshold, min_length, max_length

Output: High utility itemsets (HUIs)

```
1  α ← Ø;
2  foreach T_j ∈ D do
3      η ← | T_j |;          // no. of items in T_j
4      if η < min_length then
5          Remove T_j from D;
6      else if η > max_length then
7          sort utilities of T_j in decreasing order;
8          β ← 0;            // to store RTU
9          count ← 0
10         for count ≤ max_length do
11             β ← β + U(count);
12         U(T_j) ← β;       // update RTU(T_j) with β
13 Scan D, compute RLU(α,Z) for all items using UA[Z].
14 Secondary(α) = {Z | Z ∈ E(α) ∧ RLU(α,Z) ≥ min_util}.
15 Let ≻ be the total order of RTWU increasing values on Secondary(α)
16 Scan D, remove item Z ∉ Secondary(α) from the transactions T_j and delete empty
     transactions T_j;
17 Sort all the remaining transactions in D according to ≻_T;
18 Scan D, compute the RSU(α,Z) of each item Z ∈ Secondary(α), using UA[Z];
19 Primary(α) = {Z | Z ∈ Secondary(α) ∧ RSU(α,Z) ≥ min_util};
20 search(max_length, α, α − D, Primary(α), Secondary(α), min_util);
21 return HUIs;
```

The proposed algorithm for calculating upper-bounds is highly efficient. It can calculate all the upper-bounds by performing only one scan of the projected dataset. We also observe that *UA* is very compact and efficient data structure. In order to use *UAs* efficiently, we introduce three optimizations. 1) All items in the dataset D are used as consecutive integer numbers. After that the $UA[k]$ stores an item k on the k^{th} position in the array. This optimization allows to access each item of the *UA* in $O(1)$ time. 2) Before each *UA* uses a reinitializing technique to reuse the same created arrays; hence memory requirement is greatly reduced. In our work, only three *UAs* are used to calculate *RTWU*, *RSU*, *RLU* and length of each itemset. Hence the proposed algorithm consumes very low memory compared to the state-of-the-art HUIs mining algorithms. 3) *UA* re-initializes for calculating the *RSU* or *RLU* of single-item extensions of an itemset α.

Only related *UA* re-initializes corresponding to extended items α.

3.1.5 The Proposed Algorithm

Algorithm 2: The *Search* procedure

Input: *max_length* : a maximum length threshold specified by user, α : an itemset, $\alpha - D$: a projected dataset, $Primary(\alpha)$: the primary items of α, $Secondary(\alpha)$: the secondary items of α, *min_util*

Output: HUIs that are extension of α

1 $\eta \leftarrow \mid \alpha \mid$ *// no. of items in* α
2 **if** $\eta \leq max\_length$ **then**
3 **foreach** *item* $Z \in Primary(\alpha)$ **do**
4 $\beta \leftarrow \alpha \cup \{Z\}$
5 Scan $\alpha - D$, compute $U(\beta)$ and create $\beta - D$; *// using transaction merging*
6 **if** $U(\beta) \geq min\_util$ **then**
7 Output β
8 Scan $\beta - D$, Compute $RSU(\beta, Z)$ and $RLU(\beta, Z)$ where item $Z \in Secondary(\alpha)$, using two *UAs*
9 $Primary(\beta) = \{Z \in Secondary(\alpha) \mid RSU(\beta, Z) \geq min\_util\}$;
10 $Secondary(\beta) = \{Z \in Secondary(\alpha) \mid RLU(\beta, Z) \geq min\_util\}$;
11 $search(max\_length, \beta, \beta - D, Primary(\beta), Secondary(\beta), min\_util)$;

In this section, we propose an efficient algorithm EHIL for discovering HUIs. EHIL is an extension of EFIM [28]. EHIL includes several strategies to discover HUIs with length constraints. We utilize array-based utility counting technique to efficiently calculate the upper-bounds.

The **Algorithm 1** takes a transactional dataset and three threshold *min_util*, *min_length*, *max_length* as input. Line 1 initially considers the empty itemset α. The **foreach** loop in lines 2-12 scan each transaction and comply length constraints on each transaction. Line 3 initializes the cardinality of the transaction to η. Lines 4-5 remove the transactions that have less number of items than *min_length* threshold. Lines 6-12 process the transactions which are having more number of items than *min_length* threshold. Line 7 sorts all the transactions in descending order according to their utility values. Lines 8-9 initialize the β with zero value and count variables. Lines 10-11 calculate the RTU value of each transaction using variable β and count. Line 12 updates the utility of the transaction. Line 13 scans the dataset and calculates RLU of each item using the UA. Line 14 obtains the

Secondary items for itemset α by comparing the *RLU* of each item with the threshold *min_util*. These *Secondary* items are then consider in the extension of itemset α. Line 15 sorts the items in ascending order according to $\succ$ order as suggested in HUI-Miner [24]. Line 16 scans the dataset D and removes all the items those are not the member of *Secondary* for α. These removed items cannot be the member items of any HUIs as suggested by *Property 3.1.8*. Line 17 sorts all the remaining transaction by the $\succ_T$ order. Thereafter, the transaction merging performs. Line 18 scans the dataset again and calculates the *RSU* for each in *Secondary* for the itemset α. *UA* calculates the *RSU*. At the end of the procedure, Line 20 explores the itemset α by calling the recursive procedure *search* with *min_length*, *max_length*, α, *Primary*(α), *Secondary*(α) and *min_util*.

The **Algorithm 2** takes seven inputs such as α, the projected dataset $\alpha - D$, *Primary* items, *Secondary* items and three thresholds, *max_length*, *min_length* and *min_util*. Line 1 initializes with the length of the itemset α. Line 2 checks each time the length of α. If the number of items in α is not greater than *max_length*, then the algorithm recursively calls itself and extends the items of α. Otherwise, the algorithm returns back. The **foreach** loop in lines 3-11 extends the each item of α. Line 4 extends each single item of α with primary item Z. Line 5 scans the dataset and calculates the utility of itemset β. At the same time the projected dataset ($\beta - D$) is created. The transaction merging performs while creation of projected dataset. Line 6 checks the utility value of β with *min_util* threshold. If the utility of items in itemset β is not less than *min_util* then the itemset is a HUIs. Line 8 scans the dataset and calculates the *RSU* and *RLU* with Z and β using two *UAs*. Line 9 and line 10 determine the *Primary* and *Secondary* items for β respectively. Line 11 calls **Algorithm 2** to extend itemset β. Since the algorithm starts from single items and calls recursively to explore the search space of itemsets by appending single items at a time.

TABLE 3.5: Final HUIs of the running example

Itemset	Utility	Itemset	Utility
{D,C}	31	{D,E}	37
{D,C,B}	34	{C,B}	33
{D,C,E}	37	{C,B,E}	39
{D,B}	39	{B,E}	36
{D,B,E}	45	– –	– –

3.1.6 An Illustrative Example

In this section, a simple illustrative example is given to find HUIs from a transactional dataset. The transactional dataset (TABLE 2.1) and external utility (TABLE 2.2) are given in Section 2.1 in Chapter 2. We carry on the same datasets and example here. Moreover, min_util, min_length and max_length threshold are assumed as 30, 2 and 3 respectively.

The proposed algorithm uses *Definition 3.0.1* to overestimate the transaction utility. The TU and RTU of the running example are shown in TABLE 3.1. The RLU is also calculated using depth-first search which is equal to $RTWU$ as explained in Section 3.1.3. The TWU and $RTWU$ values of each item with different max_length threshold are shown in TABLE 3.2 and TABLE 3.3.

The utility value of items in RLU is 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 order $\succ$ of ascending $RTWU$. Thereafter, the items are removed which are not the elements of *Secondary* set. At the same time, empty transactions are removed from the dataset. And then the remaining transactions are sorted according to total order $\succ_T$. 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 2** finds descendant nodes in sub-tree using dept-first search. The *search* algorithm is recursively called to extend all items. The final HUIs of the running example are shown in TABLE 3.5.

TABLE 3.6: Statistical information about datasets

Dataset	# of transactions	# of distinct items	Avg. length	Max. Length
chess	3196	75	37	37
mushroom	8124	119	23	23

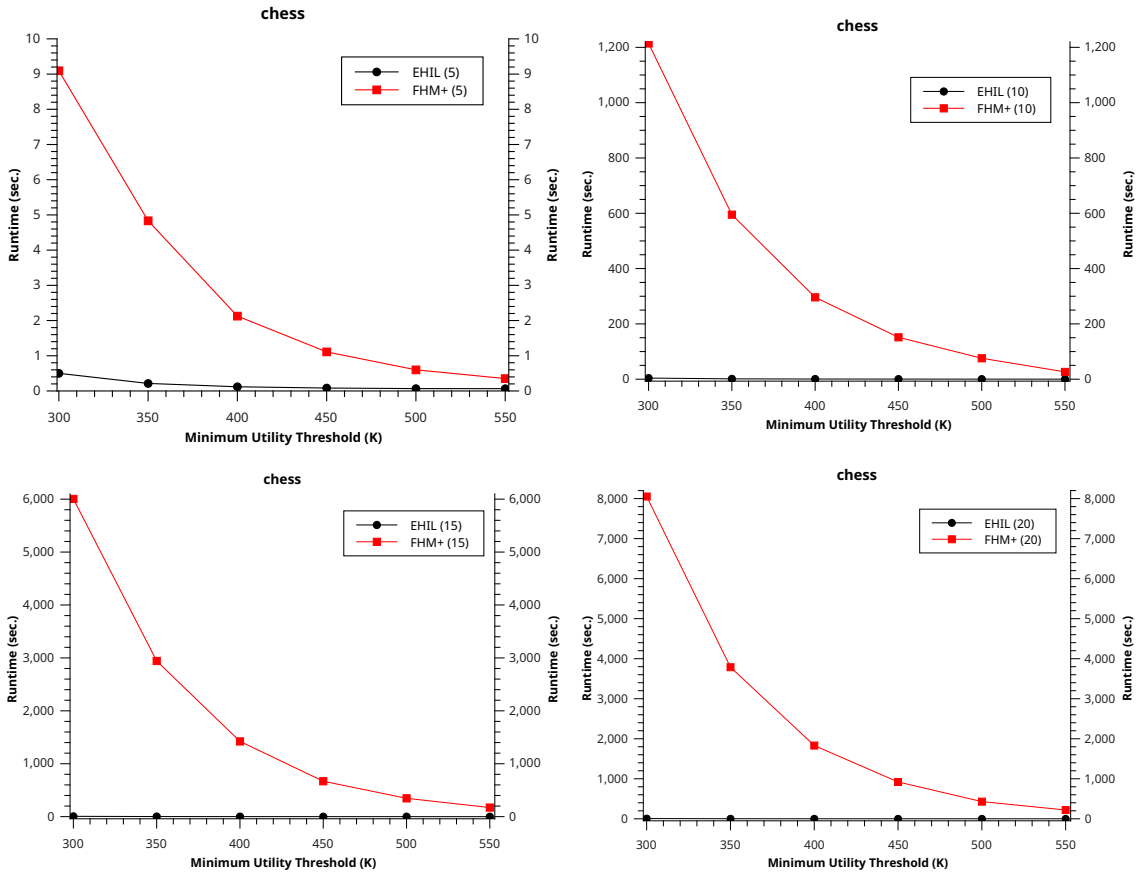

FIGURE 3.2: Execution times on chess dataset with different *max_length* constraints

3.2 Experimental Results

In this section, we compare the performance of EHIL with the state-of-the-art algorithm FHM+ [47]. All our experiments are performed on a Intel Core-i7-6700 machine, 3.40 GHz CPU with 8 GB of memory and Windows 10 Pro (64-bit Operating System). Both the algorithms are implemented in java. The source code of FHM+ algorithm is available at spmf [76]. TABLE 3.6 shows the detailed characteristics of all the datasets. We ran all our experiments ten times and reported the average results.

In order to analyze the performance of the algorithms in different situations, we test the algorithms with chess and mushroom datasets, which are available at spmf [76]. For comparing the proposed algorithm EHIL with FHM+, we execute on all datasets by decreasing *min_util*. *min_util* was decreased until both the algorithms take too much time or out of memory or a clear winner is observed. In the experiment, both the algorithms

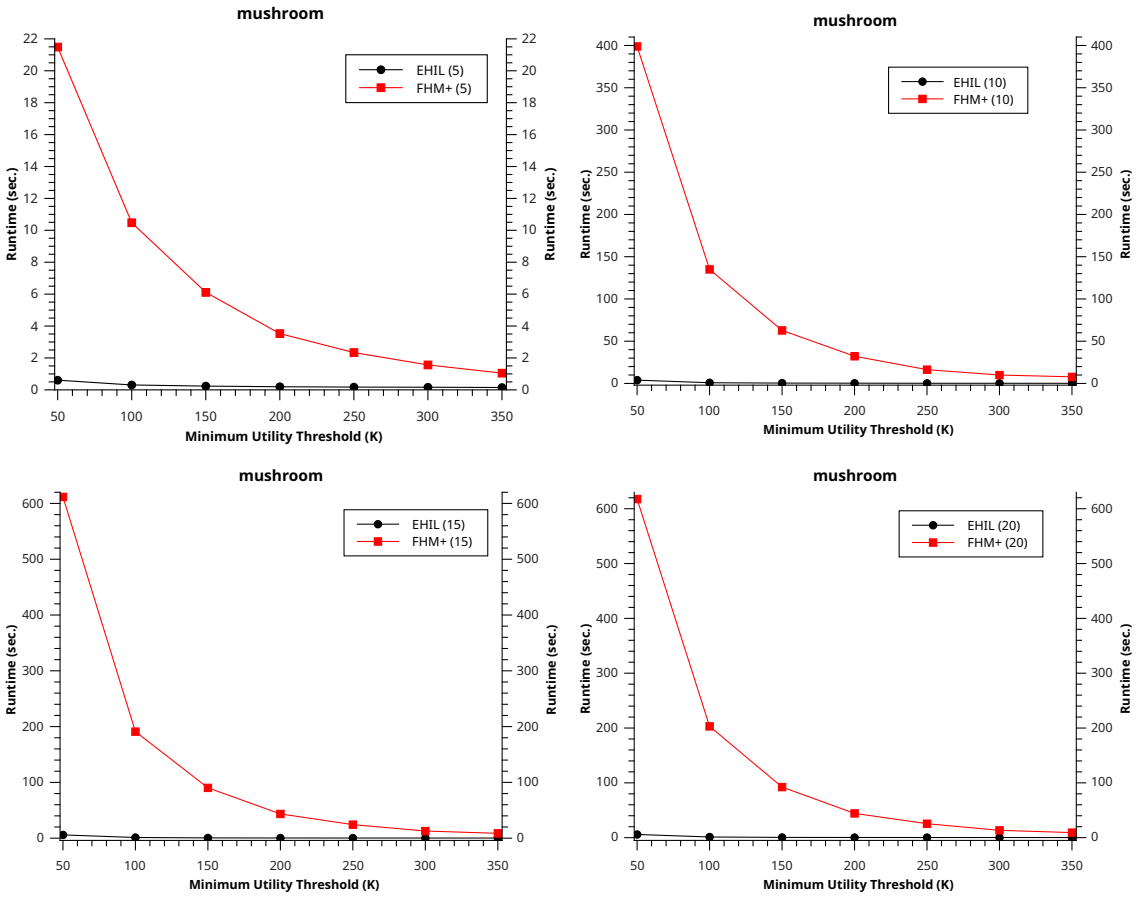

FIGURE 3.3: Execution times on mushroom dataset with different *max_length* constraints

are run with four different *max_length* threshold values (5, 10, 15, 20). A *min_length* threshold was set to 2 as it has no influence on efficiency. The *max_length* constraint greatly reduces the execution time and the number of patterns. It also consumes less memory than the state-of-the-art FHM+ algorithm.

3.2.1 Runtime Performance Comparison

FIGURE 3.2 and 3.3 show the execution time on chess and mushroom dataset respectively. FIGURE 3.2 and FIGURE 3.3 show that when *min_util* decrease, the runtime increases for both the algorithms. EHIL algorithm's execution time is far less sensitive to decrease of *min_util* threshold compare to FHM+ algorithm. EHIL algorithm outperforms because of transaction merging and dataset projection techniques to

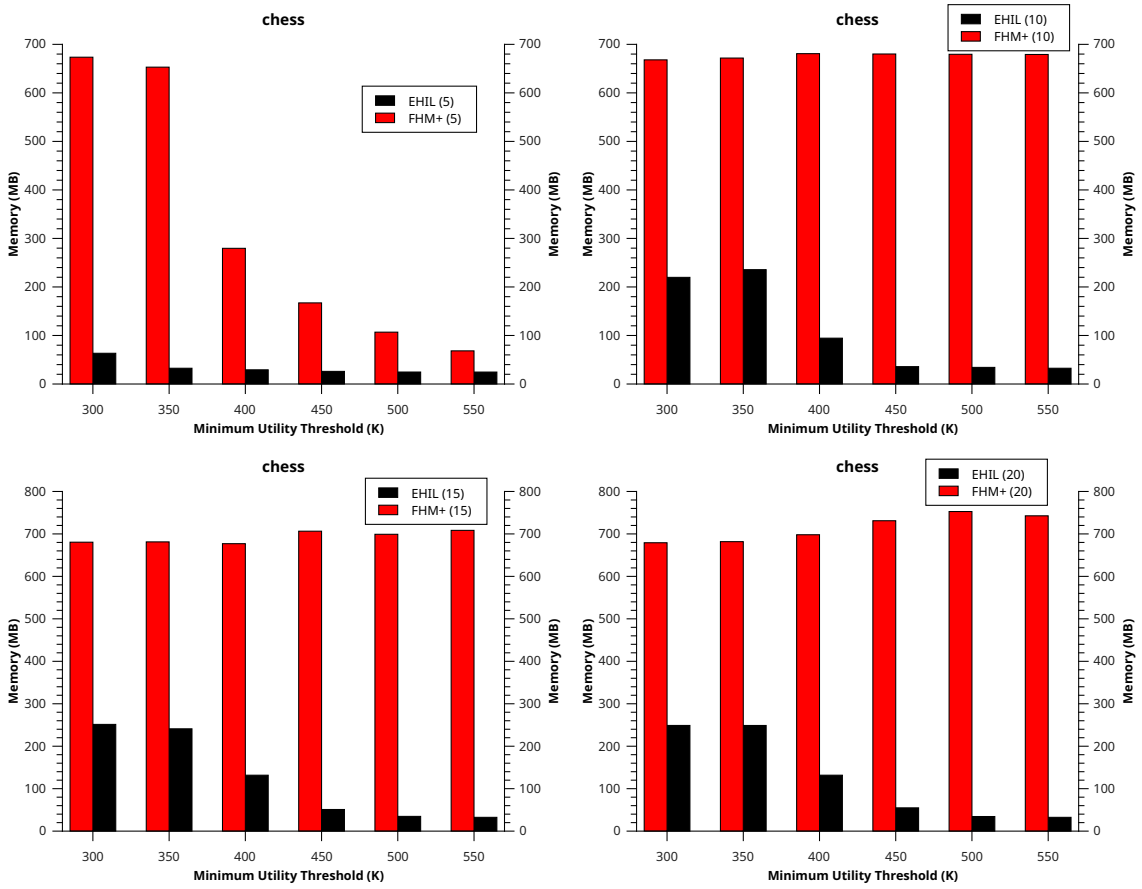

FIGURE 3.4: Memory usage on chess dataset with different *max_length* constraints

compact the dataset. Another reason for the better performance is array-based utility counting technique. Length based pruning strategies play an important role to decrease the execution time for EHIL algorithm.

3.2.2 Memory Performance Comparison

In this section, we compare the memory usage comparison of both the algorithms. The memory tests are conducted under the same experimental conditions as those of the runtime tests. We can observe that EHIL consumes the fewer memories in the experiments on both the datasets. FIGURE 3.4 and FIGURE 3.5 show the memory usage comparison for different values of *max_length* (5, 10, 15 and 20) on chess and mushroom datasets respectively. The memory usage increases when *min_util* threshold decreases.

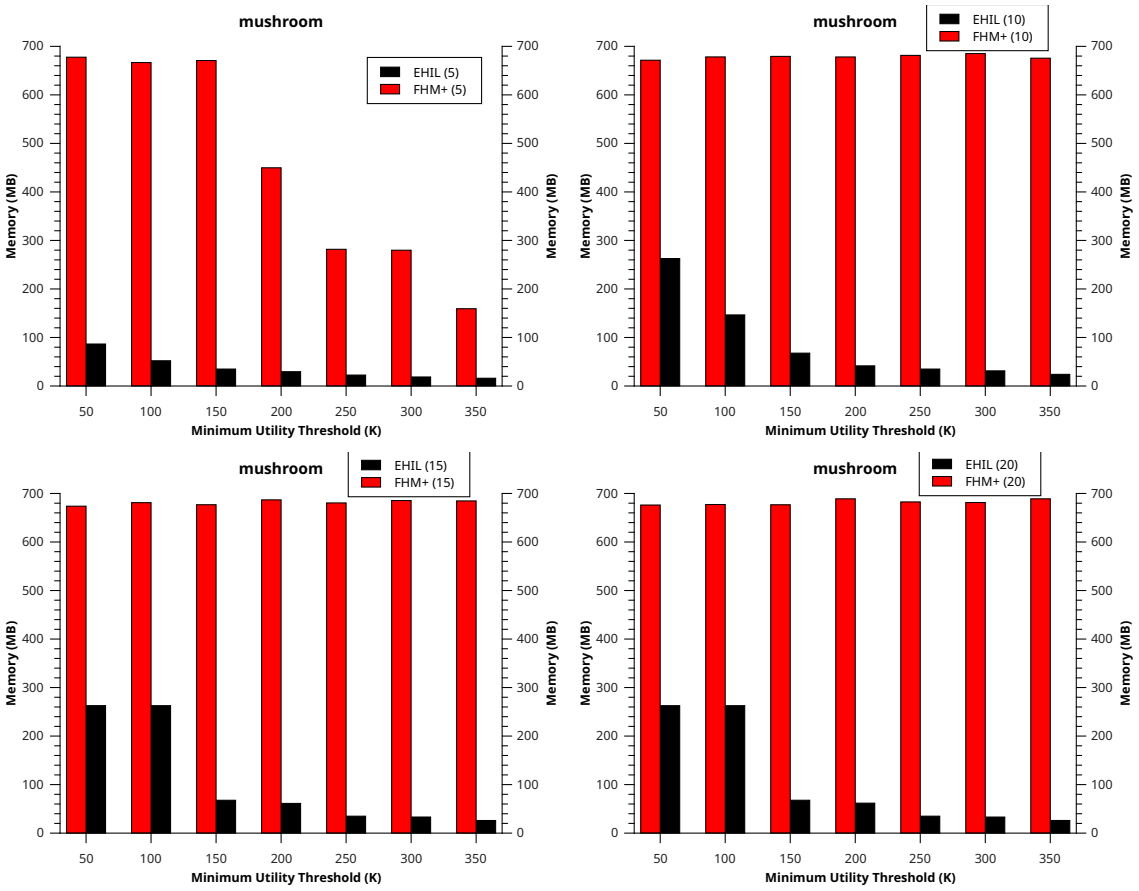

FIGURE 3.5: Memory usage on mushroom dataset with different *max_length* constraints

Figure shows that EHIL always consumes less memory than FHM+ on both the datasets and with all the *max_length* constraints.

TABLE 3.7: Relative runtime improvement analysis on chess dataset

min_util	Length (5)	Length (10)	Length (15)	Length (20)
550	5.351	142.422	862.058	1032.159
500	8.939	244.812	1125.723	1251.778
450	13.409	344.084	1343.069	1775.040
400	17.880	458.425	1429.643	1895.920
350	22.591	471.130	1354.621	1795.091
300	18.144	297.850	978.472	1242.498

TABLE 3.8: Relative runtime improvement analysis on mushroom dataset

min_util	Length (5)	Length (10)	Length (15)	Length (20)
350	7.104	40.815	36.803	39.906
300	9.459	47.629	52.634	48.981
250	13.475	74.806	86.427	87.179
200	17.923	102.704	128.227	131.247
150	25.909	141.253	170.888	185.771
100	34.008	157.456	154.112	163.281
50	35.275	101.403	104.396	105.635

TABLE 3.9: Relative memory consumption analysis on chess dataset

min_util	Length (5)	Length (10)	Length (15)	Length (20)
550	2.773	20.897	21.797	22.851
500	4.326	19.834	20.127	21.873
450	6.423	19.001	13.860	13.335
400	9.559	7.206	5.139	5.298
350	20.093	2.850	2.827	2.740
300	10.631	3.039	2.707	2.729

TABLE 3.10: Relative memory consumption analysis on mushroom dataset

min_util	Length (5)	Length (10)	Length (15)	Length (20)
350	9.927	28.087	26.280	26.41
300	15.020	21.928	20.647	20.553
250	12.488	19.502	19.482	19.531
200	15.244	16.331	11.216	11.152
150	19.197	10.028	9.991	9.983
100	12.816	4.629	2.593	2.578
50	7.829	2.556	2.564	2.574

TABLE 3.11: Relative runtime best, average and minimum comparison

dataset		Length (5)	Length (10)	Length (15)	Length (20)
chess	min	5.351	142.422	862.058	1032.159
	max	22.591	471.13	1429.643	1895.920
	avg	14.386	326.454	1182.265	1498.748
mushroom	min	7.104	40.815	36.803	39.906
	max	35.275	157.456	170.888	185.771
	avg	20.450	95.152	104.784	108.857

TABLE 3.12: Relative memory consumption best, average and minimum comparison

dataset		Length (5)	Length (10)	Length (15)	Length (20)
chess	min	2.773	2.850	2.707	2.729
	max	20.093	20.897	21.797	22.851
	avg	8.967	12.138	11.076	11.471
mushroom	min	7.829	2.556	2.564	2.574
	max	19.197	28.087	26.280	26.410
	avg	13.217	14.723	13.253	13.254

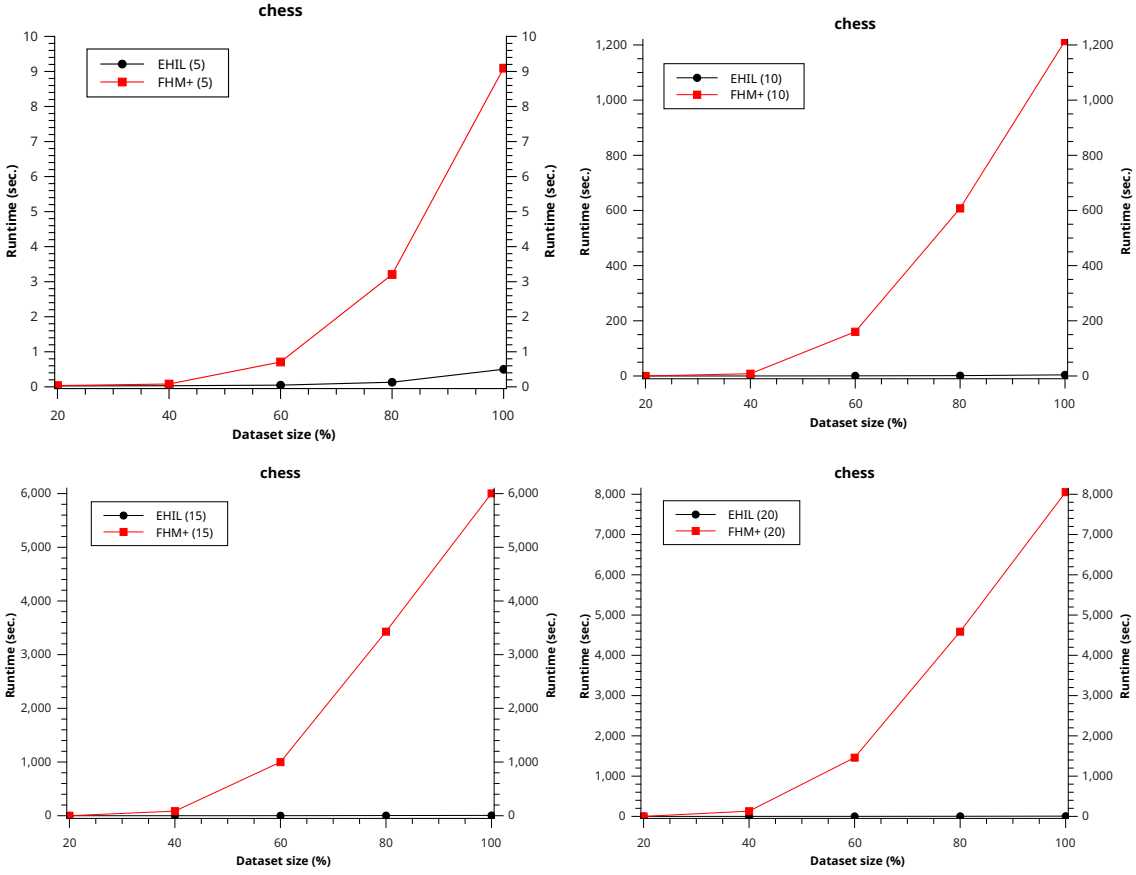

FIGURE 3.6: Scalability Runtime Comparison on chess dataset with different *max_length* constraints

3.2.3 Relative Runtime and Memory Comparison Analysis

In this section, we describe the relative runtime and relative memory comparison. We analyze the performance improvement by measuring the total execution runtime and total memory consumption by EHIL and FHM+. For this experiment, we took the total execution runtime by EHIL as the baseline (100%). The relative runtime is calculated as

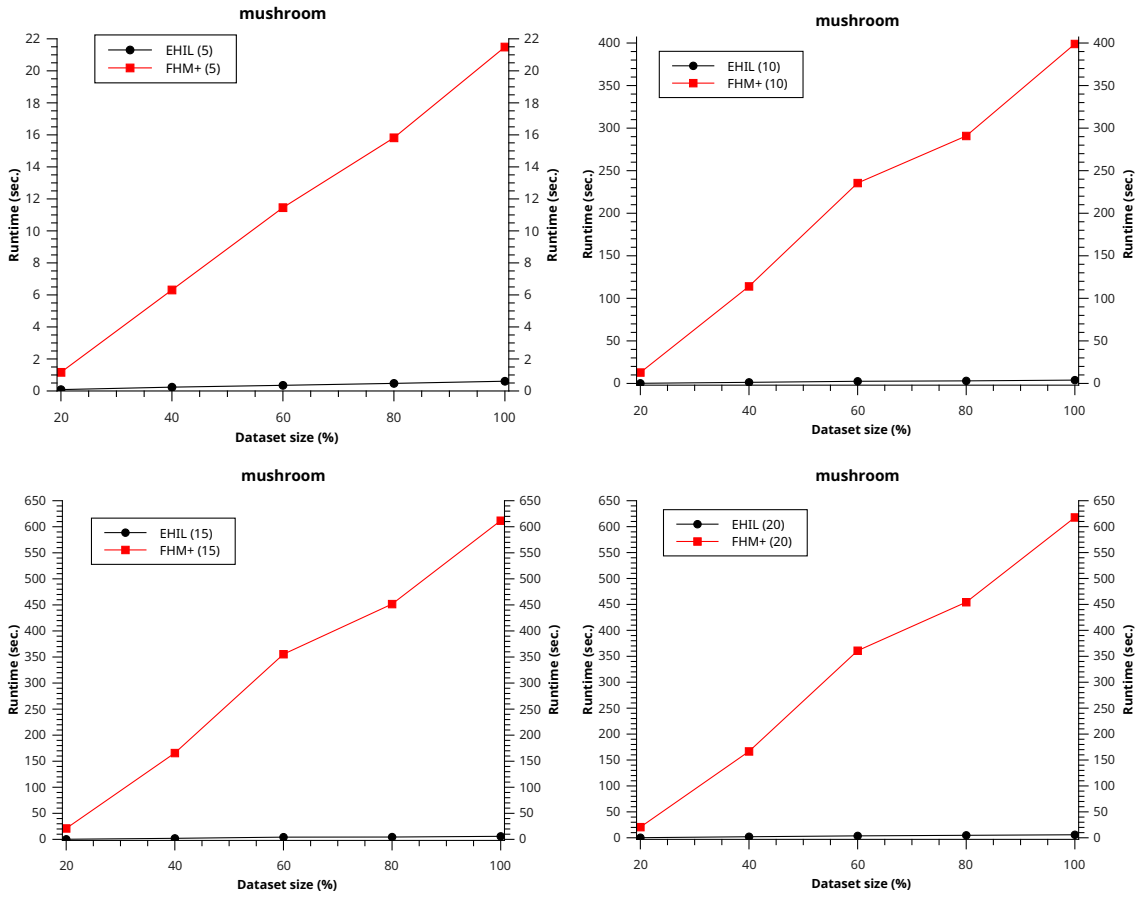

FIGURE 3.7: Scalability Runtime Comparison on mushroom dataset with different *max_length* constraints

the ratio of runtime of FHM+ algorithm to that of EHIL algorithm. Same phenomena are applied for the relative memory usage comparison.

3.2.3.1 Relative runtime comparison

TABLE 3.7 and TABLE 3.8 show the relative runtime comparison for different *max_length* (5, 10, 15 and 20) thresholds on chess and mushroom dataset respectively. These tables show that how much EHIL algorithm is faster than FHM+ algorithm. For example, EHIL algorithm is 5.351 times faster than FHM+ algorithm for *min_util* 550k and with *max_length* 5 on chess dataset.

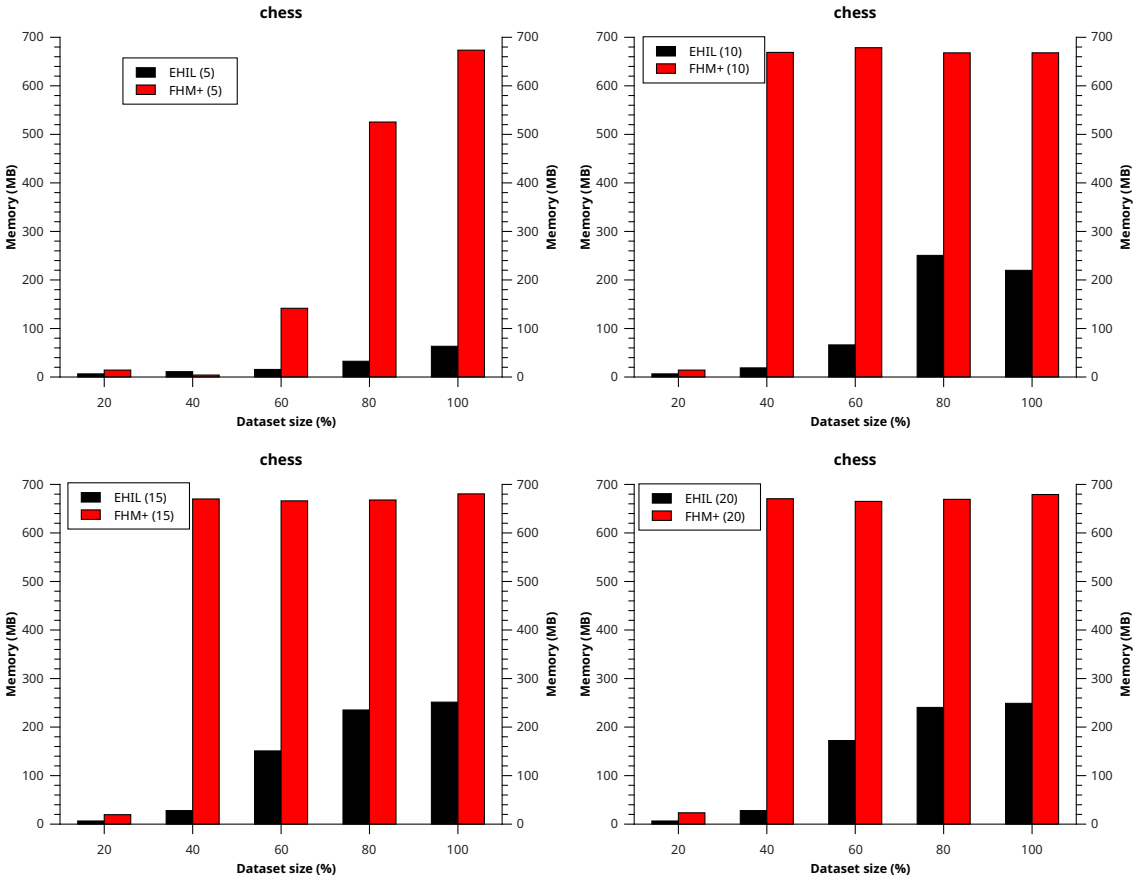

FIGURE 3.8: Scalability Memory Comparison on chess dataset with different *max_length* constraints

3.2.3.2 Relative memory comparison

TABLE 3.9 and TABLE 3.10 show the relative memory comparison for different *max_length* (5, 10, 15 and 20) thresholds on chess and mushroom datasets respectively. The relative memory comparison tests were conducted on the same experimental conditions as the relative runtime comparison tests. The comparison shows that EHIL algorithm consumes how many times less memory than FHM+ algorithm. For example, EHIL algorithm consumes 2.773 times less memory than FHM+ algorithm for *min_util* 550k and with *max_length* 5 on chess dataset.

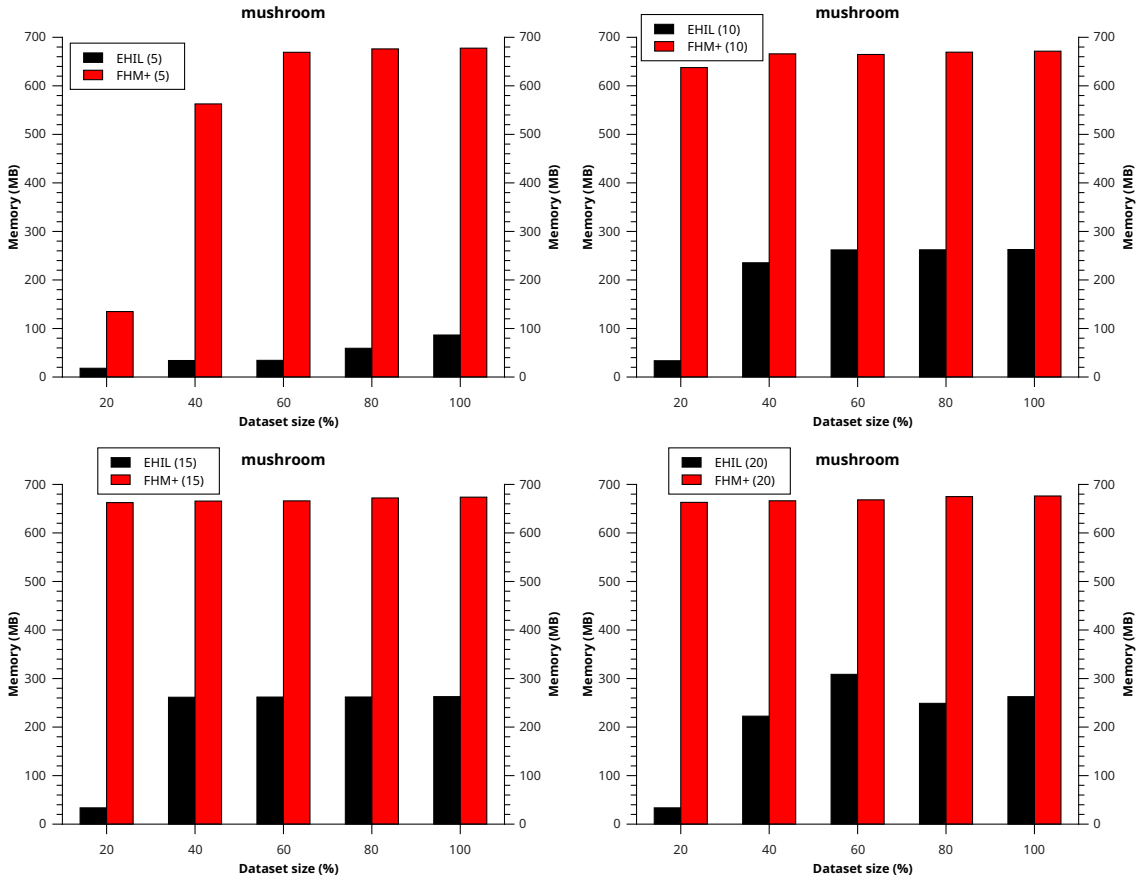

FIGURE 3.9: Scalability Memory Comparison on mushroom dataset with different *max_length* constraints

3.2.3.3 Relative best, average and minimum comparison

We now examine the relative best, average and minimum comparison of EHIL and FHM+ algorithm. TABLE 3.11 and TABLE 3.12 show that EHIL algorithm always performs better than FHM+ algorithm on all (best, average and minimum) cases on both the datasets. TABLE 3.11 shows how much EHIL algorithm is faster than FHM+ algorithm in maximum, average and minimum cases and TABLE 3.12 shows that how many times EHIL consume less memory than FHM+ in maximum, average and minimum cases.

3.2.4 Scalability

In this section, we evaluate the scalability of proposed algorithm EHIL. For the scalability comparison experiment, min_util threshold is fixed to the lowest minimum threshold that is used in each dataset for runtime and memory experiment. As shown in FIGURE 3.6 and 3.7, EHIL has a good scalability with varying size of the datasets. The size of the datasets is varied from 20% to 100% to evaluate the scalability results.

The runtime of EHIL linearly increase or remain constant as the dataset size increases. The runtime of FHM+ algorithm increases exponentially when the size of the dataset increases. The difference between the runtime of both algorithms grows wider when the dataset size increases as shown in FIGURE 3.6 and 3.7. The memory usage of EHIL is almost stable with the increased size of the dataset. FHM+ algorithm memory usage increases rapidly as shown in FIGURE 3.8 and 3.9.

3.3 Summary

In this chapter, we tackle with the problem of mining too many very small and very large itemsets. We proposed a novel algorithm that mines HUIs considering length constraints. The minimum and maximum length constraints restrict the length of HUIs. The minimum length constraint removes very small itemsets and maximum length constraint restrict the higher length of the itemsets. We redefined sub-tree and TWU pruning strategies and incorporate length constraints in these pruning strategies. We use transaction merging and dataset projection based merging as a preprocessing of the dataset which reduce the dataset scanning cost. We utilized array-based utility counting technique that calculates the utility in negligible runtime and memory space. We present a detailed example to show the working of proposed algorithm with dummy transactional datasets. The experimental results show that proposed algorithm EHIL greatly reduced the runtime and memory requirements. The results also show that EHIL algorithm outperforms the state-of-the-art algorithm FHM+ for both in runtime and memory for all our observations. We have seen that when decrease in minimum utility threshold, runtime and memory requirements increase for FHM+ algorithms, whereas EHIL algorithm is almost constant. The relative runtime comparison shows that EHIL is

maximum up to 1896 times and minimum 5 times faster than FHM+. In memory comparison, EHIL usages is maximum up to 28 times and minimum 2.5 times less memory than FHM+ algorithm. The experimental results show that EHIL outperforms FHM+ for all tested cases with all *max_length* constraints. The scalability analysis shows that runtime of EHIL is increase linearly where runtime of FHM+ increase exponentially. The proposed algorithm is more scalable for memory usage comparison.