# Chapter 5

# High Utility Itemsets Mining with Negative Utility Value

In Chapter 3 and Chapter 4, we have developed efficient pattern-growth methods to mine HUIs. These algorithms work only for itemsets with positive utility values. However, in the real-world, items are found with both the positive and the negative utility values. To address this issue, we propose an algorithm named EHIN (**E**fficient **H**igh utility **I**temsets mining with **N**egative utility) to find all HUIs with negative utility.

Several HUIs mining algorithms [11, 12, 21, 22, 33, 78] use $TWU$ based upper bound to prune the search space and discover HUIs. $TWU$ is interesting because it supports overestimation for HUIs mining and it is also used to prune search space. $TWU$ is widely used in positive utility based utility mining, such as in [11, 12, 21, 33, 78, 79]. $TWU$ measure cannot be directly applied on HUIs with negative utility because it assumes that there is no any item with negative utility value. To handle this type of problem, Chu et al. proposed HUINIV-Mine algorithm to discover HUIs for both positive and negative utility items [29]. And for overestimation, the utility is counted by summation of positive utilities value only. For understanding the concepts of HUIs with negative utility we utilize transactional datasets presented in TABLE 2.8 and TABLE 2.9 in Chapter 2. The preliminarily definitions are presented in Section 2.5.

Chu et al. discussed that it is very expensive to consider both the positive and negative utility in terms of time and memory [29]. Fournier-Viger et al. also used $RTU$ to find

HUIs with negative utility [30]. In recent years, single-phase algorithms such as d2HUP [26], HUI-Miner [24], FHM [27], HUP-Miner [25] and EFIM [28] are also used with ordinary $TU$ because of their positive utility nature. The proposed algorithm is variation of the EFIM algorithm. Now we present the utility-list to calculate the remaining utility of negative utility itemsets.

*Definition* 5.0.1. (Utility-list). Let $\succ$ be a total order on items from $I$, for an itemset $X$ in a dataset $D$, The utility-list [24, 25, 27, 28, 61, 80] contains three fields as $tid$, $iutil$ and $rutil$ for each transaction $T_j$ where $T_j$ contains itemset $X$. Where $iutil$ indicates the utility and $rutil$ indicate the remaining utility $(RU)$ of an itemset $X$ in $T_j$ (see [24, 25] for details of utility-list and remaining utility).

The $\succ$ order is defined as the order of $TWU$ ascending because it reduced the search space. In running example, the order of items as $a$, $c$, $d$, $b$ and $e$, in increasing $TWU$ order.

*Property* 5.0.1. (Pruning search space using utility-list). For an itemset $X$, if the sum of $U(X) + RU(X)$ is less than $min\_util$, then itemset $X$ and it's all supersets are low utility itemsets. Otherwise, the itemset is HUIs. The detail and proof of remaining utility upper bound $(REU)$ based upper bound are given in [24].

For example, let us consider the running example. The utility-list of item $\{a\}$ is $\{(T_1,4,7)$, $(T_5,4,0)$, $(T_6,4,13)\}$ and hence the utility is $11+4+17 = 32$. The utility-list of itemset $\{a,d\}$ is $\{(T_1,8,3)$, $(T_6,12,1)\}$ and hence the utility is 24.

To deal with negative utility values, FHN introduced the modified structure of utility-list. FHN contains four fields as $tid$, $putil$, $nutil$ and $rputil$ for each transaction where $putil$ indicates the positive utility $nutil$ indicates the negative utility and $rputil$ indicates the remaining positive utility of an itemset. The $\succ$ indicates the total order of sorted items which is also in $RTWU$ ascending order for positive utility items and negative items come after the positive utility items. We use sub-tree based strategy instead of utility-list based pruning strategy.

Most of the one-phase algorithms use utility-list structure to mine HUIs. First time, Liu et al. introduced the utility-list structure and remaining utility upper-bound. The remaining utility upper-bound is tighter than $TWU$. Hence, utility-list based upper bound is widely used in HUIs mining. The utility-list structure is utilized by HUI-Miner, HUP-Miner

and FHM algorithms to discover the complete set of HUIs for positive utility. Utility-list structure cannot be directly applied for items with negative utility. FHN modifies the basic structure of utility-list and finds complete set of HUIs. However, utility-list structure and FHN are very expensive in terms of execution time and memory consumption. FHN suffers from high memory and execution time to process the each itemset in search space. The utility-list gives $O(n)$ in worst case when it contains an entry for each transaction ($n$ is the number of transactions). Hence, it consumes huge memory space to maintain the utility-list. Moreover, building utility-list is expensive in terms of time complexity. FHN needs four utility-list to join small itemsets which need $O(n^4)$ time in worst-case [30]. To join the utility-list is the main bottleneck in terms of execution time and memory usage. The utility-list may considers the itemsets that do not appear in the dataset. To address the above limitations, we attempt to design an pattern-growth approach based efficient algorithm to mine HUIs with positive and negative utility values.

## 5.1 EHIN Algorithm

In this section, we propose an efficient algorithm for mining HUIs with negative utility items, namely EHIN. EHIN utilizes dataset projection and transaction merging techniques to reduce the dataset scanning. EHIN utilizes the proposed upper-bounds, redefined sub-tree utility and redefined local utility to prune unpromising itemsets. EHIN also utilizes array-based utility counting technique to efficiently calculate the upper-bounds. The proposed algorithm utilizes various properties and pruning strategies to mine HUIs with negative utility.

### 5.1.1 The Search Space

The search space of HUIs mining problem has been represented as a set-enumeration tree [24, 27, 28]. The items are represented by $\succ$ total order of sorted items. During implementation ordering of itemsets is done according to increasing $RTWU$ as this reduces the search space for HUIs. But for the sake of simplicity in the running example, a lexicographical ordering has been assumed instead of $\succ$ order. The set-enumeration tree of items $a, b, c, d$ and $e$ for the lexicographical order is shown in FIGURE 5.1. All $2^m$

itemsets can be represented by a set-enumeration tree, where $m$ is the number of items in a dataset. This representation of itemsets has been shown in set-enumeration tree in FIGURE 5.1 and this would become search space for proposed algorithm. The proposed algorithm explore the search space using a depth-first search. The proposed algorithm recursively appends one item at a time to itemset $\alpha$ according to the $\succ$ order using dept-first search and generates larger itemsets as shown in FIGURE 5.1. Given a set-enumeration tree, an itemset represented by a node is called an extension of an itemset represented by an ancestor node of the node. Now we present some definitions that are related to explorations of itemsets in a dept-first search.

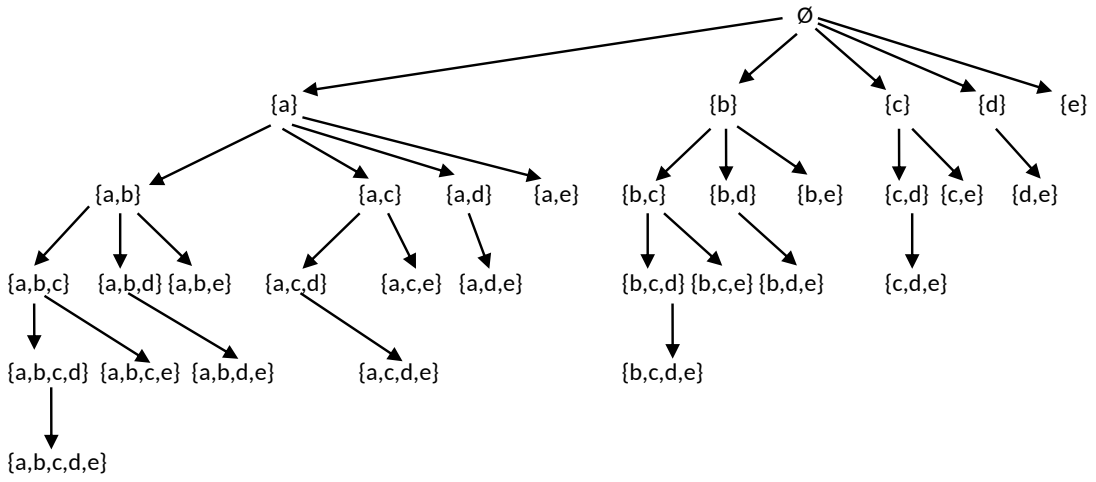


FIGURE 5.1: Set-enumeration tree for $I = \{a,b,c,d,e\}$

## 5.1.2 Efficient Dataset Scanning Techniques

As we discussed earlier, EHIN calculates the utility and upper bounds through dataset scans. So we need dataset reduction operation to reduce the dataset scanning time. Hence, EHIN utilizes dataset projection and transaction merging techniques.

### 5.1.2.1 Dataset Scanning using Projection

Dataset projection techniques are utilized in this work to reduce the memory requirement and speed up the execution of mining processing. Dataset projection technique relies on

the observation that when an itemset $\alpha$ is taken into account with the depth-first search. The items which do not belong to the extended itemset $E(\alpha)$ are ignored while scanning the dataset to calculate the utility of itemsets within the sub-tree of itemset $\alpha$. A dataset without these items is called a projected dataset [28, 61, 80].

*Definition* 5.1.1. (Projected transaction). For the itemset $\alpha$, the projected transaction $T$ is denoted as $\alpha{-}T$ and is defined as $\alpha{-}T = \{i \mid i{\in}T \wedge i{\in}E(\alpha)\}$ [28].

*Definition* 5.1.2. (Projected dataset). For the itemset $\alpha$, the projected dataset $D$ is denoted as $\alpha{-}D$ and is defined as $\alpha{-}D = \{\alpha{-}T \mid T{\in}D \wedge \alpha{-}T{\neq}\emptyset\}$ [28].

In the running example, consider dataset $D$ and $\alpha = \{c\}$. The projected dataset $\alpha - D$ contains transactions $\alpha - T_2 = \{e\}, \alpha - T_3 = \{d,e\}, \alpha - T_4 = \{d,e\}, \alpha - T_6 = \{d,e\}$ and $\alpha - T_7 = \{e\}$.

The projection technique reduces the dataset scanning cost and hence, larger transactions become smaller. To implement projection technique efficiently, each original transaction is sorted according to the $\succ$ total order. How to efficiently implement $\succ$ is an important issue regarding the dataset projection. The basic or ordinary technique is to copy the transactions for each projection. Some dataset projection techniques are performed earlier in IHUP [11], UP-Growth [12], UP-Growth+ [21] and MU-Growth [56]. But these are not efficient techniques for dataset projection. Our dataset projection approach is inspired by EFIM algorithm's projection technique. But before applying projection technique, we need to perform sorting among items in each transaction according to the $\succ_T$ total order. After that, pseudo-projection is performed in each projected transaction. This pseudo-projection is pointed by offset pointer on the corresponding original transaction. EHIN performs dataset projection in linear time and space complexity. Even so, as the longest itemsets are explored, the size of the projected dataset is decreased.

#### 5.1.2.2 Dataset Scanning using Transaction Merging

To further reduce the dataset scanning cost, a transaction merging technique is used. This is relayed on the observation as the dataset contain the identical transactions. Identical transactions contain the exactly same items but may not have same quantity values (internal utility). The merging technique identify these identical transactions and replace them with a single transaction.

*Definition* 5.1.3. (Identical transactions). A transaction $T_a$ is identical to a transaction $T_b$ if they contain the same items. The identical transactions may not have the same quantity values.

*Definition* 5.1.4. (Transaction merging). For the dataset $D$ and the identical transactions as $T_{a_1}, T_{a_2}, T_{a_3}, T_{a_n}$ replaced by a new transaction $T_M = T_{a_1} = T_{a_2} = T_{a_3} = T_{a_n}$ (Identical transactions may not contain the same quantity values of each item). And the quantity of these identical transactions is $k \in T_M$ and therefore is defined as $IU(k, T_M) = \sum_{i=1\ldots n} IU(k, T_{a_i})$. To achieve more dataset reduction, we need to merge the transaction in projected datasets. The projected transactions merging produces higher dataset reduction than original transaction merging because projected transactions are smaller than original transactions. The reason behind this is that the projected transactions could be more likely identical.

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 $IU(b, T_M) = 4$, $IU(c, T_M) = 7$ and $IU(e, T_M) = 3$.

*Definition* 5.1.5. (Projected transaction merging). Let the identical transaction as $T_{a_1}, T_{a_2}, T_{a_3}, T_{a_n}$ in the dataset $\alpha - D$ is replaced by a new transaction $T_M = T_{a_1} = T_{a_2} = T_{a_3} = T_{a_n}$. And quantity of these identical transactions is $k \in T_M$ and is defined as $IU(k, T_M) = \sum_{i=1\ldots n} IU(k, T_{a_i})$ [28].

For example, consider dataset $D$ of our running example and $\alpha = \{c\}$. The projected dataset $\alpha - D$ contains transactions $\alpha - T_2 = \{e\}, \alpha - T_3 = \{d, e\}, \alpha - T_4 = \{d, e\}, \alpha - T_6 = \{d, e\}$ and $\alpha - T_7 = \{e\}$. Transactions $\alpha - T_3$, $\alpha - T_4$ and $\alpha - T_6$ can be replaced by a new transaction $T_{3\_4\_6} = \{d, e, \}$ where $IU(d, T_{3\_4\_6}) = 6$ and $IU(e, T_{3\_4\_6}) = 6$.

Transaction merging technique is desirable for reducing the size of the dataset. But identifying the identical transactions is the main problem to implement transaction merging technique. To achieve this, we need to compare all the transactions with each other. But this technique to compare all the transactions among one another is not an efficient technique. We present following approach to overcome this problem.

*Definition* 5.1.6. (Total order on the transactions). For the dataset $D$, the total order $\succ_T$ is defined as the lexicographical order when the transactions are read backwards. For more details of total order $\succ_T$ on the transactions follow [28, 61].

We sort the original dataset by following a new total order $\succ_T$ on the transactions. For example, let us suppose the transactions be $T_x = \{a,b,c\}, T_y = \{b,c\}$ and $T_z = \{a,b,e\}$. We have that $T_z \succ_T T_x \succ_T T_y$. The dataset sorted according to the $\succ_T$ order provides following property.

*Property* 5.1.1. (Transaction order in $\succ_T$ sorted dataset). Let us sort the dataset $D$ according to $\succ_T$, where $\succ_T$ is defined as lexicographical order when the transactions are read backwards. Identical transactions appear consecutively in the projected dataset $\alpha - D$. More details and proof see [28].

The projected dataset follows the above property where we get all the identical transactions by comparing each transaction with the next transaction. Therefore, by using the above property, the transaction merging technique can be performed easily by scanning the dataset efficiently. Accordingly, we find the identical transactions in the projected dataset. 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. For a projected dataset $\alpha - D$, the identical transactions always appear consecutively. We find this when we read the transaction from backward. All the one-phase HUIs mining algorithms do not perform transaction merging technique except the EFIM and EFIM-Closed. The utility-list based algorithms like FHM [27], HUP-Miner [25] and hyper-link based algorithms, d2HUP [26] do not use transaction merging because they use a vertical dataset representation.

### 5.1.3 Pruning Strategies

In this subsection, we present two new upper bound named redefined sub-tree utility and redefined local utility. As we discussed earlier, these upper bounds are more tighter. These upper bounds are mathematically equivalent to remaining utility (RU) and $TWU$ based upper bounds. The key difference is that utilized upper bounds are computed while traversing the sub-tree in enumeration-tree.

#### 5.1.3.1 Prune search space using redefined Local Utility

*Definition* 5.1.7. (Redefined local utility). The local utility of item $z$, with respect to itemset $\alpha$ is defined as below, where item $z \in E(\alpha)$. $RLU(\alpha,z) = \sum_{T \in (\alpha \cup \{z\})} [U(\alpha,T) + RE(\alpha,T)]$ [28].

For example, if itemset $\alpha = \{a\}$, we calculate $RLU$ as $RLU(\alpha,b) = (11+17) = 28$, $RLU(\alpha,c) = 17$, $RLU(\alpha,d) = 28$ and $RLU(\alpha,e) = 28$.

*Property* 5.1.2. (Redefined local utility based overestimation). Let an itemset $\alpha$ and an item $z$, where $z \in E(\alpha)$. Let $z$ can be an extension of $\alpha$, such that $z \in Z$. Therefore, $RLU(\alpha,z) \geq U(Z)$ always holds. The proof is shown in [28].

**Pruning using redefined local utility** Let an itemset $\alpha$ and an item $z$, where $z \in E(\alpha)$. If $RLU(\alpha,z) < min\_util$, then the single item $z$ and all extensions of $\alpha$ containing item $z$ are low-utility in a sub-tree. Furthermore, item $z$ is ignored for exploring all sub-trees of $\alpha$.

#### 5.1.3.2 Prune search space using redefined sub-tree Utility

*Definition* 5.1.8. (Redefined sub-tree utility). Let the itemset $\alpha$ and the item $z$, where $z \in E(\alpha)$, that can be extended $\alpha$ to follow the depth-first search to the sub-tree. The Sub-tree utility of the item $z$, if $\alpha$ 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 example, $\alpha = \{a\}$. We have that $RSU(\alpha,c) = (5+1+2)+(10+6+11)+(5+1+20) = 61$, $RSU(\alpha,d) = 25$ and $RSU(\alpha,e) = 34$.

*Property* 5.1.3. (Redefined sub-tree utility based overestimation). Let the itemset $\alpha$ and the item $z$, where $z \in E(\alpha)$. The utility value of $RSU(\alpha,z)$ $U(\alpha \cup \{z\})$ such as $RSU(\alpha,z) \geq U(Z)$ keeps the extension $Z$ of $\alpha \cup \{z\}$. The proof is shown in [28].

**Pruning using the redefined sub-tree utility** Let the itemset $\alpha$ and the item $z$, where $z \in E(\alpha)$. 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. Some sub-tree can be pruned of an itemset $\alpha$. Therefore, the number of sub-trees is reduced. Hence, the search space is reduced.

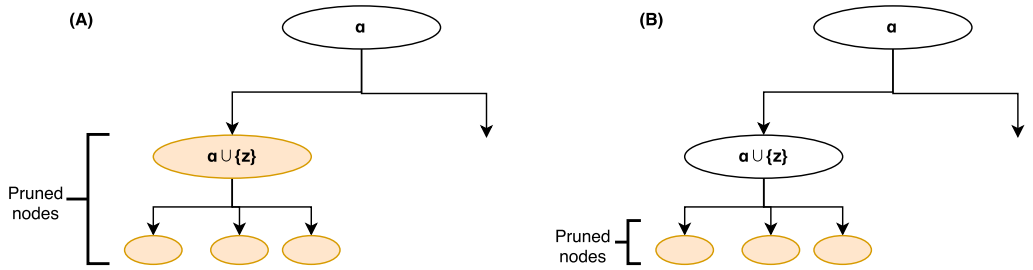


FIGURE 5.2: Comparison of RSU (left) and REU (right) upper bounds

The relationship between the proposed upper-bounds and the main ones used in the previous work and the relationship between the proposed upper-bounds ($TWU$, $REU$, $RLU$ and $RSU$) and the upper-bound used in the work ($TWU$, $REU$) are given as follows.

*Property* 5.1.4. For the itemset $\alpha$, the item $z$ and an itemset $Y = \alpha \cup \{z\}$. The relationship $TWU(Y) \geq RLU(\alpha, z) \geq REU(Y) = RSU(\alpha, z)$ holds [28].

Above property shows the relationship between the utilized upper bounds and traditionally used upper bounds ($TWU$ and $REU$). The redefined local utility upper bounds are more tighter than $TWU$ which is widely used in two-phase based algorithms to prune the search space. Hence, utilized local utility upper bound can be more efficient. As shown in the above relationship of upper bounds, the $RSU$ and $REU$ are mathematical equivalents [28]. The major difference is that redefined sub-tree utility upper bound is calculated during the dept-first search in search space at the itemset $\alpha$ and $REU$ is calculated at the child itemsets of $\alpha$ using join the utility-lists. Hence, the redefined sub-tree utility can prune the whole sub-tree of $\alpha$ including node $z$ and $REU$ prune only the child nodes of $\alpha$ against $min\_util$ threshold as shown in FIGURE 5.2. Therefore, we utilize redefined sub-tree utility upper bounds than $REU$ upper bound for prune the search space. Later in this chapter, for an itemset $\alpha$, we categorize the itemsets as *primary* and *Secondary*.

*Definition* 5.1.9. (*Primary* and *Secondary* items). For an itemset $\alpha$, the *Primary* items of itemset $\alpha$ are denoted as $Primary(\alpha) = \{z \mid z \in E(\alpha) \wedge RSU(\alpha, z) \geq min\_util\}$. The itemset $\alpha$ 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 $Primary(\alpha) \subseteq Secondary(\alpha)$ [28].

The itemsets whose redefined sub-tree utility are not less than $min\_util$ are denoted as *Primary* items and itemsets whose local utility is not less than $min\_util$ are denoted as *Secondary* items.

### 5.1.4 Calculate Upper Bounds using Utility Array

Previously, we presented new upper-bounds for pruning. Now we discuss an array-based approach to calculate upper-bounds, it is an array-based structure named as utility-array ($UA$) [28, 61, 80].

*Definition* 5.1.10. (Utility Array). For the set of items $I$ that are appeared in a dataset $D$. The $UA$ has the length $\mid I \mid$. The entry for an item $z$ in the array is denoted as $UA[z]$. Each entry is used to store a utility value.

*Calculate $RLU(\alpha)$ using $UA$.*
Initially, $UA$ is initialized to 0. Then the $UA[z]$ for all item $z \in T_j \cap E(\alpha)$ is calculated as $UA[z] = UA[z] + U(\alpha,T) + RE(\alpha,T)$ for each transaction $T_j$ of the dataset $D$. After the dataset scanning, the $UA[k]$ contains utility of $RLU(\alpha,k) \forall k \in E(\alpha)$. After processing the last transaction we have the local utility of all positive itemset.

*Calculate $RSU(\alpha)$ using $UA$.*
Initially, $UA$ is initialized to 0. Then the $UA[z]$ for all 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 $RSU(\alpha,k) \forall k \in E(\alpha)$.

All the above techniques for calculating upper-bound are highly efficient. They can calculate all the upper-bounds by performing only one scan of projected dataset. They take linear time to calculate all upper-bound, whereas HUI-Miner, HUP-Miner and FHM calculate upper-bounds by performing join the utility-list that take up-to $O(3N)$ time. We also observe that *UAs* are very compact and efficient data structure. Moreover, to utilize *UAs* efficiently, we utilize three optimizations. Firstly, all items in the database are renamed as consecutive integers. Then, $UA[i]$ for an item $x$ is stored in the $i^{th}$ position of the array. Using this optimization, every item is accessed in $O(1)$ time. Secondly, the same created arrays are reused after reinitializing. For example, the same $UA$ is reused to calculate the redefined local utility for any itemset that was used earlier to calculate the local utility for another itemset. Hence, memory requirement is greatly reduced. In this work, only two utility-arrays are used to calculate of redefined sub-tree utility and the redefined local utility. Hence, the proposed algorithm uses very low memory compared to FHN and HUINIV-Mine algorithm. Memory comparison is shown in experimental result in memory uses Section 5.2.3. Lastly, for faster re-initialization of the utility-array

where single item extension of an itemset is $\alpha$, we only reinitialize the corresponding utility-array to calculate the upper-bound.

The characteristics of the existing algorithms and our proposed algorithm are depicted in TABLE 5.1. Table shows the pruning strategies used, the state-of-the-art algorithms for proposed algorithm and existing algorithms. Table also shows that an algorithm is extension of which algorithm. It also shows the runtime and memory comparison with the state-of-the-art algorithm.

TABLE 5.1: Comparison of the characteristics of the existing algorithms with our proposed algorithm EHIN

| Algorithm | Phase | Pruning strategies | the state-of-art algorithm | Extension of | Search | Dataset | Runtime comparison with the state-of-art algorithm | Memory comparison with the state-of-the-art algorithm |
|---|---|---|---|---|---|---|---|---|
| HUINIV-Mine, 2009 | Two-phase | *TWU* | – | Two-phase algorithm | Breadth-first search | Transactional | – | – |
| FHN, 2016 | One-phase | EUCS, LA-Prune | HUINIV-Mine | FHM algorithm | Depth-first search | Transactional | Two to three Times faster | Upto 200 times less memory |
| **EHIN** (proposed) | One-phase | *RLU*, *RSU* | FHN | EFIM algorithm | Depth-first search | Transactional | Three to four Times faster | Up-to 10 times less memory |

The proposed algorithm EHIN utilizes several novel ideas explained in the previous sections. The main **Algorithm 8** takes a transactional dataset and threshold *min_util* as input. Lines 1-2 in the algorithm initially considers the empty itemset as $\alpha$, a set of positive utility as $\rho$ and a set of negative utility items as $\eta$. Line 3 calculates the local utility of each item using the *UA*. After that in line 4, the *Secondary* items for itemset $\alpha$ are obtained by comparing the local utility of each item with the threshold *min_util*. These *Secondary* items are then considered in the extension of itemset $\alpha$. Thereafter, line 5 sorts *Secondary* items by ascending order of $RTWU$. The algorithm considers the positive utility items followed by negative utility items while sorting in $\succ$ order. Line 6 scans the dataset $D$ to remove all the items which are not the member of the *Secondary* itemset for $\alpha$. These removed items cannot be the member items of any HUIs (Pruning using the redefined local utility). Thereafter, line 7 scans the dataset backward to sort

**Algorithm 8:** EHIN algorithm

**Input:** $D$: a tranasctional dataset, user-specified threshold: $min\_util$

**Output:** High utility itemsets (HUIs)

1 $\alpha \leftarrow \emptyset$;
2 $\eta \leftarrow$ set of negative utility items in $D$;
3 Scan dataset $D$ and compute $RLU(\alpha, z)$ for item $z \in \rho$, using $UA[x]$.
4 $Secondary(\alpha) = \{z \mid z \in \rho \wedge RLU(\alpha, z) \geq min\_util\}$.
5 Let $\succ$ be the total order of $RTWU$ increasing values on $Secondary(\alpha)$
6 Scan $D$, remove item $x \notin Secondary(\alpha)$ from the transactions $T_j$ and delete empty transactions;
7 Sort all the remaining transactions in $D$ according to $\succ_T$ with positive utility items followed by negative utility items
8 Assign offset to each transaction in $D$.
9 Scan $D$, compute the $RSU(\alpha, z)$ of each item $z \in Secondary(\alpha)$, using $UA[z]$;
10 $Primary(\alpha) = \{z \mid z \in Secondary(\alpha) \wedge RSU(\alpha, z) \geq min\_util\}$;
11 $search\_P(\eta, \alpha, D, Primary(\alpha), Secondary(\alpha), min\_util)$;
12 **return** HUIs;

**Algorithm 9:** The *search_P* procedure

**Input:** $\eta$: set of negative items, $\alpha$: an itemset, $\alpha - D$: the projected dataset, $Primary(\alpha)$: the *Primary* items of $\alpha$, $Secondary(\alpha)$: the *Secondary* items of $\alpha$ and $min\_util$: thresholds.

**Output:** The set of HUIs that are extensions of $\alpha$ with positive items

1 **foreach** *each item* $z \in Primary(\alpha)$ **do**
2  $\beta = \alpha \cup \{z\}$;
3  Scan $\alpha - D$, compute $U(\beta)$ and create $\beta - D$;
4  **if** $U(\beta) \geq min\_util$ **then**
5  Output $\beta$;
6  **if** $U(\beta) > min\_util$ **then**
7  $search\_N(\eta, \beta, \beta - D, min\_util)$;
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  $search\_P(\eta, \beta, \beta - D, Primary(\beta), Secondary(\beta), min\_util)$;

transaction by the $\succ_T$ using lexicographical order. Thus transaction merging technique is performed here as suggested in [28]. Line 9 scans the dataset again to calculate the redefined sub-tree utility of each *Secondary* item for the itemset $\alpha$ using the *UA*. Line 10 finds primary itemset for the itemset $\alpha$. Lastly, line 11 performs the dept-first search by

calling the recursive procedure starting with the itemset $\alpha$.

**Algorithm 10:** The *search_N* procedure

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

**Output:** The set of HUIs that are extensions of $\alpha$ with negative items.

1 **foreach** *each item* $z \in \eta$ **do**
2  $\beta = \alpha \cup \{z\}$;
3  Scan $\alpha - D$, compute $U(\beta)$ and create $\beta - D$.
4  **if** $U(\beta) \geq min\_util$ **then**
5  Output $\beta$
6  Calculate $RSU(\beta, z)$ for all item $z \in \eta$ by scanning $\beta - D$ once, using the negative utility-array.
7  $Primary(\beta) = \{z \in \eta \mid RSU(\beta, z) \geq min\_util\}$.
8  $search\_N(Primary(\beta), \beta, \beta - D, min\_util)$

The **Algorithm 9** has six parameters; $\eta$ denotes a set of negative items, $\alpha$ denotes the current itemsets to be extended, $\alpha - D$ represents the projected dataset, *Primary* denotes primary items for $\alpha$, *Secondary* denotes secondary items for $\alpha$ and lastly $min\_util$ represents minimum utility threshold. This algorithm finds the extensions of $\alpha$ with only positive items. The algorithm *search_P* recursively calls itself every time to extend each item of $\alpha$ in the form of $\beta$ as $\beta = \alpha \cup \{z\}$, where $z$ is an item of *Primary* item of itemset $\alpha$. A single item extension technique follows to extend itemset $\alpha$. A dataset scanning is required to calculate the utility of each extension $\beta$ and then the projected dataset $\beta - D$ is constructed. Moreover, the transaction merging is performed with the projected dataset $\beta - D$ construction process. The itemsets $\beta$ whose utility is greater or equal to $min\_util$ becomes HUIs (Line 5). To extend the itemset with negative itemset, **Algorithm 10** is called when the utility of itemset is greater than $min\_util$ (Line no. 7). Otherwise, the projected dataset $\beta - D$ is scanned again to calculate the redefined sub-tree utility and redefined local-tree utility for $\beta$ of each item $z$ so that $\beta$ can be extended using *UAs* (Line 8). Thereafter, we can get the *Primary* and *Secondary* items of $\beta$. **Algorithm 10** is repeatedly executed with extension of $\beta$ using dept-first search. Furthermore, the algorithm is called until it satisfies the threshold.

**Algorithm 10** is called when the utility of items is more than $min\_util$. It extends the itemset with the negative utility only (Line no. 2). It uses the *Property 2.5.5* to prune the

search space area for negative extensions of the desired itemsets. If utility of itemset $\beta$ is not less than *min_util* then the itemset becomes HUIs (Line 4). This algorithm again calculates the *RSU* for all negative items by scanning the dataset using negative utility by calculating *UA*. Thereafter, the algorithm is recursively calls itself until all the extended items with a negative utility that fulfill *min_util* threshold are not found.

TABLE 5.2: Transaction Utility (TU)

| $T_id$ | Transaction | Purchase quantity (IU) | Utility (U) | TU |
|---|---|---|---|---|
| $T_1$ | $a,b,d,e$ | 2, 2, 1, 3 | 4, -6, 4, 3 | 5 |
| $T_2$ | $b,c,e$ | 1, 5, 1 | -3, 5, 1 | 3 |
| $T_3$ | $b,c,d,e,$ | 2, 1, 3, 2 | -6, 1, 12, 2 | 9 |
| $T_4$ | $c,d,e$ | 2, 1, 3 | 2, 4, 3 | 9 |
| $T_5$ | $a$ | 2 | 4 | 4 |
| $T_6$ | $a,b,c,d,e$ | 2, 1, 4, 2, 1 | 4, -3, 4, 8, 1 | 14 |
| $T_7$ | $b,c,e$ | 3, 2, 2 | -9, 2, 2 | -5 |

TABLE 5.3: Redefined Transaction Utility

| $T_id$ | Transaction | TU | TU (redefined) |
|---|---|---|---|
| $T_1$ | $a,b,d,e$ | 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 | 17 |
| $T_7$ | $b,c,e$ | -5 | 4 |

TABLE 5.4: *RTWU* values of items based on redefined *TU*

| Item | $a$ | $b$ | $c$ | $d$ | $e$ |
|---|---|---|---|---|---|
| RTWU | 32 | 53 | 51 | 52 | 62 |

TABLE 5.5: Final HUIs of the running example

| Itemset | Utility | Itemset | Utility |
|---|---|---|---|
| $\{a\}$ | 12 | $\{c\}$ | 14 |
| $\{a,c,d\}$ | 16 | $\{c,d\}$ | 31 |
| $\{a,c,d,b\}$ | 13 | $\{c,d,b\}$ | 16 |
| $\{a,c,d,e\}$ | 17 | $\{c,d,e\}$ | 37 |
| $\{a,c,d,e,b\}$ | 14 | $\{c,d,e,b\}$ | 19 |
| $\{a,d\}$ | 20 | $\{c,e\}$ | 23 |
| $\{a,d,b\}$ | 11 | $\{d\}$ | 28 |
| $\{a,d,e\}$ | 24 | $\{d,e\}$ | 37 |
| $\{a,d,e,b\}$ | 15 | $\{d,e,b\}$ | 15 |
| $\{a,e\}$ | 12 | $\{e\}$ | 12 |

### 5.1.5 An Illustrative Example

In this section, a simple illustrative example is given to show how the proposed algorithm can find HUIs from a transactional dataset. Let us assume that there are seven transactions in the dataset as shown in TABLE 2.8 and there are five items that appear with their internal quantity. Also, we assume that the external utility or profit value of each single item is predefined in TABLE 2.9. Moreover, $min\_util$ is set as 10. The proposed algorithm proceeds as follows to find HUIs from the transactional dataset.

The algorithm calculates the utility of each item in a transaction and finds the $TU$ of that transaction. There are three items, $b, c$ and $e$, in $T_2$ and their quantities are $1, 5$ and 1. And the external utility of $\{b\}, \{c\}$ and $\{e\}$ in TABLE 2.9 are $-3, 5$ and 1 respectively. The utility values of items $b, c$ and $e$, in $T_2$ can be calculated as $1 \times$ (-3) = -3, $5 \times 1 = 5$ and $1 \times 1 = 1$ respectively. After the above process, the $TU$ of $T_2$ can be calculated as -3 + 5 + 1, which is 3. The results for the $TU$ values of all the transactions are shown in TABLE 5.2.

To overestimate the utility, the proposed algorithm uses $RTU$ according to *Definition 6.0.1*. To find the $RTU$, proposed algorithm calculates only positive utility values as 5 + 1, which is 6 in $T_2$. Similarly, all $RTU$ can be calculated. The $RTU$ for all transactions is shown in TABLE 5.3. The $RLU$ is calculated using depth-first search which is equal to $RTWU$ as explained in Section 5.1.3. $RTWU$ values of an item $\{a\}$ in running example that appear in three transactions $T_1$, $T_5$ and $T_6$ and their $RTU$ values are 11, 4 and 17

respectively. $RTWU$ of the item $\{a\}$ can thus be calculated as 11 + 4 + 17 which is 32. $RTWU$ of each 1-item is as shown in TABLE 5.4.

TABLE 5.6: Statistical information about datasets

| 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 |
| pumsb | 49046 | 2113 | 74 | 74 | Dense |
| T40I10D100K | 100000 | 942 | 39.6 | 77 | Dense |
| BMSPOS | 515366 | 1656 | 6.51 | 164 | Sparse |
| retail | 88162 | 16470 | 10.3 | 76 | Sparse |
| T10I4D100K | 100000 | 870 | 10.1 | 29 | Sparse |
| kosarak | 990002 | 41270 | 8.09 | 2498 | Sparse (Large) |

The $RLU$ of items is not less than $min\_util$ is then we find *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 $RTWU$ (see line 5 in **Algorithm 8**). Negative items always come after positive itemset. Thereafter, the items which are not member of *Secondary* set are removed. Hence, no item is removed from the example dataset (no change in dataset). At the same time, if all the items are removed from the transactions, we remove those empty transactions. And then the remaining transactions according to total order $\succ_T$ are sorted (see line 7 in **Algorithm 8**). After that, the proposed algorithm scans the dataset again and calculates $RSU$ of all itemsets. The items whose $RSU$ is not less than $min\_util$ is in *Primary* items. Hence, the *Primary* = $\{a,c,d,e\}$. Only the items of the *Primary* set is used to explore with depth-first search. All the items of the *Secondary* $\{a,c,d,e,b\}$ set are used as descendant nodes in each sub-tree. For this, dept-first search is used to find descent nodes in sub-tree. And nodes are mined using **Algorithm 9** and **Algorithm 10**. **Algorithm 9** is then recursively called to extend all items with positive utility. Thereafter, **Algorithm 10** is called to extend the items with negative utility items. The final HUIs of the running example are shown in TABLE 5.5

## 5.2 Performance Evaluation

In this section, we compare the performance of our proposed algorithm (EHIN) with the state-of-the-art algorithm (FHN) [73]. To the best of our knowledge, FHN algorithm is

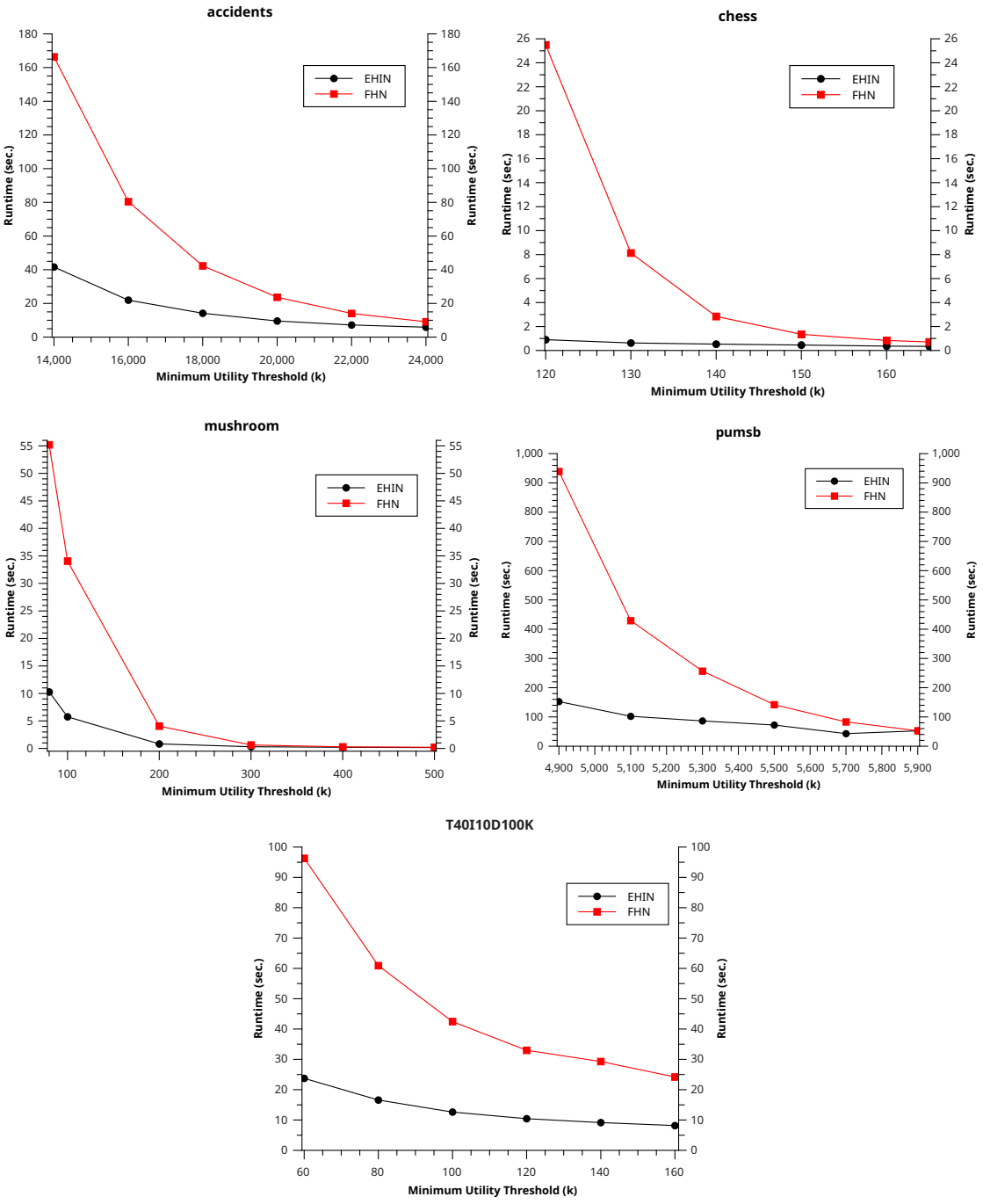


FIGURE 5.3: Execution time on dense datasets

the best algorithm for mining HUIs with negative utility. We implemented the proposed algorithm by extending the open-source java library [77]. We conducted all the experiments on 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). To ensure robustness of the

results, we ran all our experiments ten times to report the average results.

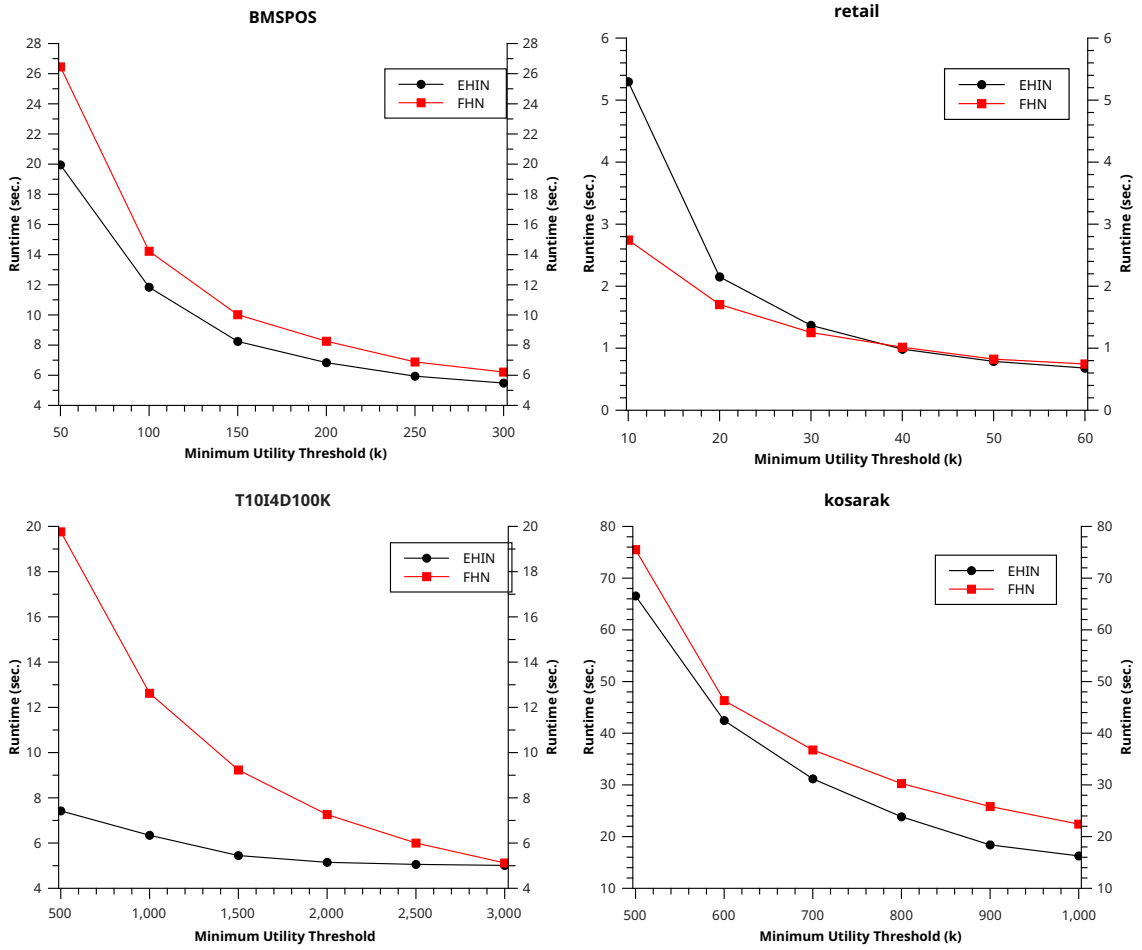


FIGURE 5.4: Execution time on sparse datasets

To analyze the performance of our algorithm with compared algorithm in different situations, we tested the algorithms with nine different datasets available from spmf [77]. Among these nine datasets, seven are real and two are synthetic. The detailed characteristics of these datasets are shown in TABLE 5.6. The experimental results on all these datasets are separately shown and are discussed in Section 5.2.1 and Section 5.2.2. The report of memory consumptions and scalability are shown respectively in Section 5.2.3 and 5.2.5.

To compare the proposed algorithm EHIN with FHN algorithm, we executed both of the algorithms on all datasets by decreasing *min_util*. *min_util* was decreased until both the algorithms took too much time or out of memory or a clear winner was observed. The experimental results on all these datasets are separately shown in the next sections.

We analyze the performance improvement by measuring the total execution runtime and total memory consumed by EHIN and FHN algorithms. For this experiment, we took the total execution runtime by EHIN algorithm as the baseline (100%). The relative runtime is calculated as the ratio of the total execution runtime of FHN algorithm to the total execution runtime of EHIN.

### 5.2.1 Experiments on Dense Datasets

The performances of both the algorithms on the dense datasets are shown in FIGURE 5.3. It is evident that EHIN algorithm outperforms FHN algorithm for all dense datasets. As we observe from FIGURE 5.3, EHIN is less sensitive to runtime for all given $min\_util$ for accident, chess, pumsb and T40I10D100K datasets. Whereas FHN algorithm decrements in $min\_util$ which causes exponential increase in runtime. However, for the mushroom dataset, the runtime of EHIN and FHN algorithms remains close to each other for $min\_util$ with $300k$ to $500k$. Although they are very close to each other, EHIN outperforms FHN algorithm for $min\_util$ with $100k$ and $200k$. The efficiency of EHIN algorithm compared to FHN algorithm is computed for all the cases of dense datasets. EHIN algorithm performs better for all dense datasets because of transaction merging and dataset projection techniques to compact the dataset. Another reason for better performance is array-based utility counting technique.

### 5.2.2 Experiments on Sparse Datasets

The performance of both the algorithms on the sparse datasets is shown in FIGURE 5.4. It is evident that EHIN algorithm outperforms FHN algorithm for all the sparse datasets except retail. As we observe from FIGURE 5.4 EHIN algorithm outperforms FHN algorithm in runtime for BMSPOS, T10I4D100K and kosarak datasets. The efficiency of EHIN is hampered while performing transaction merging and dataset projection where the dataset has a large number of distinct items. This is the reason for less efficiency of EHIN in case of retail dataset for $10k$ to $30k$ $min\_util$ threshold.

### 5.2.3 Memory Usage

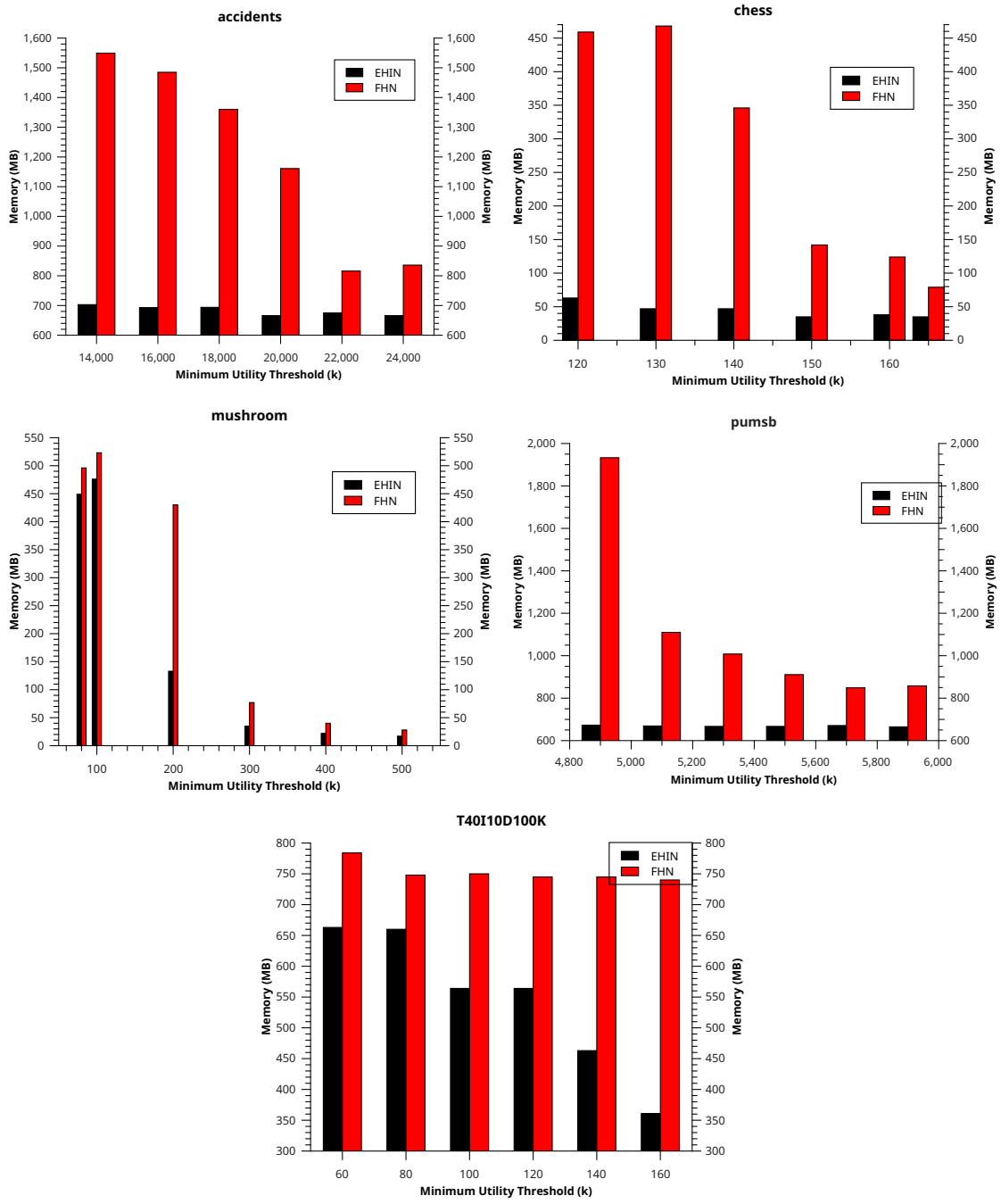


FIGURE 5.5: Memory usage on dense datasets

FIGURE 5.5 and FIGURE 5.6 show the memory consumption of the algorithms on dense and sparse datasets respectively. EHIN algorithm uses almost stable memory for accidents, kosarak and pumsb datasets with different *min_util* thresholds. For chess, mushroom and retail datasets, memory consumption increases as minimum utility

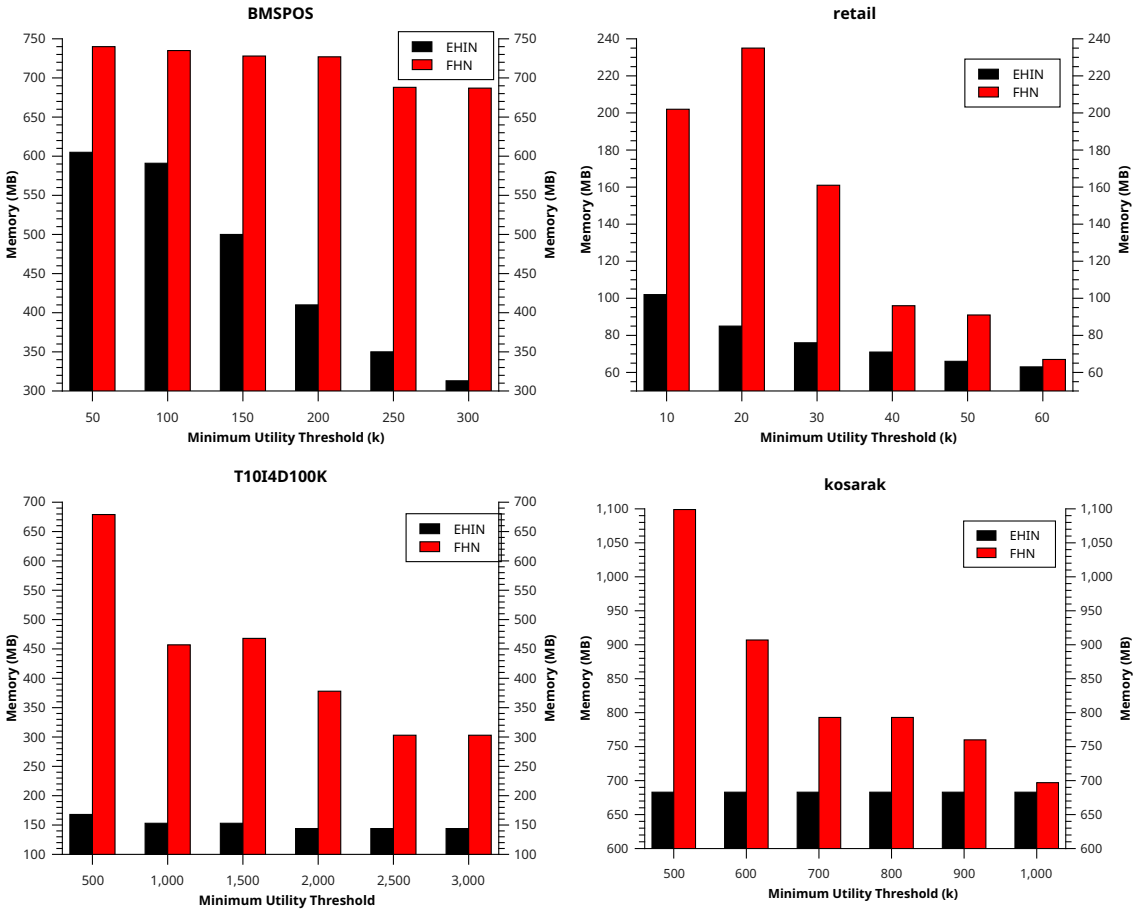


FIGURE 5.6: Memory usage on sparse datasets

threshold decreases. For all datasets except chess, EHIN algorithm uses half amount of memory. For chess dataset, EHIN algorithm uses almost six times less memory than FHN algorithm. For accidents, chess, kosarak, mushroom, retail and pumsb, EHIN algorithm uses 2.20, 9.96, 1.61, 3.23, 2.76 and 2.87 times less memory, respectively. We can see that EHIN algorithm always consumes less memory than FHN algorithm. We observe that the memory usage of EHIN algorithm increases almost linearly when the number of distinct items increases. EHIN algorithm increases slowly when the number of transactions increases.

Our proposed algorithm performs always better in memory usage for all dataset either they are sparse or dense. Although our algorithm takes long runtime in case of sparse dataset like retail, it takes less memory. For sparse datasets like retail, the trade-off which our algorithm provides is useful for the applications where memory is limited.

TABLE 5.7: Relative runtime improvement analysis on dense datasets

| **accidents** | | **chess** | | **mushroom** | | **pumsb** | | **T40I10D100K** | |
|---|---|---|---|---|---|---|---|---|---|
| $min\_util(k)$ | $(\times Times)$ | $min\_util(k)$ | $\times Times$ | $min\_util(k)$ | $\times Times$ | $min\_util(k)$ | $\times Times$ | $min\_util(k)$ | $\times Times$ |
| 14000 | 4.00 | 120 | 28.32 | 80 | 5.37 | 4900 | 5.50 | 60 | 4.06 |
| 16000 | 3.66 | 130 | 12.95 | 100 | 5.92 | 5100 | 4.20 | 80 | 3.67 |
| 18000 | 2.98 | 140 | 5.35 | 200 | 4.92 | 5300 | 2.97 | 100 | 3.36 |
| 20000 | 2.47 | 150 | 2.96 | 300 | 2.01 | 5500 | 1.96 | 120 | 3.16 |
| 22000 | 1.95 | 160 | 2.29 | 400 | 1.35 | 5700 | 1.93 | 140 | 3.2 |
| 24000 | 1.55 | 165 | 2.09 | 500 | 1.35 | 5900 | 1.01 | 160 | 2.96 |

## 5.2.4 Relative Runtime and Memory Comparison Analysis

In this section, we analyze the relative runtime and memory usage of both the algorithms. Here we present how many times the proposed algorithm is faster than the state-of-the-art algorithm FHN. For this analysis, we take the total execution runtime of EHIN algorithm as the baseline (100%). The relative runtime is calculated as the ratio of the runtime of FHN algorithm to that of EHIN algorithm. Same phenomena are applied for relative memory usage comparison.

TABLE 5.8: Relative runtime improvement analysis on sparse datasets

| **BMSPOS** | | **retail** | | **T10I4D100K** | | **kosarak** | |
|---|---|---|---|---|---|---|---|
| $min\_util(k)$ | $\times Times$ | $min\_util(k)$ | $\times Times$ | $min\_util$ | $\times Times$ | $min\_util(k)$ | $\times Times$ |
| 50 | 1.33 | 10 | 0.52 | 500 | 6.00 | 500 | 1.13 |
| 100 | 1.20 | 20 | 0.79 | 1000 | 4.00 | 600 | 1.09 |
| 150 | 1.21 | 30 | 0.92 | 1500 | 3.33 | 700 | 1.18 |
| 200 | 1.21 | 40 | 1.03 | 2000 | 3.00 | 800 | 1.27 |
| 250 | 1.16 | 50 | 1.05 | 2500 | 2.8 | 900 | 1.40 |
| 300 | 1.13 | 60 | 1.10 | 3000 | 2.67 | 1000 | 1.38 |

### 5.2.4.1 Relative runtime analysis

The relative runtime comparison is shown for dense and sparse datasets in TABLE 5.7 and TABLE 5.8 respectively. In the running example, for $min\_util$ at 14000$k$ EHIN algorithm is four times faster than FHN algorithm. TABLE 5.7 shows the relative runtime comparison of dense datasets. Here, it can be observed that when $min\_util$ decreases, EHIN algorithm becomes more faster than FHN algorithm. Hence, EHIN algorithm provides faster relative runtime which is the best on lowest $min\_util$. For sparse datasets also, EHIN algorithm is many times faster except retail dataset as shown in TABLE 5.8.

TABLE 5.9: Relative memory usage analysis on dense datasets

| **accidents** | | **chess** | | **mushroom** | | **pumsb** | | **T40I10D100K** | |
|---|---|---|---|---|---|---|---|---|---|
| $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ |
| 14000 | 2.20 | 120 | 7.29 | 80 | 1.10 | 4900 | 2.87 | 60 | 1.18 |
| 16000 | 2.14 | 130 | 9.96 | 100 | 1.10 | 5100 | 1.66 | 80 | 1.13 |
| 18000 | 1.96 | 140 | 7.36 | 200 | 3.23 | 5300 | 1.51 | 100 | 1.33 |
| 20000 | 1.74 | 150 | 4.06 | 300 | 2.20 | 5500 | 1.37 | 120 | 1.32 |
| 22000 | 1.21 | 160 | 3.26 | 400 | 1.82 | 5700 | 1.27 | 140 | 1.61 |
| 24000 | 1.26 | 165 | 2.26 | 500 | 1.65 | 5900 | 1.29 | 160 | 2.05 |

TABLE 5.10: Relative memory usage analysis on sparse datasets

| **BMSPOS** | | **retail** | | **T10I4D100K** | | **kosarak** | |
|---|---|---|---|---|---|---|---|
| $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ | $min\_util(k)$ | $\times Times(less)$ |
| 50 | 1.22 | 10 | 1.98 | 500 | 4.04 | 500 | 1.61 |
| 100 | 1.24 | 20 | 2.76 | 1000 | 2.99 | 600 | 1.33 |
| 150 | 1.46 | 30 | 2.12 | 1500 | 3.06 | 700 | 1.16 |
| 200 | 1.77 | 40 | 1.35 | 2000 | 2.62 | 800 | 1.16 |
| 250 | 1.97 | 50 | 1.38 | 2500 | 2.1 | 900 | 1.11 |
| 300 | 2.19 | 60 | 1.06 | 3000 | 2.1 | 1000 | 1.02 |

#### 5.2.4.2 Relative memory usage analysis

In this section, we analyze the relative memory usage of both the algorithms. Here, we present how much less memory is consumed by EHIN algorithm. TABLE 5.9 and TABLE 5.10 show the memory usage comparison of dense and sparse datasets respectively. It can be observed that EHIN algorithm consumes less memory for both dense and sparse datasets.

TABLE 5.11: Relative runtime analysis on best, average and minimum case

| **Dataset** | **Best** | **Average** | **Minimum** |
|---|---|---|---|
| accidents | 4 | 3.34 | 1.55 |
| chess | 28.32 | 12.22 | 2.09 |
| mushroom | 5.92 | 5.38 | 1.35 |
| pumsb | 5.5 | 3.54 | 1.01 |
| T40I10D100K | 4.06 | 3.54 | 2.96 |
| BMSPOS | 1.33 | 1.24 | 1.13 |
| retail | 1.1 | 0.74 | 0.52 |
| T10I4D100K | 2.66 | 1.74 | 1.02 |
| kosarak | 1.4 | 1.19 | 1.09 |

#### 5.2.4.3 Relative runtime analysis on best, average and minimum case

We also analyze the best, average and minimum relative runtime comparison. TABLE 5.11 shows the relative runtime comparison of EHIN algorithm. This experiment presents that EHIN algorithm is upto 28 times faster in the best case. In the worst case, EHIN

algorithm is two times slower only (for retail dataset). EHIN algorithm is good in best, average and minimum relative runtime comparison except for the retail dataset.

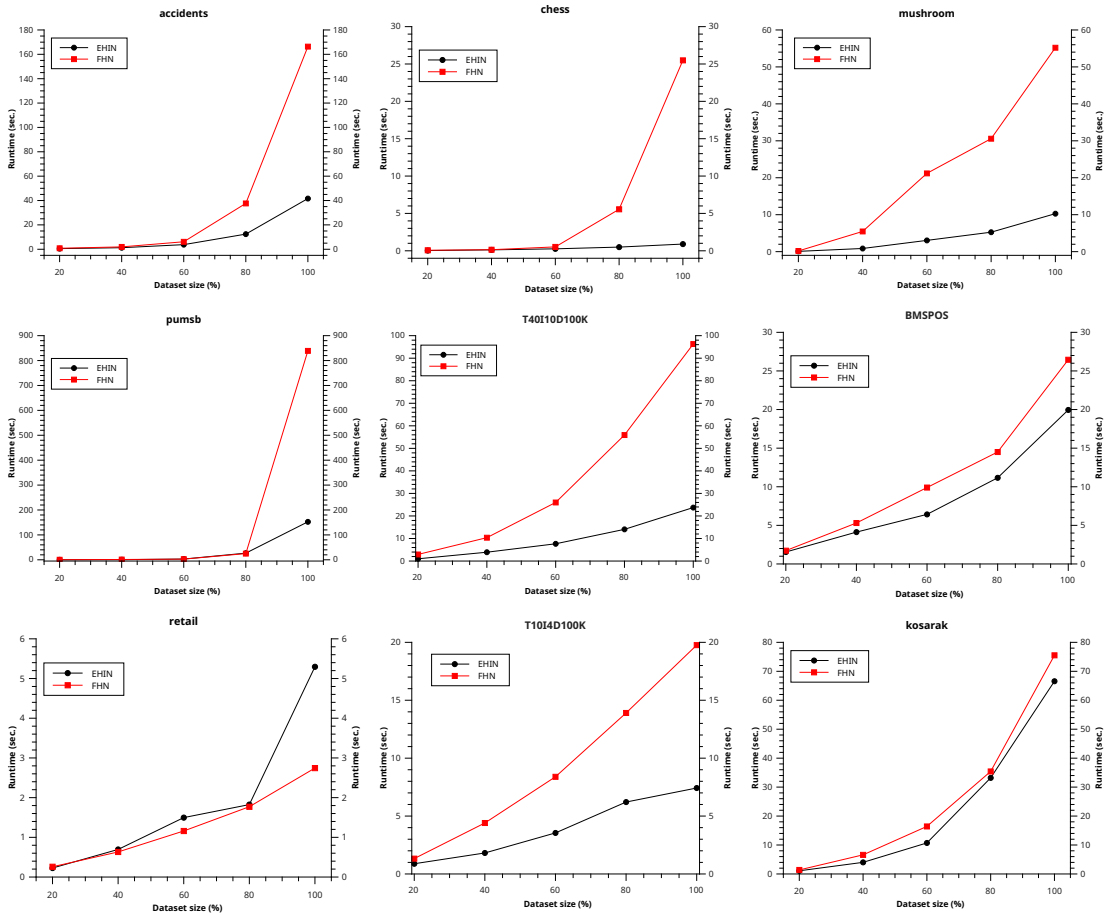


FIGURE 5.7: Scalability Runtime Comparison on various datasets

TABLE 5.12: Relative memory analysis on best, average and minimum case

| Dataset | Best | Average | Minimum |
|---|---|---|---|
| accidents | 2.2 | 1.76 | 1.21 |
| chess | 9.96 | 6.11 | 2.26 |
| mushroom | 3.23 | 1.41 | 1.1 |
| pumsb | 2.87 | 1.66 | 1.27 |
| T40I10D100K | 2.05 | 1.38 | 1.13 |
| BMSPOS | 2.19 | 1.56 | 1.22 |
| retail | 2.76 | 1.84 | 1.06 |
| T10I4D100K | 4.04 | 2.86 | 2.1 |
| kosarak | 1.61 | 1.23 | 1.02 |

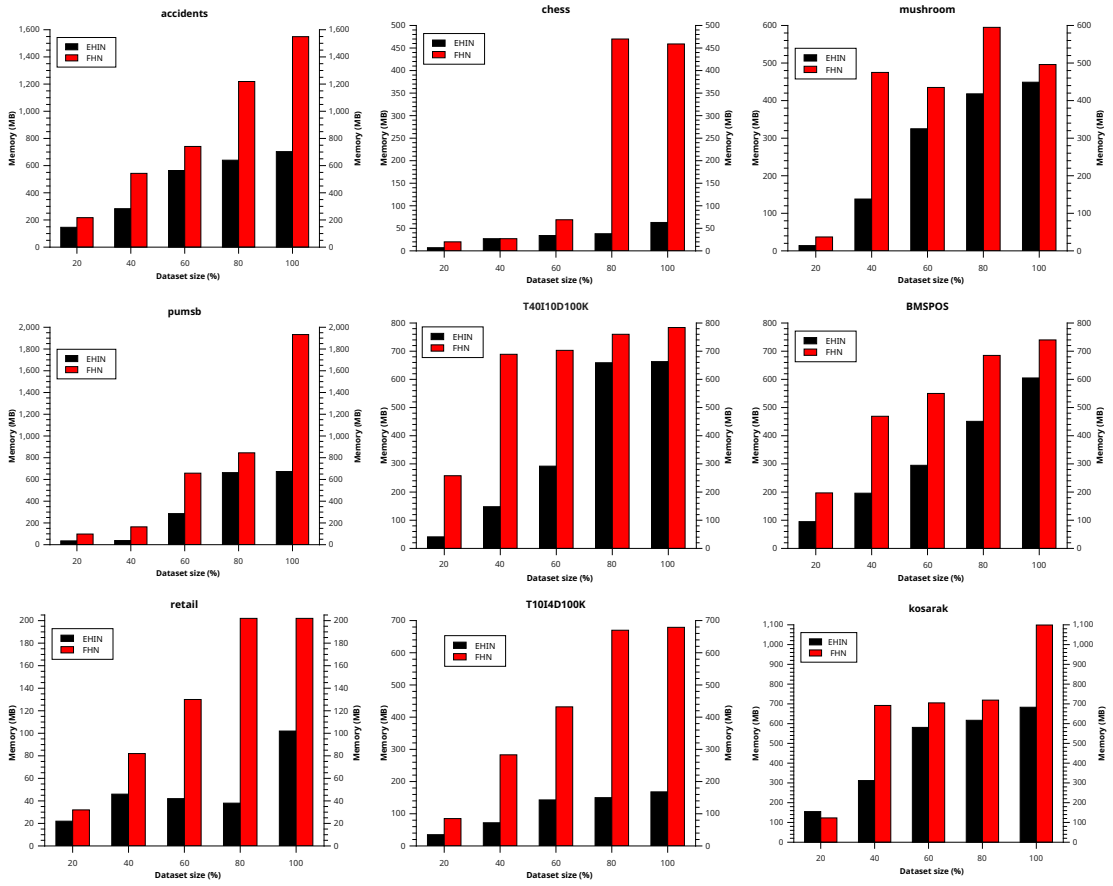


FIGURE 5.8: Scalability Memory Comparison on various datasets

#### 5.2.4.4 Relative memory usage analysis on best, average and minimum case

Here we analyze the best, average and minimum less memory usage comparison. TABLE 5.12 shows how much less memory EHIN algorithm consumes. This experiment shows that EHIN algorithm consumes up to 10 times less memory compared to FHN algorithm. EHIN algorithm consumes less memory always for dense and sparse datasets.

### 5.2.5 Scalability

In this section, we evaluate the scalability of the proposed algorithm. For the scalability comparison, *min_util* threshold is fixed to the lowest minimum threshold that is used in each dataset for runtime and memory evolution. As shown in FIGURE 5.7, the proposed

algorithm has a good scalability with the varied size of the datasets. The size of datasets is varied from 20% to 100% to evaluate the scalability results.

The runtime of both algorithms linearly increases as the dataset size increases. The difference between the runtime of both algorithms grows wider when the dataset size increases as shown in FIGURE 5.7. The memory usage of EHIN algorithm is steadily increased with the increased size of dataset. In FHN algorithm, memory usage increases rapidly as shown in FIGURE 5.8.

## 5.3 Summary

In this chapter, the issue of HUIs mining with negative utility has been addressed. Previous algorithms like HUINIV-Mine and FHN algorithms attempt this issue. Our proposed algorithm EHIN utilized and redefined two new upper bounds named redefined sub-tree and redefined local utility. We also redefine $TU$ and $TWU$ according to the negative utility. A new array-based utility counting is utilized to calculate upper bounds quickly. To reduce the dataset scanning cost in term of time and memory usage, dataset projection and transaction merging techniques are utilized. The extensive experimental evaluation on nine datasets shows that proposed algorithm greatly reduces the execution and memory requirements. We observe that EHIN algorithm outperforms the state-of-the-art algorithm FHN for both in runtime and memory aspect in all our observations. We have seen that the decrease in minimum utility threshold increases both the time and memory requirements for both algorithms. We presents an illustrative example of the proposed algorithm with the example dummy transactional dataset. The experimental results show that EHIN algorithm is 28 times faster in execution time and consumes up to 10 times less memory than FHN algorithm. Moreover, a key advantage is that EHIN algorithm always performs better for dense datasets.