Chapter 7

Closed High Utility Itemsets Mining with Negative Utility Value

In previous chapter, we have developed efficient algorithm named EHNL for mining HUIs with negative utility and length constraints. Although, EHNL remove very small and too long itemsets, it mine a large number of redundant HUIs. In order to overcome this issue, closed HUIs mining algorithms have been proposed which avoids duplicate itemsets. However, closed HUIs mining algorithms work only with positive utility value. Mining closed HUIs mining algorithm with negative utility has not yet been proposed, although negative utility mining is commonly seen in many real-world applications. To address this issue, we propose an efficient algorithm named CHN (**C**losed **H**igh utility itemsets mining with **N**egative utility).

Definition 7.0.1. (Closed high utility itemsets). An itemset X is called a closed HUIss if it is HUIs and there does not exist any HUIs Y such that $X \subset Y$ and $support(X) = support(Y)$.

For example, we utilize transactional datasets presented in TABLE 2.8 and TABLE 2.9 in Chapter 2 for understand the definitions and strategies. CHUIs for the example is shown in TABLE 7.2 where min_util is 15.

Problem statement. We determine all the non-redundant itemsets containing negative utility item whose utility is not less than user-defined min_util threshold.

7.1 The Proposed Algorithm

In this section, we give a step-by-step analysis of the proposed algorithm. Section 7.1.1 describes the dataset cost reduction techniques to reduce the dataset scanning. Section 7.1.2 describes the pruning strategies. Section 7.1.3 introduces array-based utility counting technique. Section 7.1.4 describes the CHUIs mining strategies. Finally, Section 7.1.5 gives the pseudo-code and explanation of the proposed algorithm.

7.1.1 Efficient Dataset Scanning Techniques

The proposed algorithm reduces the dataset scanning costs by reducing the dataset size using dataset projection and transaction merging techniques. The projected dataset is scanned only once for merge the identical transactions.

Definition 7.1.1. (Transaction merging). Let the identical transactions as T_1, T_2, T_3, T_n replace by a new transaction T_M= T_1= T_2 = T_3 = T_n (Identical transactions may not contain the same quantity values of each item). And quantity of each item in these identical transactions is $x \in T_M$ is defined as $IU(x, T_M) = \sum_{i=1,\ldots,n} IU(x, T_i)$.

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

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

Definition 7.1.2. (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 $\alpha - D$ = $\{\alpha - T \mid T \in D \wedge \alpha - T \neq \emptyset\}$.

Definition 7.1.3. (Projected transaction merging). Let the identical transactions as T_1, T_2, T_3, T_n in the projected dataset $\alpha - D$ is replace by a new transaction T_M= T_1= T_2 = T_3 = T_n, And quantity of these identical transactions $x \in T_M$ is defined as $IU(x, T_M) = \sum_{i=1,\ldots,n} IU(x, T_i)$.

The projected dataset $\alpha - D$ for $\alpha = \{D\}$ contains the identical projected transactions as $\alpha - T_1 = \{E,B\}$, $\alpha - T_3 = \{E,B\}$, $\alpha - T_4 = \{E\}$ and $\alpha - T_6 = \{E,B\}$ is further merged by a new projected transaction using offset pointers in the original dataset. Transactions $\alpha - T_1$, $\alpha - T_3$ and $\alpha - T_6$ can be replaced by a new transaction $T_{1\_3\_6} = \{E,B\}$ where $IU(E,T_{1\_3\_6}) = 6$ and $IU(B,T_{1\_3\_6}) = 5$. The example also shows, when as we mine long itemsets the $\alpha - D$ become shorter and shorter.

Transaction merging technique is desirable to reduce the size of the dataset. The main problem to implement this technique is that to identify the identical transactions. To achieve this, we need to compare all transactions with each other. But this technique to compare all the transactions to each is not an efficient technique. To overcome this problem, we follow the same transaction sorted technique $\succ_T$ proposed in [28]. This sorting technique is not computationally expensive and perform only once.

Definition 7.1.4. (Total order on the transactions). For the dataset D, the total order $\succ_T$ is defined as lexicographical order when the transactions are being read backward. For more details of total order $\succ_T$ on the transactions, we can see in [28].

We sort the original dataset according to a new total order $\succ_T$ on the transactions. Sorting of transactions has been performed in lexicographical order 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.

7.1.2 Pruning Non-HUIs

We have so far introduced one novel tighter upper-bound $RTWU$ which considers negative utility for reducing the search space. Now we are going to introduce a new strategies to prune non-HUIs. This strategy are more efficient and much tighter upper-bound on the utility of itemsets. This strategy are calculated the utility value when the tree is traversed in dept-first search manner.

7.1.2.1 Prune search space using Redefined Sub-tree Utility

Definition 7.1.5. (Redefined Sub-tree Utility). For an itemset α and an item $x \in E(\alpha)$, that can be extended α to follow the depth-first search into the sub-tree. The *RSU* of the item x, if α is $RSU(\alpha, x) = \sum_{T_j \in (\alpha \cup \{x\})} [U(\alpha, T_j) + U(x, T_j) + \sum_{i \in T_j \wedge i \in E(\alpha \cup \{x\})} U(i, T)]$.

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

Property 7.1.1. (*RSU* Overestimation). For an itemset α and an item $x \in E(\alpha)$. The utility value of $RSU(\alpha, x) \geq U(\alpha \cup \{x\})$ and accordingly, $RSU(\alpha, x) \geq U(x)$ keeps the extension x of $\alpha \cup \{x\}$. The proof of sub-tree utility is presented in [28].

The relationships between the redefined upper-bounds (*RTWU* and *RSU*) and the state-of-the-art upper bound *REU* are following.

REU upper-bound is presented in [24, 27] and works with utility-list data structure. The proposed *RSU* is an redefined upper-bound of *SU*. *SU* is basically proposed in [28] which works only for positive utility items. *SU* calculates the utility of itemset by dept-first search in tree. Similarly the redefined upper-bound *RSU* calculates the utility at itemset α by dept-first search rather than at the child itemset of α. FIGURE 5.2 in Chapter 5 shows the difference between the proposed *RSU* upper-bound and *REU*. The figure shows that if an itemset α with an item x has less utility than *min_util* then the itemset with their child is pruned for the *RSU* upper-bound. And in *REU* upper-bound, if the itemset α with an item x has less utility than *min_util* then the only child nodes are pruned as shown in FIGURE 5.2. The relationship between the *RWTU* and *TWU* is already explained by the *Property 2.5.4*.

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

Definition 7.1.6. (*Primary* and *Secondary* items). For an itemset α. The *Primary* items of itemset α is the set of items, $Primary(\alpha) = \{x \mid x \in E(\alpha) \wedge RSU(\alpha, x) \geq min\_util\}$. The itemset α is the set of items $Secondary(\alpha) = \{x \mid x \in E(\alpha) \wedge RTWU(\alpha, x) \geq min\_util\}$. $RTWU(\alpha, x) \geq RSU(\alpha, x)$, so $Primary(\alpha) \subseteq Secondary(\alpha)$.

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

FIGURE 7.1: Calculate $RTWU$ using utility-array

7.1.3 Calculate Upper Bounds using Utility Array

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

Definition 7.1.7. (Utility Array) For the set of items I appear in a dataset D. The UA is an array of length $|I|$ that have an entry denoted as $UA[x]$ for each item $x \in I$. Each entry is called UA that is used to store a utility value.

7.1.3.1 Calculating $RTWU$ of all items using UA

UA is initialized to 0. Then, the $UA[x]$ for each item $x \in T_j$ is calculated $UA[x] = UA[x] + RTU(T_j)$ for each transaction T_j in the dataset D. After the dataset scanning the $UA[x]$ contains $RTWU(x)$ where each item $k \in I$.

For example, $RTWU$ of the sample transactional dataset is shown in FIGURE 7.1. The length of UA is set equal to the number of items in the transactional dataset. $RTWU$ calculation process initially sets the UA with the zeros as shown in the Step 1 in FIGURE 7.1. Step 2 reads the transaction T_1 and updates the UA with RTU. Transaction T_1 has the items A, B, D and E, hence, the only respective position of UA is updated with RTU value 11. Step 3 reads the transaction T_2 and updates the UA with RTU value 6. Transaction T_2 has the items B, C and E and hence the respective positives in UA are updated with the RTU value 6 as shown the step 3 in FIGURE 7.1. Similarly all the transactions read and update the UA and finally we find $RTWU$ value for each item.

7.1.3.2 Calculating $RSU(\alpha)$

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

7.1.3.3 Calculating the $support(\alpha)$:

UA is initialized to 0. Then, the $UA[x]$ for each item $x \in T_j$ is calculated $UA[x]$ = $UA[x]$ + 1, if item $x \in T_j \cap E(\alpha)$. The support an itemset α is the number of transactions T_j containing X in the dataset D.

7.1.4 Closed HUIs Mining Strategies

The main problem in CHUIs mining is how to check if an itemset is closed or not. To solve this problem, we utilize bi-direction extension techniques presented in BIDE algorithm [81] and incorporate these techniques into HUIs mining. BIDE is an algorithm for mining closed itemset for frequent sequence mining. The utilized and modified bi-direction extension techniques are based on forward and backward-extension checking.

Forward and backward-extension checking is used to extend an itemset if it is non-closed. An itemset X is non-closed if there must exist at least one item x in X that can be used to extend itemset X to get a new itemset X' having the same support.

Definition 7.1.8. (Forward-extensions). For an itemset $\beta = \alpha \cup \{i\}$. The itemset β has a forward-extension if there exists an item $y \succ i$ where $y \in E(\beta)$ and $support(\beta) = support(\beta \cup y)$.

In this extension checking, item y occurs after an item i, we call item y a forward-extension for an itemset β. Therefore itemset β is non-closed. For example, item x' occurs after item x, we call x' a forward-extension item.

Definition 7.1.9. (Backward-extensions). For an itemset $\beta = \alpha \cup \{i\}$. The itemset β has a backward-extension if there exists an item $y \prec i$ where $y \notin \beta$ and $support(\beta) = support(\beta \cup y)$.

In this extension checking, item y occurs before item i, we call item y a backward-extension for an itemset β. Therefore itemset β is non-closed itemset. For example, item x' occurs before item x, we call x' a backward-extension item.

Property 7.1.2. (Bi-directional extension closure checking). An itemset $\beta = \alpha \cup \{i\}$ is closed HUIs if it is HUIs and if there exists neither forward-extension item nor backward-extension item for itemset β. Otherwise, β must be non-closed itemset.

Rationale. An itemset is non-closed if it has either forward-extension or backward-extension item as given in *Definition 7.1.8* and *7.1.9*.

Definition 7.1.8, *7.1.9* and *Property 7.1.2* help us to prune the search space also.

Property 7.1.3. (Forward-extension pruning). For an itemset $\beta = \alpha \cup \{i\}$, if an itemset β has an forward-extension then whole subtree of an itemset β is pruned during dept-first search.

Property 7.1.4. (Backward-extension pruning). For an itemset $\beta = \alpha \cup \{i\}$, if an itemset β has an backward-extension then whole subtree of an itemset β is pruned during dept-first search.

Forward and backward-extension techniques are very powerful and allow us to go directly from an itemset β to its closure and prune the rest of its sub-tree during dept-fist search.

Forward and backward-extension techniques have two important properties.

Property 7.1.5. (Forward-extension, effect on utility) If an item Y is added to the left side of an itemset, the utility of the extended itemset can only be lower or equal to the utility.

Proof: The items in the transactions are sorted according to the total order $\succ$ as explained in *Definition 6.0.3*. Hence the items on the left side always have less or equal $RTWU$ value. Therefore, the extended itemset has less or equal utility value.

Property 7.1.6. (Backward-extension, effect on utility) If an item Y is added to the right side of an itemset, the utility of the extended itemset can only be higher or equal to the utility.

Proof: The items in the transactions are sorted according to the total order $\succ$. Hence, items on the right side always have higher or equal $RTWU$ value. Therefore, the extended itemset has higher or equal utility value.

Algorithm 14: CHN algorithm

Input: D: a tranasctional dataset, min_util : minimum utility threshold specified by user.
Output: Closed High utility itemsets (CHUIs)

1 $\alpha \leftarrow \emptyset$.
2 $\rho \leftarrow$ set of positive utility items in D.
3 $\eta \leftarrow$ set of negative utility items in D.
4 Scan dataset D and calculate $RTWU(x)$ for all items $x \in I$ using $UA[x]$.
5 $Secondary(\alpha) = \{x | x \in I \wedge RTWU(\alpha, x) \geq min\_util\}$.
6 Sort $Secondary(\alpha)$ according to the total order $\succ$ where positive utility items followed by negative utility items.
7 Remove items $\notin Secondary(\alpha)$ from transactions and sort items in each transaction.
8 Remove all the empty transactions from the dataset D.
9 Sort all the remaining transactions in the dataset D according to $\succ_T$.
10 Scan D, compute $RSU(\alpha, x)$ of each item $x \in Secondary(\alpha)$ using UA
11 $Primary(\alpha) = \{x \mid x \in Secondary(\alpha) \mid RSU(\alpha, x) \geq min\_util\}$.
12 Assign offsets to negative items in each transaction in D.
13 *search_pos*(α, D, $Primary(\alpha)$, $Secondary(\alpha)$, min_util).
14 **return** CHUIs.

7.1.5 CHN Algorithm

This section presents a novel algorithm for mining CHUIs with negative utility named CHN. It utilizes several novel ideas explained in the previous sections. **Algorithm 14** takes a transactional dataset D and user-defined threshold as parameters. It outputs the set of CHUIs. Line 1 takes the itemset α as an empty set. Line 2 and line 3 initialize the ρ and η with positive and negative items respectively. Line 4 scans the dataset and calculates $RTWU$ for all the items using utility array as shown in FIGURE 7.1. Line 5 generates the *Secondary* items using *Definition 7.1.6*. Line 6 sorts the items of *Secondary* according to the total order $\succ$ as defined in *Definition 6.0.3*. Line 7 removes the items which are not the member of the set *Secondary* because these deleted items cannot be member of HUIs as described in *Property 2.5.3*. Line 8 removes all the empty transactions. Line 9 sorts all the remaining transactions according to the $\succ_T$ as defined in *Definition 7.1.4*. Line

Algorithm 15: The *search_pos* procedure

Input: α: an itemset, $\alpha - D$: projected dataset, $Primary(\alpha)$: *Primary* items of α, $Secondary(\alpha)$: *Secondary* items of α and *min_util*: thresholds

Output: The set of HUIs that are extensions of α with positive items.

1 **foreach** *item* $i \in Primary(\alpha)$ **do**
2 $\beta = \alpha \cup \{i\}$; ▷ Single-item extension of α.
3 Scan $\alpha - D$, create $\beta - D$ and compute $U(\beta)$,
4 **if** β *has no backward extension* **then**
5 Calculate $support(\beta, y)$, $RSU(\beta, y)$ and $RTWU(\beta, y)$ $\forall$ item $y \in Secondary(\alpha)$ by scanning $\beta - D$.
6 **if** $support(\beta) == support(\beta \cup y)$ $\forall y \succ x \wedge y \in E(\alpha)$ *and* $RSU(\beta) > min\_util$ **then**
7 *prom_neg*(β, $\beta - D$, $support(\beta)$, *min_util*).
8 **else**
9 $Primary(\beta) = \{y \in Secondary(\alpha) \mid RSU(\beta, y) \geq min\_util\}$.
10 $Secondary(\beta) = \{y | y \in Secondary(\alpha) \mid RTWU(\beta, y) \geq min\_util\}$.
11 *search_pos*(β, $\beta - D$, $Primary(\beta)$, $Secondary(\beta)$, *min_util*).
12 **if** β *has no forward extension and* $RSU(\beta, y) > min\_util$ **then**
13 *prom_neg*(β, $\beta - D$, $support(\beta)$, *min_util*).

Algorithm 16: The *prom_neg* procedure

Input: η: set of promising negative items, β: an itemset, $\beta - D$: projected dataset and *min_util*: thresholds

Output: The set of HUIs that are extensions of β with negative items.

1 $prom\_neg = \{i \mid i \in \eta \wedge RSU(\beta \cup i) \geq min\_util\}$.
2 **if** $RSU(\beta, y) > min\_util$ *and* $support(\beta) == support(\beta \cup y)$ $\forall y \succ i \wedge x \in prom\_neg$ **then**
3 Output($\beta \cup y$).
4 **else**
5 *search_neg*(β, $\beta - D$, $Secondary(\beta)$);
6 **if** β *has no forward extension and* $RSU(\beta, y) > min\_util$ **then**
7 Output(β).

10 scans the dataset again and finds the *RSU* for each member items of *Secondary* set using *Definition 7.1.5*. Line 11 finds the *Primary* items using *Definition 7.1.6*. Line 12 assigns the offset pointers to the first negative item in each transaction. The offset pointer uses as the link from the positive items to the negative items. Line 13 calls the recursive procedure **Algorithm 15** (*search_pos*) to extend the itemset α with the positive items by performing dept-first search. The algorithm terminates and finds all the CHUIs (line 14).

Algorithm 17: The *search_neg* procedure

Input: *prom_neg*: set of promising negative items, β: an itemset, $\beta - D$: projected dataset and *min_util*: thresholds

Output: The set of HUIs that are extensions of β with negative items.

```
1  foreach item i ∈ prom_neg do
2      γ = β ∪ {i};                                   ▷ Single-item extension of β.
3      Scan β − D, create γ − D and compute U(γ).
4      Calculate support(γ,x), RSU(γ,x) ∀ item x ∈ prom_neg by scanning γ − D.
5      new_prom_neg = {i | i ∈ prom_neg ∧ RSU(γ∪i) ≥ min_util }.
6      if support(γ) == support(γ∪x) ∀x ≻ i ∧ x ∈ new_prom_neg then
7          Output(γ∪x);
8      else
9          search_neg(γ, γ − D, Secondary(γ));
10         if γ has no forward extension and RSU(γ,y) > min_util then
11             Output(γ);
```

Algorithm 15 performs single positive item extension for the itemset α by calling itself recursively. It calls a **foreach** loop to perform single-item extension (line 1). Line 2 performs the single-item extensions of α as $\beta = \alpha \cup \{i\}$. Line 3 scans the dataset $\alpha - D$ and creates the projected dataset $\beta - D$ at the same time and thus utility of itemset β is calculated. Line 4 checks for the backward extension of itemset β. If a backward extension is not found, algorithm returns to **Algorithm 16**, otherwise algorithm proceeds further. Line 5 calculates the support, *RSU* and *RTWU* of each member item of set *Secondary* with the itemset β. If all the items that can extend β have the same support as β, therefore optimization is performed to directly send $\beta \cup Secondary(\alpha)$ items to **Algorithm 16** (lines 6–7). Otherwise, **else** is executed (line 8). Line 9 and 10 calculate the *Primary* and *Secondary* items respectively. Line 11 calls the procedure **Algorithm 15** recursively with the itemset β. After returning from this recursive procedure, forward extension of β is checked and if no such extension is found, the procedure **Algorithm 16** is called (lines 12–13).

Algorithm 16 takes inputs as the current itemset β, the projected dataset $\beta - D$ and *min_util*. This algorithm first calculates list of promising negative items *prom_neg* by comparing utility of $\beta \cup \{i\}$ with *min_util* threshold (line 1). If forward-extension property is followed, the current itemset along with all *prom_neg* items is output as HUI and the algorithm returns to calling procedure (line 2–3). Otherwise, recursive procedure

Algorithm 17 is called (line 5). Line 6 checks the forward extension of the itemset β, if forward extension is not possible then only β is output as HUI (line 7).

Algorithm 17 performs single negative item extension for the input itemset. It takes input as the *prom_neg* (set of promising negative items), itemset β, projected dataset $\beta - D$ and min_util. It calls **foreach** loop and performs single negative item extension (line 1). Line 2 initializes the itemset γ with the itemset β and single negative item. The negative item is a member of *prom_neg* set. Line 3 scans the input projected dataset $\beta - D$ and creates the projected dataset for $\gamma - D$ and thus calculates the utility of itemset γ. Line 4 calculates the support and RSU of itemsets $(\gamma \cup x)$. Line 5 finds a new set of promising negative items (*new_prom_neg*). Line 6 checks the support of itemset γ with support of $(\gamma \cup x)$. If the support of these sets is equal, then the itemset $(\gamma \cup x)$ is an output set. Otherwise, line 8 executes the **else** block. Line 9 calls **Algorithm 17** (*search_neg*) recursively. Line 10 checks the forward extension of the itemset β, if forward extension is not possible then only γ is output as HUI (line 11).

7.1.6 An Illustrative example

In this section, an example is given to process the proposed closed HUIs mining algorithm. The example dataset and external utility are shown in TABLE 2.8 and TABLE 2.9 respectively. Assume min_util threshold is 15. The procedure firstly calculates RTU for the example dataset using *Definition 6.0.1*. TABLE 2.10 shows TU and RTU of each item. RTU is always greater than equal to TU. Then $RTWU$ of each item is calculated using *Definition 6.0.2*. TABLE 2.11 shows $RTWU$ values of the running example. $RTWU$ is also always greater than equal to TWU. Hence, $RTWU$ is used as upper bound. The procedure then utilizes $RTWU$ based *Property 2.5.3* to prune the unpromising candidate itemsets. The procedure also performs dataset scanning techniques which are discussed in Section 7.1.1.

After removing the unpromising candidate items, the procedure sorts the items of the transaction using *Definition 6.0.3* and also sorts the transaction of the dataset using *Definition 7.1.4*. TABLE 2.12 shows the sorting items according to the total order $\succ$. Then the procedure finds *Primary* and *Secondary* items using *Definition 7.1.6*. *Primary* and *Secondary* items for the running example dataset are $\{A, C, D\}$ and $\{A, C, D, E, B\}$

TABLE 7.1: HUIs of the running example

Itemset	Utility	Itemset	Utility
{A, C, D}	16	{C, D, E}	37
{A, C, D, E}	17	{C, D, E, B}	19
{A, D}	20	{C, E}	23
{A, D, E}	24	{D}	28
{A, D, E, B}	15	{D, E}	37
{C, D}	31	{D, E, B}	15
{C, D, B}	16	–	–

TABLE 7.2: CHUIs of the running example

Itemset	Utility	Itemset	Utility
{A, C, D, E}	17	{C, D, E}	37
{A, D, E, B}	15	{C, E}	23
{C, D, E, B}	19	{D, E }	37

respectively. Then offset to the negative items are assigned eg., item B. Then, **Algorithm 15**, **16** and **17** are called to extend itemsets α with positive and negative items. Finally, procedure finds CHUIs. Final HUIs and CHUIs of the running example are shown in TABLE 7.1 and TABLE 7.2 respectively. Number of rules of CHUIs is always less than HUIs as depicted in TABLE 7.1 and TABLE 7.2. HUIs have superset with same support are not CHUIs. Hence, CHUIs can have only non-redundant HUIs.

TABLE 7.3: 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)

7.2 Experimental Results

In this section, we check the performance of our proposed algorithm (CHN). We implemented the proposed algorithm by extending the open-source java library [77]. Experiments were performed on a PC with an Intel Core-i7-6700 machine, 3.40 GHz CPU with 8 GB of memory, running on Windows 10 Pro (64-bit Operating System). The experimental datasets used for the experimentation were downloaded from [77]. Some statistical information regarding these data sets is given in TABLE 7.3. To ensure the robustness of the results, we ran all our experiments ten times and reported the average results.

In order to evaluate the influence of the design techniques in CHN, we check the performance of two versions of CHN named CHN(RSU-Prune) and CHN(TM). CHN utilizes both the dataset reduction technique TM and pruning strategy RSU. CHN(RSU-Prune) utilizes only pruning strategy RSU where TM is disabled for this version. Similarly, CHN(TM) utilizes only dataset reduction technique TM where pruning strategy RSU is disabled for this version. All versions utilize $RTWU$ based pruning strategy.

We compare the performance of CHN and its versions with FHN [30], which is to our best knowledge, the state-of-the-art method for HUIs mining with negative utility value. In literature, no algorithm is present that mine closed HUIs with negative utility. Although CHN and FHN produce different results, they mine HUIs from the datasets having negative utility items. HUINIV-Mine is also an algorithm that produces HUIs mining with negative utility value. However, the execution time of HUINIV-Mine cannot be drawn in figures since it needs too much more execution time and memory.

To evaluate the performance, we executed all the version on all the datasets by decreasing min_util threshold. For the experiment, we decrease min_util threshold until all the versions take too much time or out of memory. The experimental results on dense and sparse datasets with all the versions of the proposed algorithm are shown in the next section.

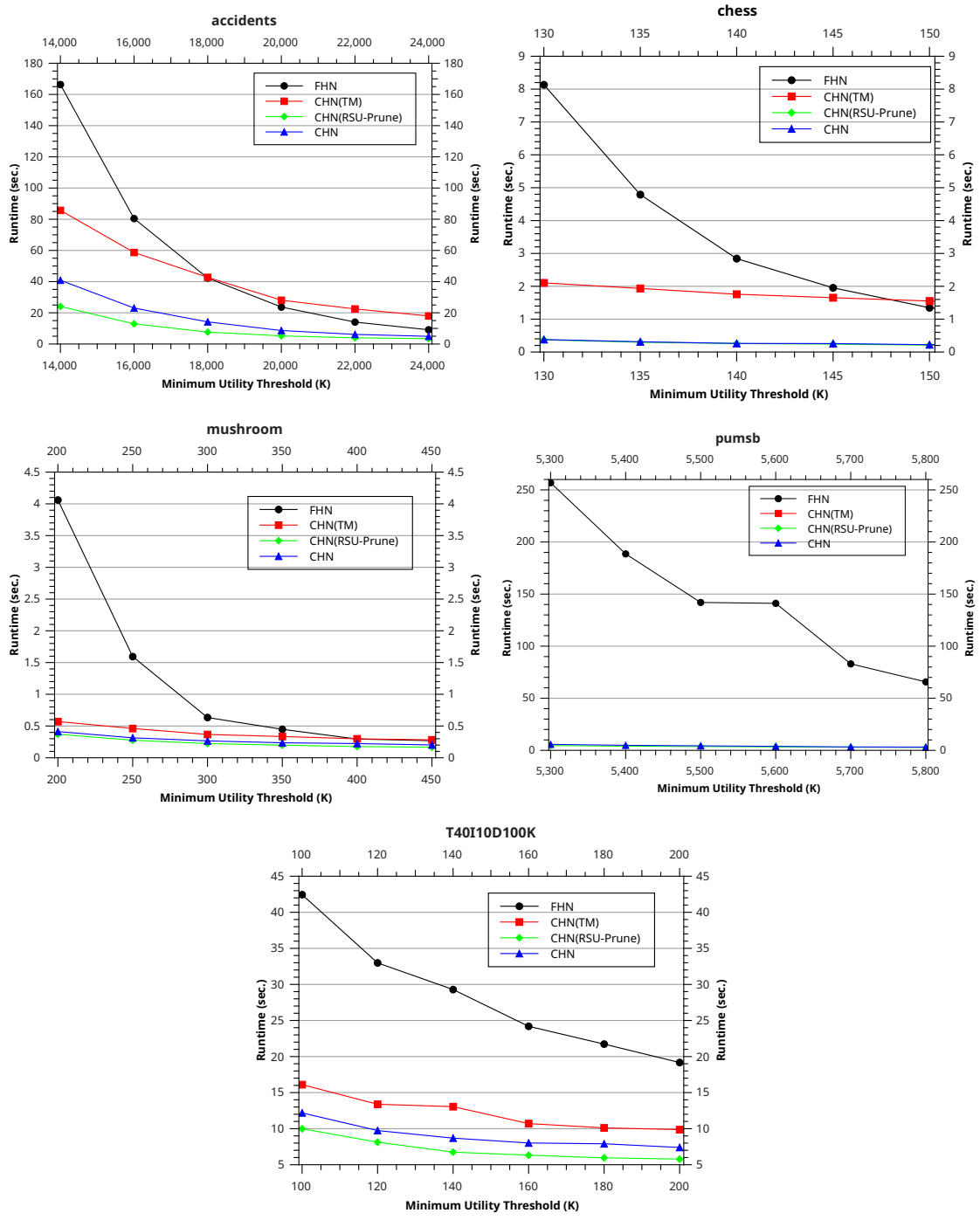

FIGURE 7.2: Execution time on dense datasets

7.2.1 Runtime Performance on Dense Datasets

We evaluate the runtime performance of all the versions of CHN and FHN on dense datasets. Five dense datasets are used for runtime comparison. The performance of the

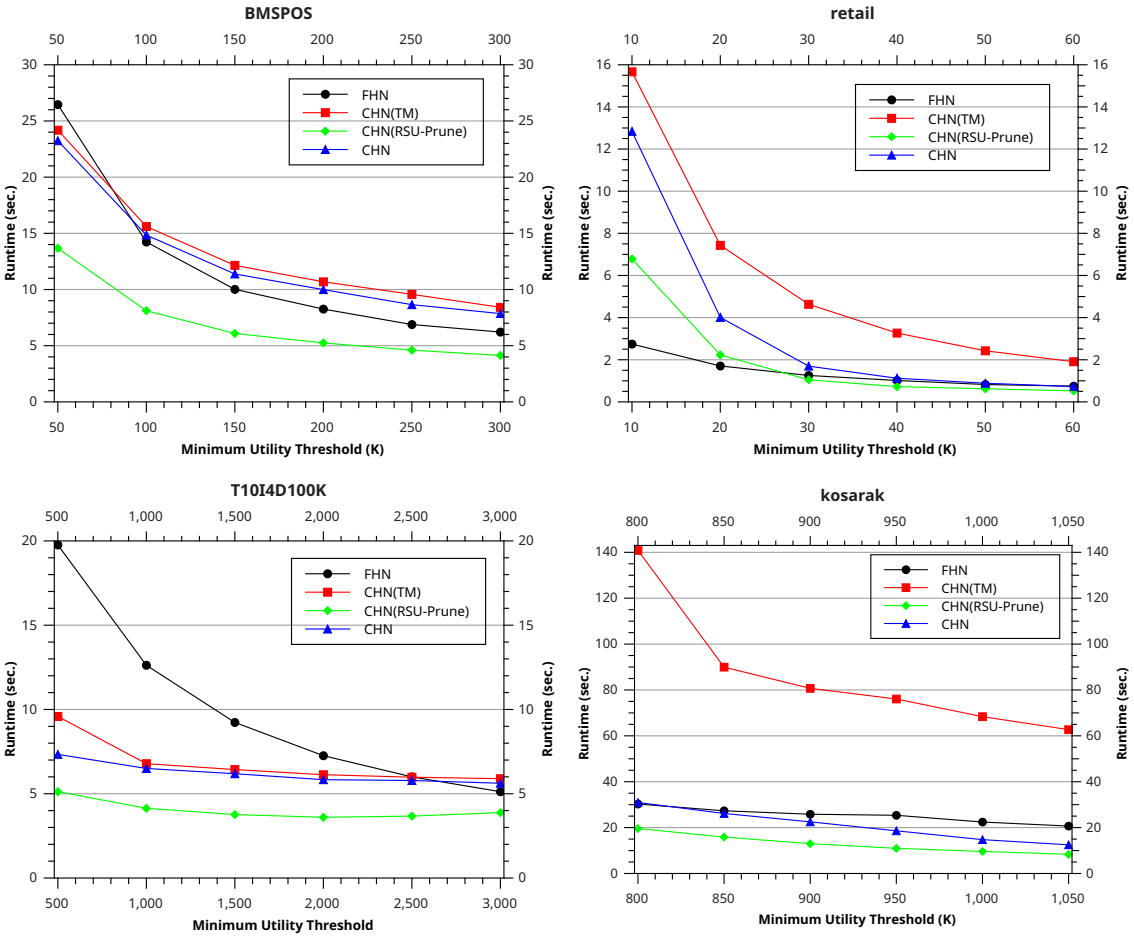

FIGURE 7.3: Execution time on sparse datasets

algorithms on all dense dataset is shown in FIGURE 7.2. For accidents and T40I10D100K datasets, CHN(RSU-Prune) algorithm outperforms all other versions and FHN which indicates that CHN and CHN(TM) requires more computations (for transaction merging) than CHN(RSU-Prune). CHN(RSU-Prune) consistently requires less runtime than all other algorithms. CHN(TM) always takes more runtime than CHN and CHN(RSU-Prune) because of lack of pruning strategy (*RSU*). However, CHN(TM) performs better than FHN. The runtime performance shows that proposed pruning strategy (RSU-Prune) plays an important role in closed HUIs mining with negative utility. The runtime of CHN and CHN(RSU-Prune) is almost similar for chess, mushroom and pumsb dataset. For pumsb dataset, the execution time of CHN(TM) cannot be drawn in FIGURE 7.2 since it needs more execution time. Transaction merging does not perform well for datasets having large number of distinct items and large average length of items such as pumsb dataset. FHN always take more runtime

compare to the proposed algorithm except CHN(TM) for pumsb dataset. When *min_util* is set lower, the gap between the proposed algorithms and FHN become larger which indicates that proposed algorithms can run for lower *min_util* threshold than FHN. FHN not performs well because it join utility-lists of smaller itemsets for generating larger itemsets. FHN considers itemsets which do not appear in the dataset, as they explore the search space of itemsets by combining smaller itemsets, without scanning the dataset. The proposed algorithms performs better because dense datasets contain lots of long items and transactions. Transaction merging perform well in dense datasets.

7.2.2 Runtime Performance on Sparse Datasets

In this section, we evaluate the runtime performance of all the versions of CHN and FHN on sparse datasets. Here, four sparse datasets are used for runtime comparison. The performance of the algorithms on all sparse dataset is shown in FIGURE 7.3. CHN(RSU-Prune) always performs well for all the dataset except retail. FHN performs well than CHN and CHN(TM) for BMSPOS dataset. However, when *min_util* threshold decreases the runtime of FHN increases rapidly and performs bad than proposed algorithms. Contradictorily, for retail dataset CHN(RSU-Prune) performs well than FHN, but when *min_util* threshold decreases the runtime of CHN(RSU-Prune) increases rapidly as shown in FIGURE 7.3. The proposed algorithms not performs well for retail dataset because retail has large number of distinct items. The proposed techniques such as transaction merging not perform well for highly sparse datasets have large number of distinct items because transaction merging takes much time to merge the transactions. These type of datasets do not have identical transactions such as retail and kosarak. For T10I4D100K dataset, all the versions of CHN performs well than FHN for all *min_util* thresholds. For kosarak dataset, CHN and CHN(RSU-Prune) performs well than FHN. CHN(TM) does not performs well for the datasets having large number of distinct such as retail and kosarak. Transaction merging based algorithms do not performs well for sparse datasets which contain lots of short items.

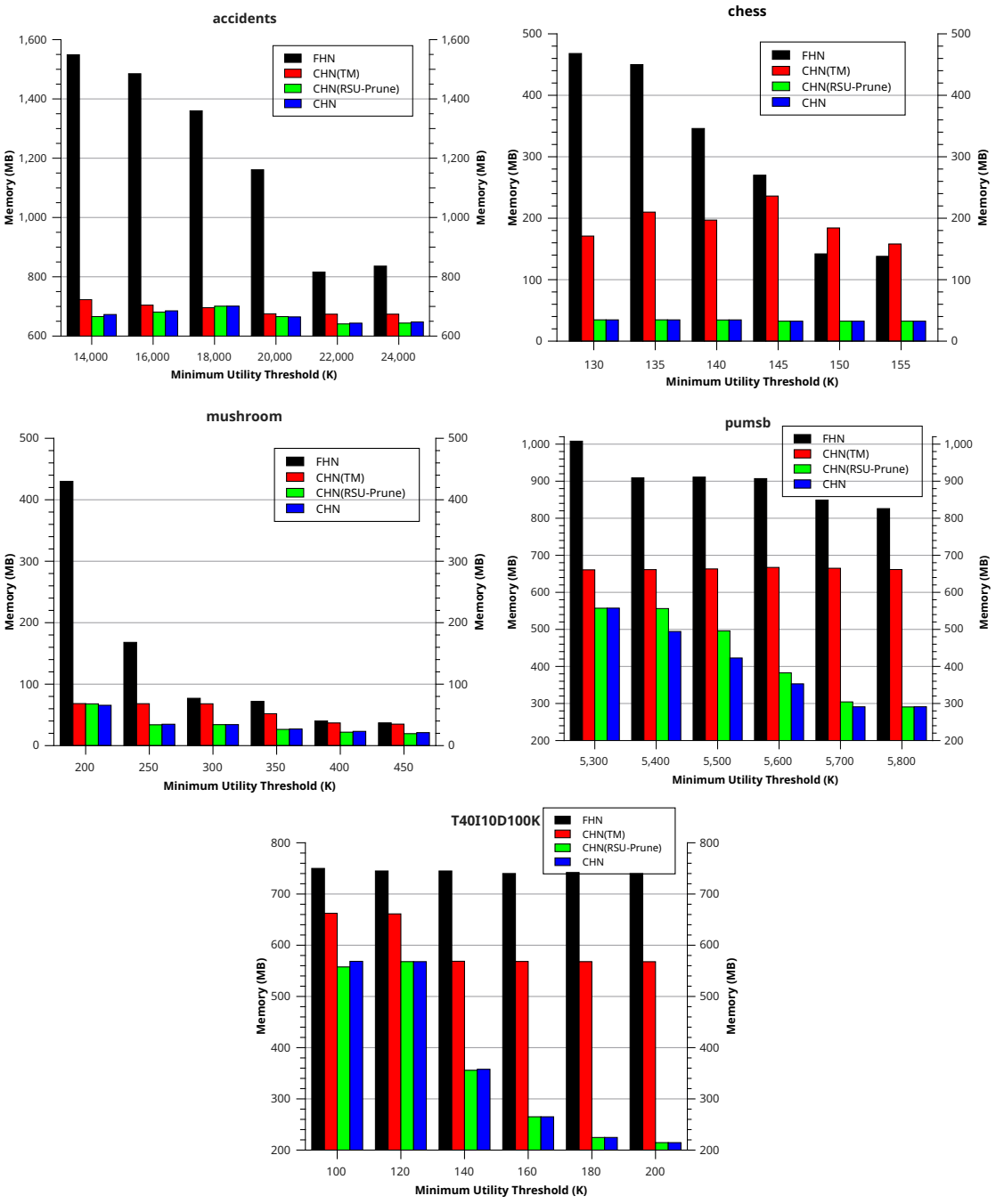

FIGURE 7.4: Memory usage on dense datasets

7.2.3 Memory Usages on Dense Datasets

FIGURE 7.4 shows the memory usage of the algorithms on the dense datasets. For accidents dataset, the proposed algorithms generally consume similar memory which is always less than the state-of-the-art algorithm FHN. Besides, the memory consumption

of CHN(TM) is always higher than other two algorithms (CHN(RSU-Prune) and CHN) except min_util is $18000K$. For chess, mushroom and T40I10D100K datasets, CHN(RSU-Prune) and CHN consume same amount of memory. CHN(RSU-Prune) and CHN are more stable when min_util threshold is decreased compared to CHN(TM) for chess dataset. And here also FHN consumes higher memory than all the proposed algorithms. As min_util threshold decreases, the memory consumption of FHN increase rapidly. When min_util is least ($200K$) CHN and CHN(RSU-Prune) perform better.

For the pumsb dataset, CHN always consumes less memory than other two variance algorithms and FHN as shown in FIGURE 7.4. For this dataset transaction merging and *RSU* techniques perform better together. Hence CHN uses less memory than other algorithms.

FHN uses higher memory most of the time because all the utility-lists as presented in memory while joining them. CHN(TM) not performs as well because it does not have an efficient pruning strategy *RSU*.

7.2.4 Memory Usages on Sparse Datasets

FIGURE 7.5 shows the memory usage of the algorithms on sparse datasets. For BMSPOS, retail and kosarak datasets, CHN and CHN(RSU-Prune) consumes less memory than CHN(TM) and FHN. While min_util decreases CHN uses relatively less memory than other proposed algorithms. CHN(RSU-Prune) and CHN are more stable when min_util threshold is decreased compared to CHN(TM) and FHN. All the proposed algorithm consume similar amount of memory for T10I4D100K dataset but FHN consumes much high memory. For large dataset kosarak, CHN and CHN(RSU-Prune) use less memory than FHN. For all the sparse datasets also FHN usage more memory than proposed algorithms as shown in FIGURE 7.5.

From the above performance study, we conclude that the proposed algorithms have good overall performance for both dense and sparse datasets. We can see that as min_util decreases CHN outperforms CHN(RSU-Prune) and CHN(TM) for most of the datasets. CHN performs better as min_util decreases and transaction merging strategy performs

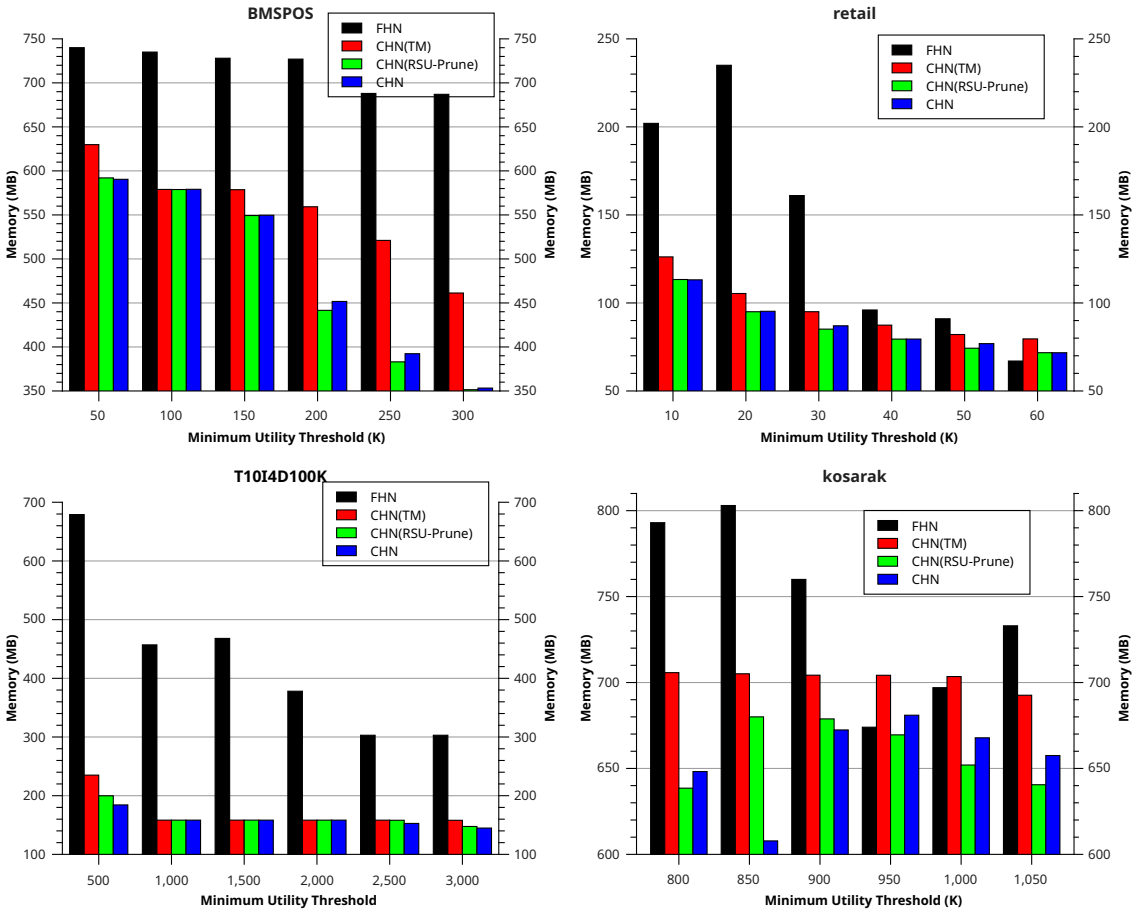

FIGURE 7.5: Memory usage on sparse datasets

well, while datasets need more processing time. The state-of-the-art algorithm FHN always fall behind the proposed algorithms.

7.2.5 Discussion

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

The runtime improvement performance of CHN over CHN(RSU-Prune), CHN(TM) and FHN at the lowest *min_util* are shown in TABLE 7.4. For example, CHN is more than 2.097 and 4.071 times faster than CHN(TM) and FHN respectively as shown in TABLE 7.4. CHN is 0.589 relatively times faster (1.696 times slower) than CHN(RSU-Prune). For memory usage comparison, CHN uses 1.075 and 2.303 times less memory than CHN(TM) and FHN respectively. The results reveal that CHN is not faster with respect to runtime than CHN(RSU-Prune) for most of the datasets. But CHN is always faster with respect to runtime than CHN(TM). However, memory improvement performance of CHN is always quite significant on all the datasets than CHN(RSU-Prune), CHN(TM) and FHN as shown in TABLE 7.4. CHN and its variants show poorer results on runtime performance for the retail dataset because retail is highly sparse. All the experimental evaluations show that the proposed algorithms significantly outperform than the state-of-the-art algorithm FHN with respect to both runtime and memory usage.

TABLE 7.4: Runtime and memory improvements of CHN over CHN(RSU-Prune), CHN(TM) and FHN

	Runtime			Memory		
Dataset	CHN(RSU-Prune)	CHN(TM)	FHN	CHN(RSU-Prune)	CHN(TM)	FHN
accidents	0.589	2.097	4.071	0.990	1.075	2.303
BMSPOS	0.588	1.041	1.138	1.003	1.067	1.253
chess	0.976	5.536	21.408	1.000	4.935	13.510
kosarak	0.634	4.553	0.978	0.985	1.089	1.223
mushroom	0.898	1.382	9.845	1.035	1.042	6.553
pumsb	0.800	648.829	44.791	1.000	1.185	1.808
retail	0.529	1.220	0.214	1.002	1.115	1.786
T10I4D100K	0.699	1.307	2.694	1.084	1.276	3.685
T40I10D100K	0.820	1.321	3.483	0.981	1.165	1.319

7.2.6 Effect of Techniques

In another performance comparison, we evaluate the influence of various techniques and ideas utilized by the proposed algorithm. We utilize transaction merging and *RSU* techniques to check the performance of the closed negative utility based mining algorithm. In order to assess the effectiveness of these techniques, we present two

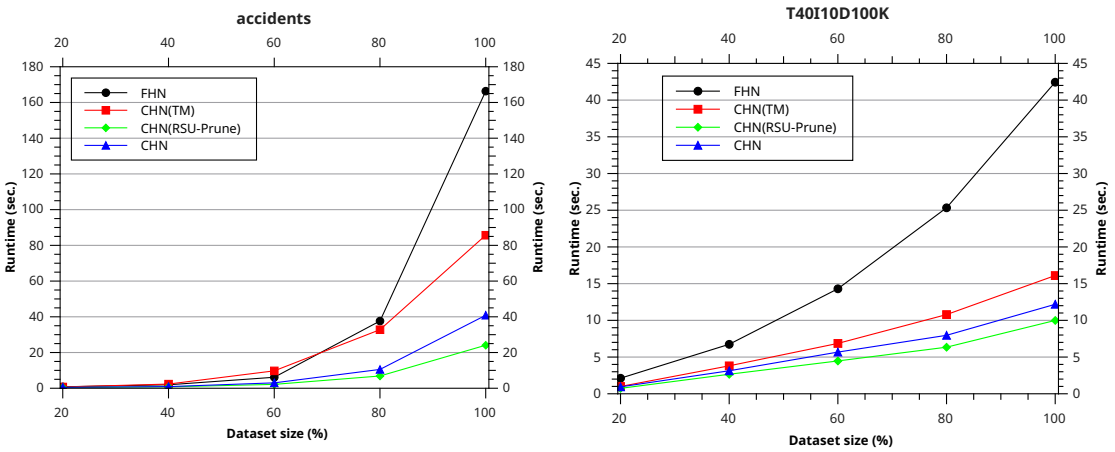

FIGURE 7.6: Scalability Runtime Comparison on accidents and T40I10D100K datasets

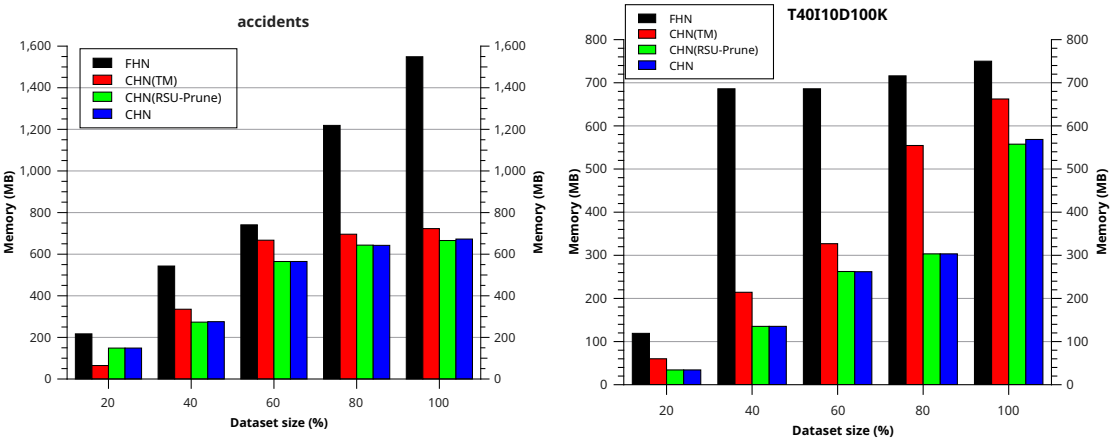

FIGURE 7.7: Scalability Memory Comparison on accidents and T40I10D100K datasets

versions of the proposed algorithms named CHN(RSU-Prune) and CHN(TM) which utilize only redefined sub-tree and transaction merging techniques respectively. CHN(TM) takes more runtime on almost all the dataset because it does not utilize *RSU* pruning strategy. CHN(TM) uses only transaction merging and *RTWU* pruning techniques. *RTWU* based strategy prune limited and less number of non-HUIs itemset compared to *RSU* based technique.

7.2.7 Scalability

In order to test the scalability of proposed algorithms, we use one real (accidents) and one synthetic (T40I10D100K) datasets. We fix *min_util* threshold to the lowest minimum

threshold that is used in each dataset for runtime and memory evolution. In order to evaluate the scalability of proposed algorithms, the size of datasets is varied from 20% to 100%. FIGURE 7.6 and FIGURE 7.7 show the runtime and memory usage scalability respectively of the proposed algorithms with FHN. These figures show that the runtime and memory usage of proposed algorithms increase linearly with increased dataset size. The proposed algorithms increase linearly for runtime and memory usages where FHN increases exponentially as shown in FIGURE 7.6 and FIGURE 7.7. Therefore, the proposed algorithms have good scalability under different dataset sizes and parameters compare to FHN.

7.3 Summary

In this chapter, we proposed a novel closed HUIs mining algorithm named CHN was introduced. The proposed algorithm considers negative utility. Negative utility exists in many real-life applications. In literature, only HUINIV-Mine [29] and FHN [30] algorithms are proposed to solve the negative utility itemsets mining. To the best of our knowledge, this is the first piece of work to mine closed HUIs with negative utility. The proposed CHN algorithm uses tree based data structure to store and maintain the information of the items. Two pruning strategies have been developed to remove non-HUIs and decrease execution time. Bi-directional extension closure checking technique is proposed to speed up the mining process. It also proposed bi-directional extension based two pruning strategies to prune the non-closed HUIs. CHN utilized transaction merging and dataset projection techniques to reduce the dataset scanning cost. CHN presented redefined TWU and redefined sub-tree strategies to prune the search space. Furthermore, CHN utilized utility-array based utility counting techniques to improve the performance. We presented an illustrative example of the proposed algorithm to understand the mining process. The experimental results on dense and sparse datasets show that the proposed algorithms are efficiently mine the closed HUIs. CHN is up to 44 times faster in execution time than FHN. CHN consume up to 13 times less memory than FHN. The comparison evaluation shows that the proposed algorithms significantly performs better than the current state-of-the-art algorithm FHN.