Positive Correlation Based Efficient High Utility Pattern Mining Approach

Dharavath Ramesh(✉), Krishan Kumar Sethi, and Aman Rathore

Department of Computer Science and Engineering, Indian Institute of Technology (ISM), Dhanbad 826004, Jharkhand, India
{ramesh.d.in,kksethi}@ieee.org

Abstract. The problem of high utility pattern (HUP) mining is an interesting task and comes with various applications. It has been argued in various past studies that the patterns with strong mutual correlation are more useful for decision making. A more powerful tool that takes the inherent correlation into account is more desirable. This paper presents an Efficient Correlated High Utility Miner (ECHUM) that considers the positive correlation among the items to find correlated high utility itemsets. The ECHUM uses a list structure is to avoid multiple scanning of the dataset. The search space is significantly reduced by using two upper-bounds named *sub-tree utility* and the *local utility*. Also, several database projection methods and transaction merging methods are used to reduce the complexity. Several pruning strategies are adopted to make the algorithm fast and memory efficient. A variety of experiments conducted shows that ECHUM is 2–3 times faster, and the memory usage is also 3–4 times lesser than the existing algorithm.

Keywords: High utility pattern mining · Correlation · Pruning strategy · Database projection

1 Introduction

High Utility Itemset Mining (HUIM) appears to be one of the most effective methods for pattern detection, which considers the quantity and profit of an itemset to discover the set of high utility itemsets (HUIs). Several Algorithms, like Two-Phase [1], tree-based [2–4], and one-phase [5–8] were introduced to find HUIs. The two-phase algorithms search potential candidate itemsets in the first phase and HUIs in the second phase. All two-phase algorithms suffer from scalability issues. The tree-based algorithms use a tree structure to produce the candidate itemsets and HUIs. The one-phase algorithms utilize a vertical list structure to search candidates and HUIs in a single phase. Several other one-phase algorithms like FHM [6], HUP miner [7], and EFIM [8] were introduced with several efficient pruning strategies. The EFIM algorithm introduces various novel ideas to improve time efficiency and memory optimization compared to the HUI miner [5]. It presents database projection and transaction merging to reduces the merging complexity and size of the database. EFIM also uses two upper-bounds sub-tree utility and local

© Springer Nature Switzerland AG 2021
K. R. Venugopal et al. (Eds.): ICInPro 2021, CCIS 1483, pp. 273–286, 2021.
https://doi.org/10.1007/978-3-030-91244-4_22

utility for efficient pruning. A fast utility counting method was introduced to calculate the utilities. This new counting method uses an array to store the utilities.

The main focus of the existing HUI mining algorithms [9, 10] is to search for the set of high utility items in a transactional dataset. However, they do not consider the mutual relationship among items of the discovered patterns. For example, a high-profit item like gold may appear with some low-profit items like cotton resulting in a high utility itemset. Still, cotton and gold are unrelated and could have occurred by chance. Hence, it is important to develop a framework that considers the correlation factor and produces high utility itemsets with inherent correlation. Many correlation measures have been commonly used for pattern mining, e.g., support [11], confidence [12], frequency affinity [13], all-confidence [14], and coherence [14]. HUIPM [13] and FDHUP [15] algorithms were introduced in utility-based pattern mining to discover HUIs using frequency affinity. Both algorithms use co-occurrence to calculate the correlation measure, which makes them ineffective. A projection-based algorithm called CoHUIM [16] was introduced, which considers the inherent correlation instead of the co-occurrence frequency. It discovers a set of high utility itemsets correlated using the correlation factor called Kulc [17]. The Kulc factor follows null transaction unvarying property to ensure that the correlation measure should not be affected by the number of null transactions in the database. However, the projection-based CoHUIM was found inefficient as it produces a huge number of candidate itemsets resulting in an extra memory consumption and efficiency decline. Correlated high-Utility Pattern Miner (CoUPM) [18] is a one-phase algorithm that uses the utility-list structure to store important details of the itemsets. This algorithm also uses the correlation factor Kulc to discover correlated patterns efficiently.

We observed that CoUPM [18] is an extension of the HUI-miner [5], which is computationally expensive for the small value of minimum utility. We propose an algorithm ECHUM, which exploits efficient EFIM [8]. Our choice is based on the fact that EFIM takes three times lesser time and takes eight times lesser memory than the HUI-miner. We have used the Kulc measure to find the inherent correlation in the discovered patterns. We adopted three major techniques to search the Correlated High Utility Itemsets (CoHUI). Firstly, database projection and transaction merging are used to reduce the size of the dataset and merging complexity. Secondly, sub-tree utility and local utility upper-bounds are used to prune the unpromising candidate itemsets. Thirdly, a fast counting method is adopted that uses an array to store the utilities to lower the upper bounds. We performed comprehensive experiments to compare ECHUM with the existing CoUPM algorithm. It has been observed that the ECHUM is faster than CoUPM.

The remaining paper is structured as follows. Section 2 discusses various preliminaries. The proposed algorithm is described in Sect. 3. The performance evaluation of the proposed algorithm has been carried out in Sect. 4. We conclude the paper in Sect. 5.

2 Preliminaries

Let D be a dataset consists of a collection of items I. An itemset X with k different items is called k-itemset. If X is a subset of transaction Tq then X is said to be contained in that transaction. An example of transaction and profit tables are shown in Tables 1 and 2.

Table 1. Transaction table

TID	Transactions	Count	TU
T_1	A_2 A_3 B_2	<1,1,2>	11
T_2	A_1 A_2 B_1	<3,1,2>	21
T_3	C_1	<1>	1
T_4	B_1 B_2	<2,4>	26
T_5	A_2 B_1	<3,1>	11
T_6	A_1 A_3 B_2 C_1	<1,2,1,3>	12
T_7	B_1 B_2 C_1	<3,3,4>	31
T_8	A_2 B_1 C_1	<2,2,2>	16
T_9	A_1 B_1 B_2	<2,2,1>	20
T_{10}	A_1 C_1	<4,5>	17

Table 2. Profit table

Stocks	A_1	A_2	A_3	B_1	B_2	C_1
Utility	3	2	1	5	4	1

Definition 1. Utility of item i in a transaction *T*, i.e., *U(i, T)* is the multiplication of its profit *p(i)* and quantity in that transaction *q(i, T).*

$$U(i, T) = p(i) \times q(i, T) \tag{1}$$

The utility value for an itemset X in a transaction T and dataset D is calculated as follows.

$$U(X, T) = \sum_{i \in X \wedge X \subseteq T} U(i, T) \tag{2}$$

$$U(X) = \sum_{X \in T \wedge T \subseteq D} U(X, T) \tag{3}$$

E.g., $U(B_2, T_4) = 4 \times 4 = 16$; $U(A_1B_2, T_6) = 1 \times 3 + 1 \times 4 = 7$; $U(A_1B_2) = (1 \times 3 + 1 \times 4) + (2 \times 3 + 1 \times 4) = 17$.

Definition 2. Transactional utility associated with any transaction *T* is represented by *tu(T)* and calculated as follows.

$$TU(T) = \sum_{i \in T} u(i, T) \tag{4}$$

E.g., $tu(T_4)$ is $(2 \times 5 + 4 \times 4) = 26$.

Definition 3. The transaction weighted utility of an itemset X is represented by *TWU(X)* and calculated as follows.

$$TWU(X) = \sum_{X \in T} TU(T) \tag{5}$$

E.g., $TWU(C_1) = 1 + 12 + 31 + 16 + 17 = 77$. The TWU values for running dataset are depicted in Table 3.

Table 3. Transactional weighted utility

Times	A_1	B_2	A_3	B_1	B_2	C_1
TWU	70	59	23	125	100	77

Definition 4 (High Utility Itemset). If the utility of an itemset X is greater than or equal to the minimum utility threshold *minutil*, it is said to be a high utility itemset.

$$HUI = \{X | u(X) \geq minutil\} \tag{6}$$

Definition 5 (Remaining Utility). Let *T/X* be the items appearing after X in a transaction T. The remaining utility is denoted as *ru(X)* and calculated as follows.

$$ru(X, T) = \sum_{i \in T/X} u(i, T) \tag{7}$$

E.g., $ru(B_1) = (4 \times 4) + (3 \times 4 + 4 \times 1) = 32$.

Property 1. If the value of Transactional Weighed Utility of X, i.e., *TWU(X)* is smaller than *minutil* then X and all its supersets are pruned.

Definition 6. The upper limit of remaining utility for itemset X, i.e., *reu(X)*, is calculated as follows.

$$reu(X) = u(X) + re(X) \tag{8}$$

E.g., $reu(B_1) = u(B_1) + ru(B_1) = 60 + 32 = 92$.

Property 2. If the remaining utility upper bound for X is smaller than the *minutil* then X and all its supersets are pruned.

Definition 7. The correlation value among the items of an itemset determines the strength of the inherent correlation. We use *Kulc* measure to find the correlation for an itemset X in ECHUM.

$$Kulc(X) = 1/k \sum_{i \in X} SUP(X)/SUP(i) \tag{9}$$

The value of *Kulc* lies between 0 and 1 and helps determine the positive correlation among the items.

2.1 Pruning Strategies

2.1.1 Exploring the Search Space

ECHUM explores the search space recursively in a depth-first search manner. In the search space, items are arranged in increasing order of their TWU. This order is selected because it reduces the number of candidates generated.

Definition 8. Extensions of itemset α is denoted by $E(\alpha)$ and defined as $E(\alpha) = \{y \mid y \in I\hat{}\wedge y > x\forall x \in a$. Let extension of itemset α be Z where $Z = \alpha \cup W$ if $W \in 2^{E(\alpha)}$ and $W \neq \emptyset$. Also, Z is a single itemset extension if $Z = \alpha \cup x$.

E.g., The single extensions of itemset $\{A_3\}$ are $\{A_3, B_2\}$, $\{A_3, A_1\}$, $\{A_3, C_1\}$ etc. The other extensions are $\{A_3, B_2, B_1\}$, $\{A_3, A_1, C_1\}$ etc.

2.1.2 2.1.2. Cost reduction Using Projections

Definition 9. For calculating the utility value for an itemset α, the itemsets that are not the extensions of α, i.e., $E(\alpha)$, are disregarded. The database consisting of such itemsets is called a projected database. The projection of a transaction T is denoted by $\alpha - T$, which is equal to $\{I \mid i \in T\hat{}i \in E(\alpha)\}$. The projection of the database is denoted as $\alpha - D$, which is equivalent to $\{\alpha - T \mid T \in D \hat{}\ \alpha - T \neq \emptyset\}$.

E.g., if $\alpha = \{B_2\}$ then the projected database will consist of 3 transactions i.e. $T_1\{A_2, A_2, B_2\}$, $T_4\{B_1, B_2\}$ and $T_7\{B_1, B_2, C_1\}$.

Database projections are used in ECHUM to reduce the database size. Database projection is carried out efficiently using *pseudo-projection* by sorting the database in total order. The method of pseudo projection is explained in EFIM [8].

2.1.3 Cost Reduction by Transaction Merging

Identical transactions are merged to form a single transaction. A transaction is a duplicate of another transaction if the same items are present in both the transactions, but each item's quantity can be different.

Definition 10. Transaction merging is applied to each of the projected transaction databases $\alpha - T$, and all the duplicate transactions are combined to form a single transaction in the projected database. Consider a transactional database D; if the transactions are sorted in lexicographical order and are read from end to start, then the order is known as total order and is denoted by $>$.T.

Transaction merging is used to merge a set of transactions efficiently, and after applying this method to the projected database, the number of transactions is drastically reduced. The smaller the transaction length, the greater the probability of matching transactions. The database is sorted according to the $>$.T order to merge identical transactions efficiently. The transactions are combined efficiently using the property introduced in EFIM [8] by first sorting the transactions lexicographically and then introducing various measures to compare the transactions.

Property 3. In a transactional database sorted according to the >.T property, the duplicate transactions will appear consecutively. All the duplicate transactions can be removed directly from the database by comparing the current transaction with its next transaction.

2.1.4 Pruning Using Upper Bounds

Definition 11. Consider an itemset α associated with a single extended itemset *y*. The *sub-tree* utility of the element *y* is denoted as *su*(α,y).

$$su(\alpha, y) = \sum\nolimits_{T \in g(\alpha \cup \{y\})} [u(\alpha, T) + u(y, T) + \sum\nolimits_{i \in T \wedge i \in E(\alpha \cup \{y\})} u(i, T)] \quad (10)$$

E.g., if $\alpha = \{A_3\}$ and z = $\{B_2\}$ then su(α,*z*) = (1 + 8) + (2 + 4 + 3) = 18.

Property 4. Consider an itemset α, the utility value of the single extensions of α is smaller than or equal to the *sub-tree utility*. Mathematically, $su(\alpha, y) \geq u(Y)$, where *Y* is known to be only the single extended itemset of α.

Pruning Method 1. If the *sub-tree utility* of α is smaller than *minutil* then the single itemset extensions of α cannot be a HUI, and therefore the extensions are pruned.

Definition 12. Consider an itemset α, the extension of α denoted by $y \in E(\alpha)$, the *local utility* of *y* concerning α is indicated by $lu(\alpha, y)$.

$$lu(\alpha, y) = \sum\nolimits_{T \in g(\alpha \cup \{y\})} [u(\alpha, T) + ru(\alpha, T)] \quad (11)$$

Property 5. For an itemset α, the *local utility* value of α is always greater than or equal to the utility of the extensions of α. Mathematically, $lu(\alpha, y) \geq u(Y)$, where *Y* is any extension of α.

Pruning Method 2. For itemset α, if the *local utility* value of α is less than the *minutil* then the single itemset extensions of α cannot be a high-utility itemset. Therefore, the extensions are pruned.

Property 6. For any itemset α with an extension Y, the following property always holds.

$$TWU(\alpha) \geq lu(\alpha, y) \geq su(\alpha, y) \quad (12)$$

Definition 13. The *primary items* of an itemset α are the collection of all items, which are the extensions of α whose *sub-tree* utility is greater than *minutil*. The *secondary items* of α are the collection of those items: the extensions of α, and *local utility* is greater than *minutil*. Mathematically, $Primary(\alpha) = \{y | y \in E(\alpha) \wedge su(\alpha,y) > minutil\}$ and $Secondary(\alpha) = \{y | y \in E(\alpha) \wedge lu(\alpha,y) > minutil\}$. By Property 6, it can be said that *Primary items* are the subset of *Secondary items*.

2.1.5 Pruning Using Correlation Properties

Property 7. In a transactional database where are the items in a transaction are arranged according to their support values, the *Kulc* measure uses the following *sorted downward closure property.*

$$Kulc(i_1, i_2 \ldots\ldots\ldots, i_k) \leq Kulc(i_1, i_2 \ldots\ldots\ldots, i_k, i_{k-1}) \tag{13}$$

Pruning Method 3. Consider an itemset X for which the value of *Kulc(X)* is smaller than any of the *Kulc* values of its subsets, then the collection of every superset of X are not CoHUI's. Let X_k be an itemset and subset X_{k-1} be its subset; Mathematically, if $Kulc(X_k) \leq Kulc(X_{k-1})$, then all supersets of X_k are not in CoHUI's.

2.2 Improving Efficiency Using Fast Utility Counting

In the previous section, two utility-based pruning methods are used to eliminate ineffective itemsets. To compute the utility upper bounds, EFIM introduced a technique that uses an array data structure to store utilities known as Fast Utility Counting (FUC). The novel array structure is called utility-bin [19]. Let a dataset D containing $|I|$ items, and *U[y]* denotes the *utility bin*, where y is any item in itemset I. The array is initialized with 0. Following is the process for the calculation of upper limits by using the *utility bin* array.

TWU Upper-Bound. The TWU upper bound is calculated using the utility bin array. Initially, values in the array are set to 0. Then by proceeding transaction-wise in the database, the values of each item of array *U[z]* are modified by the total utility *TU(T)* of that transaction. i.e., $U[y] = U[y] + TU(T)$, where y is the item contained in transaction T. Finally, all the values of the utility array will be equal to the *TWU* value.

Sub-tree Utility upper-bound associated with any itemset α is calculated using the *utility bin* in the following manner. We proceed with the dataset transaction-wise, such that the sub-tree utility value modifies each item's utility in U[p] for that transaction.

$$U[p] = U[p] + u(p, T_i) + u(\alpha, T_i) + \sum\nolimits_{i \in T \wedge i >.z} u(i, T) \tag{14}$$

Where p is the extension itemset in transaction T_i. Finally, all the utility array values will be equal to the subtree utility value, i.e., $su(p, \alpha)$.

Local-Utility upper-bound associated with any itemset α, the dataset is processed transaction-wise. Utility values of each item of array *U[p]* are modified by the "local-utility" of that transaction.

$$U[p] = U[p] + \sum\nolimits_{i \in T \wedge i >.z} u(i, T) + reu(\alpha, T) + u(\alpha, T) \tag{15}$$

Where p is the extension itemset in transaction T. Finally, all the utility array values will be equal to the local utility value, i.e., $lu(y, \alpha)$.

2.3 Problem Statement

An itemset *X* is called a CoHUI if the value of *u(X)* is more than *minutil*, and *Kulc(X)* is also greater than the minimum correlation measure *minCor*.

$$\text{CoHUI} = u(X) \geq minutil \text{ AND } Kulc(X) \geq minCor \tag{16}$$

3 ECHUM Algorithm Design

The ECHUM algorithm is the combination of the two algorithms EFIM and CoHUM. The updated utility-list is used in this method, which considers the utility-list formation of EFIM and the support value of each item used in CoHUM. The algorithm takes input as the transactional database, utility table, minimum utility *minutil,* and minimum correlation *minCor*. Initially, an itemset *X* and the support values are initialized to Ø. The *local-utility* associated with every item is calculated using the utility-bin array by examining the database. The *secondary itemsets* are then generated with the help of the *local-utility* associated with every item. Now, the *secondary itemsets* are arranged based on the increasing order of their *TWU* values. The scanning of the database is then done again to withdraw the collection of items that are not present on the *secondary itemsets,* and the empty transactions are deleted. Finally, the sorting of *D* takes place once again according to their >.T order. The sub-tree utility is denoted by *su*(*X, i*) for every item present in the secondary itemset andcalculated using the utility bin array. The primary itemset list is generated using these values of the sub-tree utility array. The *primary itemset, secondary itemset, minutil, minCor,* and database *D* are then passed to the Explore function. Algorithm 1 depicts the pseudo-code of ECHUM.

Algorithm 1: ECHUM Algorithm

Input: Transactional database *D*, utility table *P*, *minutil* and *minCor*
Output: The collection of CoHUI's

1. The itemset *X* is initialized with value *Ø*;
2. Initially, the database D is examined, and the local utility of each element is calculated.
3. Using the local-utility for every item the Secondary items are generated by *Secondary(X) = { i | i∈ I ^ lu(X,i) ≥ minutil)*
4. The items in the *Secondary(X)* are sorted according to the *TWU* values.
5. The scanning of the database is then again applied to *D* to extract the absent transactions in *Secondary(X).*
6. The transactions in D are again sorted according to the TWU values.
7. Again the database D is examined and the calculation of sub-tree utility of every element for the items that are present in *Secondary(X).*
8. Using the local-utility associated with every item the Secondary items are generated by *Primary(X) = {i | i∈ Secondary(X) ^ su(X,i) ≥ minutil)*
9. Explore(*X, D, Primary(X), Secondary(X), minutil, minCor)*

Algorithm 2: Explore Procedure

Input: An itemset *X*, the projection of the original database *X-D, Primary(X), Secondary(X), minutil* and *minCor*

Output: The collection of all CoHUI's, which are the extension of *X*.

1. For each item *i* present in Primary itemsets do
2. *Y = X U { i }*
3. Update the support value and the corresponding *Kulc* value of *Y* by increasing its count;
4. Scan the projected database *X -D* for the calculation of the utility *u(Y)* and for the creation of the projection of the new database *Y-D*
5. If the following holds: *u(Y) > minutil and Kulc(Y) > minCor then output Y*;
6. Calculate *su(Y,y)* and *lu(Y,y)* for each item y∈*Secondary(X)* by examining the extended database *β-D* once.
7. *Primary(Y) = {y | y ∈ Secondary(X) | su(Y,y) ≥ minutil* and *Kulc(Y y)>minCor }*
8. *Secondary(Y) = {y | y ∈ Secondary(X) | lu(Y,y) ≥ minutil* and *Kulc(Yy)>minCor }*
9. Explore(*Y, Y -D, Primary(Y), Secondary(Y), minutil, minCor);*
10. End

The Explore Method takes input as itemset *X*, projected database $X - D$, the items that primary items to *α: Primary(X),* and secondary items to *X: Secondary(X),* user-defined *minutil* and *minCor* threshold. For every item *i* in *Primary(X),* an itemset *Y* is created, which is the extension of *X*. The support values are updated in the utility list of *Y* by incrementing its count. The projected database $X - D$ is then examined to find the utility of *Y,* and the projection of the database $Y - D$ is created. If the utility of *Y* is larger than the *minutil* and the *Kulc* value is greater than the *minCor* then the itemset *Y* is considered as CoHUI. Then the *sub-tree utility su(Y, y)* and the *local utility lu(Y, y)* are calculated for every item *y* present in *Secondary(X)* by scanning the projected database $Y - D$. Finally, the itemsets *Primary(Y)* and *Secondary(Y)* are generated from the itemset *Secondary(X)* by applying Properties 4, 5, and 6. The explore method is recursively called again to search the extensions of *X*. The final output is accumulated in the set of CoHUIs.

4 Experimental Analysis

The ECHUM algorithm is compared with the CoUPM. We performed the analysis of execution time and memory usage by varying the *minutil* and *minCor*. The experiments were carried out on an Ubuntu machine with a 2.7 GHz Intel i5 6th generation processor. The RAM allotted to the system was 8 GB, and the program code was written in Java Language. A total of 4 datasets were used to test the algorithms. The datasets were collected from SPMF library [20]. The dataset properties are given in Table 4.

4.1 Runtime Comparison

The runtime comparisons are shown in Figs. 1 and 2. In Fig. 1, the *minutil* for *foodmart*, *chess, mushroom,* and *retail* are kept fixed at 0.007%, 20%, 8%, and 0.12%, respectively. The *minCor* value is varied from 0.01 to 0.06 in *foodmart* dataset, from 0.74 to 0.79 in *chess* dataset, from 0.4 to 0.5 in *mushroom* dataset, and from 0.1 to 0.2 in *retail* dataset. For almost all the values, the running time for ECHUM is faster than the existing CoUPM algorithm. As the *minCor* value is increased, the run time required for each of

Table 4. Dataset properties

Dataset	Transactions	No. of distinct items	Average transaction length
Chess	3196	75	36
Mushroom	8124	120	23
Retail	88162	16470	10.3
Food mart	4141	1559	4.42

the algorithms is persistent. In Fig. 2, the *minCor* for foodmart, chess, mushroom, and retail are kept fixed at 0.03, 0.76, 0.42, and 0.1, respectively. The *minutil* value is varied from 0.006% to 0.011% in the food mart dataset, from 17 to 22 in the chess dataset, from 8 to 13 in the mushroom dataset, and from 0.01 to 0.015 in the retail dataset. For almost all the values, the running time for ECHUM is faster than the existing algorithm CoUPM. As the *minutil* value is increased, the run time required for each of the algorithms is persistent.

When the *minCor* is set fixed and the *minutil* is increased, the time taken by both the CoUPM and the ECHUM Algorithm decreases non-linearly. For the dataset, *foodmart* the time taken by both the algorithms is almost similar but is less for the ECHUM algorithm. For the datasets, *chess* and *mushroom,* we have a drastic time difference in both algorithms with ECHUM the faster. For *retail* dataset, the time difference is the most. The reason is that CoUPM generates more unpromising candidates due to a huge number of transactions and many distinct items. More candidates require extra memory for CoUPM. It can be seen that the ECHUM algorithm outperforms in all the cases as the number of candidates generated is very low because of efficient pruning.

4.2 Memory Comparison

The Memory Consumptions by the two algorithms ECHUM and CoUPM are shown in Figs. 3 and 4. In the first graph, the *minutil* for *foodmart*, *chess*, *mushroom,* and *retail* are kept fixed at 0.007%, 20%, 8%, and 0.12%, respectively. The *minCor* value is varied from 0.01 to 0.06 in *foodmart* dataset, from 0.74 to 0.79 in *chess* dataset, from 0.4 to 0.5 in *mushroom* dataset, and from 0.1 to 0.2 in *retail* dataset. For some datasets, the memory consumed by ECHUM is more, while for some datasets, the memory consumed is less. The overall memory consumed is lesser for the ECHUM algorithm than the CoUPM Algorithm.

For the first graph in Fig. 3, the ECHUM algorithm's memory is a little more than the CoUPM algorithm. The average memory consumed is around 100 MB for ECHUM, while it is around 90 MB for CoUPM. In the *chess* dataset, the ECHUM algorithm took less than half the memory consumed by the CoUPM algorithm for all cases. The average memory consumed by the CoUPM algorithm for the *chess* dataset is 230 MB, while for the ECHUM algorithm, it is 120 MB. The ECHUM algorithm performs best in the *mushroom* dataset in terms of memory consumption. It is almost 5 times efficient than the CoUPM algorithm. For the *retail* dataset, the memory taken is similar for both the algorithms, but it is less for ECHUM.

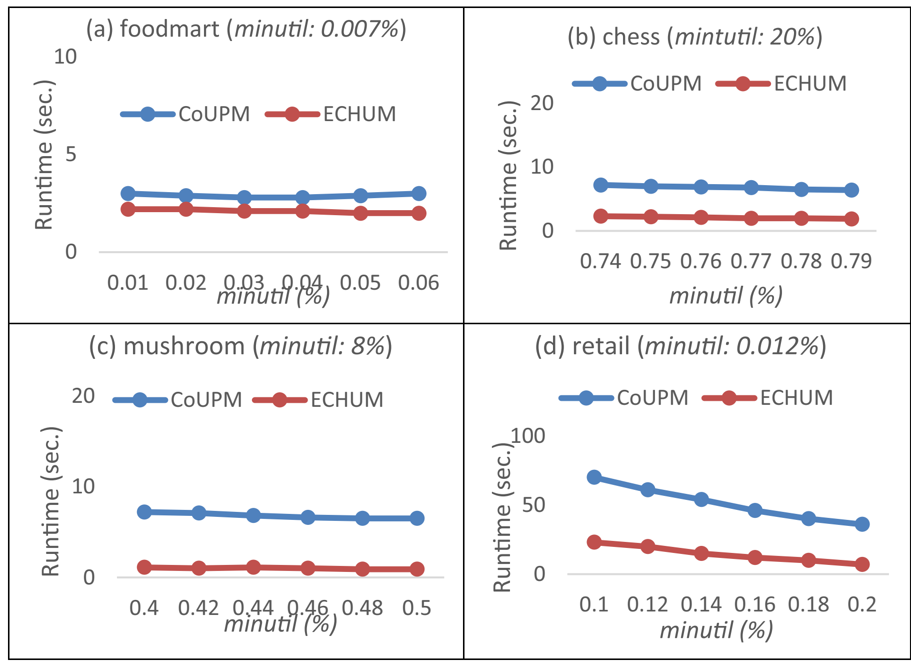

Fig. 1. Runtime comparison for fixed *minutil* and variable *minCor*

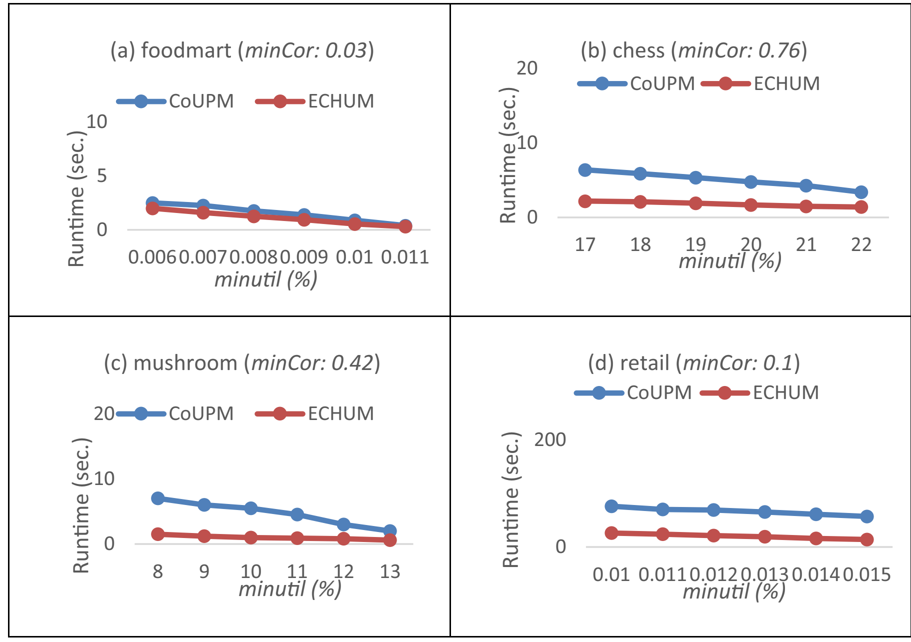

Fig. 2. Runtime comparison for fixed *minCor* and variable *minUtil*

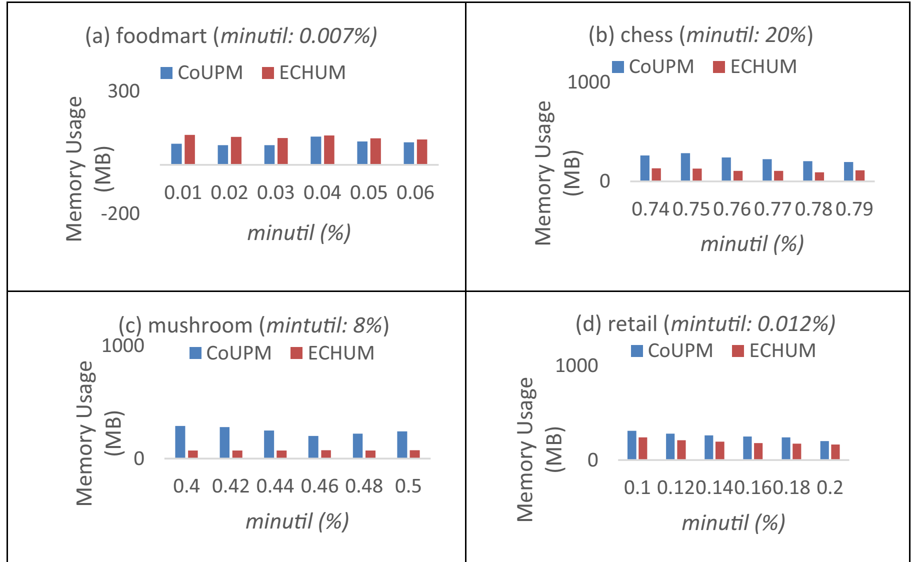

Fig. 3. Memory consumption comparison for fixed *minutil* and variable *minCor*

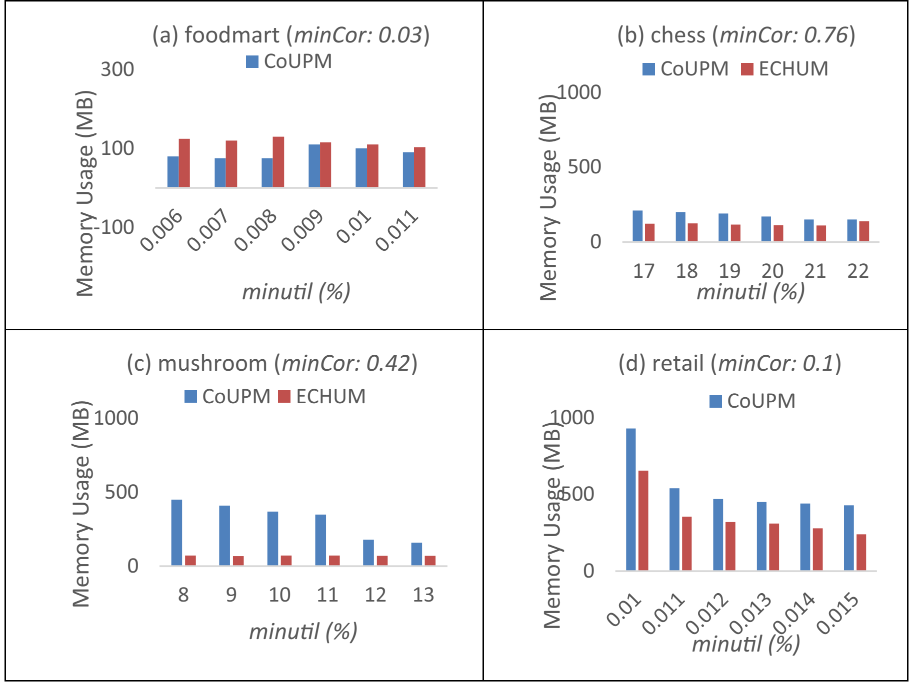

Fig. 4. Memory consumption comparison for fixed *minCor* and variable *minutil*

In Fig. 4, the minCor for *food mart*, *chess*, *mushroom,* and *retail* are kept fixed at 0.03, 0.76, 0.42, and 0.1, respectively. The *minutil* value is varied from 0.006% to 0.011% in food mart dataset, from 17 to 22 in *chess* dataset, from 8 to 13 in *mushroom* dataset, and from 0.01 to 0.015 in *retail* dataset. The first graph in Fig. 4 represents the *food mart* dataset in which for all the *minutil* values, the memory consumed by ECHUM is more than the CoUPM algorithm. The average memory consumed by the CoUPM algorithm for the *food mart* dataset is 90 MB, while for ECHUM, it is 115 MB. For the *chess* dataset, the ECHUM algorithm is faster than CoUPM in all cases. The ECHUM algorithm works best for the *mushroom* dataset as the average memory consumed is almost 5 times less than the CoUPM algorithm. For the *retail* dataset, the memory consumed by both algorithms is high, which is because of the large transactions, but the memory consumed by ECHUM is less than the CoUPM algorithm. Overall the memory consumed by the ECHUM algorithm is less than the CoUPM algorithm.

It is evident from the runtime and memory consumption graphs that ECHUM is almost 200–300% faster than CoUPM and takes nearly 50% less memory than CoUPM. The pruning methods help to reduce the search space, and also, the use of a better utility-list structure minimizes the memory usage.

5 Conclusion and Future Scope

This paper presents an efficient algorithm ECHUM to reduce the time and space required to search the CoHUIs. The idea involved in this paper is to combine the two algorithms EFIM and CoHUM, such that the overall complexity is reduced. The ECHUM uses two upper utility bounds named *sub-tree utility* and the *local utility* and a fast utility counting measure, which improves efficiency. The ECHUM also uses transaction and projection merging, which reduces the dataset size and enhances efficiency. More optimizations will be done for future work in this algorithm by developing a much tighter upper bound to reduce the search space.

References

1. Liu, Y., Liao, W., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Ho, T.B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS (LNAI), vol. 3518, pp. 689–695. Springer, Heidelberg (2005). https://doi.org/10.1007/11430919_79
2. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S., Lee, Y.K.: Efficient tree structures for high utility pattern mining in incremental databases. IEEE Trans. Knowl. Data Eng. **21**(12), 1708–1721 (2009)
3. Tseng, V.S., Wu, C.W., Shie, B.E., Yu, P.S.: UP-growth: an efficient algorithm for high utility itemset mining. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 253–262 (2010)
4. Tseng, V.S., Shie, B.E., Wu, C.W., Philip, S.Y.: Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans. Knowl. Data Eng. **25**(8), 1772–1786 (2012)
5. Liu, M., Qu, J.: Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, pp. 55–64 (2012)

6. Fournier-Viger, P., Wu, C..-W.., Zida, S., Tseng, V.S.: FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Andreasen, T., Christiansen, H., Cubero, J.-C., Raś, Z.W. (eds.) ISMIS 2014. LNCS (LNAI), vol. 8502, pp. 83–92. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08326-1_9
7. Krishnamoorthy, S.: Pruning strategies for mining high utility itemsets. Expert Syst. Appl. **42**(5), 2371–2381 (2015)
8. Zida, S., Fournier-Viger, P., Lin, J.C.-W., Wu, C.-W., Tseng, V.S.: EFIM: a highly efficient algorithm for high-utility itemset mining. In: Sidorov, G., Galicia-Haro, S.N. (eds.) MICAI 2015. LNCS (LNAI), vol. 9413, pp. 530–546. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27060-9_44
9. Gan, W., Lin, J.C.W., Fournier-Viger, P., Chao, H.C., Tseng, V.S., Philip, S.Y.: A survey of utility-oriented pattern mining. IEEE Trans. Knowl. Data Eng. **33**(4), 1306–1327 (2019)
10. Sethi, K.K., Ramesh, D., Sreenu, M.: Parallel high average-utility itemset mining using better search space division approach. In: Fahrnberger, G., Gopinathan, S., Parida, L. (eds.) ICDCIT 2019. LNCS, vol. 11319, pp. 108–124. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05366-6_9
11. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceeding of 20th International Conference on Very Large Databases. VLDB, vol. 1215, pp. 487–499 (1994)
12. Kim, W.-Y., Lee, Y.-K., Han, J.: CCMine: efficient mining of confidence-closed correlated patterns. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 569–579. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24775-3_68
13. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S., Choi, H.J.: A framework for mining interesting high utility patterns with a strong frequency affinity. Inf. Sci. **181**(21), 4878–4894 (2011)
14. Omiecinski, E.R.: Alternative interest measures for mining associations in databases. IEEE Trans. Knowl. Data Eng. **15**(1), 57–69 (2003)
15. Lin, J.-W., Gan, W., Fournier-Viger, P., Hong, T.-P., Chao, H.-C.: FDHUP: fast algorithm for mining discriminative high utility patterns. Knowl. Inf. Syst. **51**(3), 873–909 (2016). https://doi.org/10.1007/s10115-016-0991-3
16. Gan, W., Lin, J.C.W., Fournier-Viger, P., Chao, H.C., Fujita, H.: Extracting non-redundant correlated purchase behaviors by utility measure. Knowl.-Based Syst. **143**, 30–41 (2018)
17. Gan, W., Lin, J.C.W., Chao, H.C., Fujita, H., Philip, S.Y.: Correlated utility-based pattern mining. Inf. Sci. **504**, 470–486 (2019)
18. Kulczynski, S.: Die pflanzenassoziationen der pieninen, Imprimerie del'Universit (1928)
19. Song, W., Liu, Y., Li, J.: BAHUI: fast and memory efficient mining of high utility itemsets based on bitmap. Int. J. Data Warehousing Min. (IJDWM) **10**(1), 1–15 (2014)
20. Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.W., Tseng, V.S.: SPMF: a Java open-source pattern mining library. J. Mach. Learn. Res. **15**(1), 3389–3393 (2014)