International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 13 (2021) pp. 072-079
© MIR Labs, www.mirlabs.net/ijcisim/index.html

Received: 10 Jan 2021; Accepted: 15 May 2021; Published: 13 July 2021

A novel fast algorithm for mining k-item High Utility Itemsets from incremental databases

Nong Thi Hoa[1], Nguyen Van Tao [2]

[1] School of Computer Science, Duy Tan University,
30 Tran Phu, Hai Chau, Da Nang 550000, Viet Nam
nongthihoa@duytan.edu.vn

[2] University of Information and Communication Technology, Thai Nguyen University,
Quyet Thang, Thai Nguyen, Thai Nguyen 250000, Viet Nam
nvtao@ictu.edu.vn

***Abstract*: Mining High Utility Itemsets (HUIs) discovers itemsets making much profit in business from a transaction database. Therefore, mining HUIs is important for planing business. Previous studies use a tree structure and pruning strategies, or a list structure and generation of promising itemsets to decrease both time and memory for computing. Fast algorithms for mining compact HUIs have proposed to discover close HUIs or maximum HUIs. However, these studies still take a long time and consume much memory because of considering all itemsets of items in a transaction. Moreover, business managers usually make decisions more effectively based on itemsets containing several items. In this paper, we propose a novel fast algorithm for mining k-item HUIs that meets the need of manages and decreases both time and memory for mining HUIs. We use a simple list structure to store k-itemsets appearing during browsing transactions. This list stores items and utility of each itemset. Our algorithm consists of two main steps. First, the current database is vertically segmented into sub-partitions. Each row in a sub-partition contains *k* items. Then, k-item HUIs are mined from each sub-partition. The proposed algorithm obtains advantages including without candidate generation, without re-scanning database when changing the threshold of utility. Experiments are conducted on dense benchmark databases. Results of experiments show that our algorithm is better than state-of-the-art methods.**

***Keywords*:** High Utility Itemset, HUI, Mining High Utility Itemsets, mining HUIs, k-item HUIs, inc-k-HUIs-Miner, Data Mining**.**

I. Introduction

The rapid development of database techniques creates much massive data from business corporations, governments, and organizations. Valuable information in these databases need obtain to support decision making. Association rules, hiding in transaction databases, present the relationship between items. Decision makers can use these rules to plan business. Mining HUIs finds itemsets whose utility overcomes a given threshold. Mining HUIs from incremental databases updates HUIs from newest transactions. Therefore, mining HUIs from incremental databases is essential for business managers and recommendation systems in Intelligence Business.

Studies on mining HUIs focus on decrease both time and memory for computing. These studies can be divided into two categories including mining exact HUIs and mining compact HUIs. Mining HUIs usually uses a tree or list structure to store information during browsing transactions. A list structure and combining items based on identification of transactions are used to avoid generating unpromising itemsets [1] [2] [3]. A tree structure and pruning strategies are applied to decrease the search space [4] [5] [6]. Recently, several fast algorithms are proposed to discover compact HUIs such as closed HUIs [7] [8] [9] [10] and maximal HUIs [7][11]. However, these studies still consume both time of CPU and memory because of processing all items in each transaction. Moreover, decision makers usually need HUIs containing several items to plan business more effectively. HUIs which have a few items are not enough information for decision making. HUIs including many items require decision makers to consider the importance of each item. Additionally, a transaction database contains a few of HUIs with many items. Therefore, new algorithms need focus on mining HUIs including several items. In this paper, we propose a novel fast algorithm for mining k-item HUIs that meets the need of decision makers and decreases both time and memory for computing. Specifically, we make the following main contributions:

- We present a simple list structure to store k-itemsets during browsing transactions. This list is also added or updated when giving a incremental database.
- We propose an effective fast algorithm for mining k-item HUIs. Firstly, we make a vertical segmentation for the current database. As a result, we obtain sub-partitions that each row contains *k* items. Next, k-itemsets in each sub-partition are mined and stored in a global list. Then, extracting k-item HUIs from the list based on utility of itemsets.
- We conduct experiments on benchmark dense databases which have various different characteristics. We compare the performance of our approach against state-of-the-art algorithms. Results show that the proposed algorithm significantly decreases both time and memory for computing, and it outperforms compared algorithms.

This paper is organized as follows. Next sections are

Preliminaries and Related work. In Section IV, we present our approach for mining k-item HUIs. Section V shows experiment results and comparison to other algorithms. Finally, conclusions are written in Section VI.

II. Preliminaries

Let I={i_1, i_2,....., i_n} be a set of items and DB is a database composing of a utility table and a transaction table. Each item in I has a utility value in the utility table. Each transaction T in the transaction table has a unique identifier (Tid) and is a subset of I in which each item is associated with a count value.

An itemset is a subset of I and is called a k-itemset if it contains *k* items.

T_{id}	Transaction	Count
T_1	A, C, D	1, 1, 1
T_2	A, C, E, G	2, 6, 2, 5
T_3	A, B, C, D, E, F	1, 2, 1, 6, 1, 5
T_4	B, C, D, E	4, 3, 3, 1
T_5	B, C, E, G	2, 2, 1, 2

Table 1. Transaction Table.

Item	A	B	C	D	E	F	G
Profit	5	2	1	2	3	1	1

Table 2. Profit Table.

Definition 1: The external utility of item *i*, denoted as *eu(i)*, is the utility value of *i* in the utility table of DB. For example, in Table 2, *eu*(B)=2.

Definition 2: The internal utility of item *i* in transaction T, denoted as *iu*(*i*, T), is the count value associated with *i* in T in the transaction table of DB. For example, in Table 1, *iu*(C, T_2)=6.

Definition 3: The utility of item *i* in transaction T, denoted as *u*(*i*, T), is the product of *iu*(*i*, T) and *eu*(*i*), where *u*(*i*, T) = *iu*(*i*, T)**eu*(*i*). For example, *u*(A, T_2)=5*2=10.

Definition 4: The utility of an itemset X in transaction T, denoted as *u*(X, T), is the sum of utilities of all items in X in T in which X is contained, where $u(X, T)= \sum_{i \in X \wedge X \subseteq T} u(i,T)$. For example, *u*(AC, T_2)=*u*(A, T_2) +*u*(C, T_2)=5*2+1*6=16.

Definition 5: The utility of an itemset X, denoted as *u*(X), is the sum of utilities of X in all transactions containing X in DB, where $u(X)= \sum_{T \in DB \wedge X \subseteq T} u(X,T)$. For example, *u*(AC)=*u*(AC, T_1)+*u*(AC, T_2)=6+16=22.

Definition 6: The utility of transaction T in DB, denoted as *tu*(T), is the sum of utilities of all items in T, where

$tu(T)= \sum_{T \in DB \wedge i \in T} u(i,T)$. For example, *u*($T_2$)=*u*(A, T_2)+*u*(C, T_2) +*u*(E, T_2)+*u*(G, T_2)=10+6+6+5=27.

Definition 7: The transaction-weighted utility of an itemset X in DB, denoted as *twu*(X), is the sum of utilities of all transactions containing X in DB, where $twu(X)= \sum_{T \in DB \wedge X \subseteq T} tu(T)$. For example, *twu*(A)=*tu*(T_1)+ *tu*(T_2) + *tu*(T_3) =8+27+30= 65.

Definition 8: Given a threshold of utility *minutil*, an itemset X is high utility itemset if *u*(X)≥*minutil*.

Definition 9: An itemset X is *a closed high utility itemset* (CHUI) if there exists no proper superset Y⊃X in DB such that *sup*(X)=*sup*(Y) and its utility *u*(X) ≥*minutil*.

Definition 10: An itemset X is *a maximal high utility itemset* (maximal HUI) if *u*(X) ≥*minutil* and there is no HUI that is a proper superset of X.

Transaction weighted downward closure property: If twu(X)<*minutil*, all supersets Y⊃X are not high utility itemsets.

III. Related works

Studies on mining HUIs focus on decreasing both time and memory for computing. Recently, mining HUIs from incremental databases and mining compact HUIs have widely developed to meet the need of real applications.

A. *Mining HUIs from incremental databases*

Mining HUIs from incremental databases finds HUIs from newest transactions. Therefore, knowledge from databases is frequently updated to support better for decision makers. L. Judae and et al. [4] proposed an approach of pre-large concept with a proper data structure to mine high utility patterns (HUPs). This method required only one scan as well as mined in dynamic environments. They used a tree structure (PIHUPL-tree) includes a header and a tail list. This tree was ordered by lexicographic. The header accumulated *twu* of 1-itemsets and the tail list stored links pointing to nodes of PIHUPL-tree. Each node indicated the last item of an inserted transaction. After the construction, the header was reordered by a *twu* descending. Y. Unil and et al. [1] presented an algorithm for mining HUPs with one database scan. They used set of utility lists to store and maintain the utility of candidate patterns. For each transaction T containing item *i*, utility of *i* in T had three elements: a Tid of T, a pattern utility of *i* in T, and a remaining utility in T. After the global data structure was constructed from the original database or updated from increased data, it was restructured according to a *twu* ascending order. Then, an algorithm mined from a utility list for a promising candidate pattern composed of an item with the smallest *twu* value. The mining process was performed based on HUI-Miner. R. Heungmo and Y Unil [5] proposed an algorithm for mining HUPs from data streams. A tree structure (SHU-Tree) was used to maintain information of transactions and HUPs in the current window. Each node of SHU-Tree included elements: the item name, N.name; the more reduced overestimation utilities than *twu*, N.nu; two node pointers, N.parent, and N.nodelink; a set of child nodes. The proposed tree was restructured by updating information with decreased overestimation utilities. Y. Unil and H. Ryang [6] proposed algorithm HUPID-Growth (HUPs in Incremental Databases Growth) for mining HUPs. Authors used a HUPID-Tree, and a restructuring method with a TIList (Tail-node Information List). In the HUPID-Tree, each node

N consisted of N.name, N.max, N.nu, N.parent, N.nodelink, and a set of child nodes. A TIList maintained information of tail nodes in a global HUPID-Tree, and each entry consisted of a link, a support value, and a *tu* value. This algorithm composed of three steps. In the first step, a global HUPID-Tree, TIList were constructed with a single database scan. Then, the tree was restructured by arranging nodes in a *twu* descending order. If new transactions were added to the original database, this tree and the TIList were updated. In the second step, candidate patterns were generated from the tree by the HUPIDGrowth. All HUPs were identified from the extracted candidates in the last step. L. Chun-Wei and et al. [12] [13] presented an algorithm based on pre-large concepts to update discovered HUIs. Itemsets were partitioned into three parts: large, pre-large, or small transaction-weighted utilization in the original database and in inserted transactions. Then, computing procedures were executed for each part. The downward closure property from the two-phase approach was applied to reduce the number of candidate itemsets. Only a small number of itemsets which were less than a threshold must be rescanned. L. Jerry Chun-Wei and et al. [3] proposed a memory-based incremental approach to build utility list structures for mining HUIs. An Estimated Utility CoOccurrence Structure (EUCS) was applied to keep the relationship of 2-itemsets and eliminated the extension itemsets with lower utility. Fournier-Viger and et al. [2] proposed an algorithm EIHI (Efficient Incremental HUI miner) to maintain HUIs in dynamic databases. This algorithm scanned the database to calculate the *twu* of each item. Then, it identified set of items having a *twu* no less than minutil. The *twu* values were used to sort items. A second database scan was performed to collect data for the EUCS structure. All itemsets were stored on a trie-like structure (HUI-trie). Each node represented an item and each itemset was represented by a path starting from the tree root and ending by an inner node or a leaf. Then, the depth-first search exploration of itemsets was performed to find HUIs.

B. *Mining compact HUIs*

Studies on mining compact HUIs have developed to find general HUIs such as closed HUIs, maximal HUIs. Therefore, both time of CPU and memory decrease by avoiding to consider many itemsets. C. W. Wu and et al. [10] proposed the EU-List (Extended Utility-List) to maintain and calculate the utility of itemsets without scanning the original database. Next, a novel algorithm, CHUI-Miner (Closed+ HUI Miner), adopted the divide-and-conquer methodology to mine the complete set of closed HUIs without producing candidates. Structure of EU-list had three fields: Tid, EU and RU. EU, RU indicated the exact utility and the remaining utility of X in T. CHUI-Miner was an extension of DCI-Closed algorithm for mining closed itemsets. For each closed itemset X, it used EU-List to calculate its utility to determine whether it was a closed HUI. Property of remaining utility was used to prune the search space. P. Fournier-Viger and et al. [9] presented EFIM-Closed (EFficient HUI Mining - Closed) based on the strict constraint of each itemset in the search space. EFIMClosed proposed three strategies to discover closed HUIs: closure jumping, forward closure checking, and backward closure checking. EFIM-Closed reduced the cost of database scans based on techniques: High-utility Database Projection and High utility Transaction Merging. Moreover, it used two new upper-bounds on the utility of itemsets named sub-tree utility and local utility to effectively prune the search space, and applied an efficient Fast Utility Counting technique to compute utility of itemsets. P. Fournier-Viger and et al. [7] proposed a novel algorithm named CHUI-Mine (Compact HUI Miner) to discover closed HUIs and maximal HUIs. It was a one-phase algorithm and discovered representations without generating candidates. Authors used the proposed PUDC (Pivot Utility Downward Closure) property to prune the search space. Moreover, an efficient algorithm, RHUI (Recover all HUIs from maximal patterns), was presented to recover all HUIs and their exact utilities from the set of maximal HUIs. This algorithm reused EU list in [10] during mining closed HUIs or maximal HUIs. N.T.T. Loan and et al. [11] presented optimized versions of CHUI-Mine [7] to mine maximal HUIs using P-Set, Estimated Utility Cooccurrence Pruning (EUCP), and First Utility Co-occurrence Structure (FUCS). P-set of itemset X listed identification of transactions containing X. EUCP pruned candidate itemsets based on the EUCS structure. FUCS was a structure to store sum of *twu* of itemsets containing two items. D. Thu-Lan and et al. [8] presented IncCHUI that mined closed HUIs efficiently from incremental databases. They used an utility list structure that built and updated from one database scan. This utility list contained tuples of the form (Tid; iutil; rutil) for each transaction. Next, pruning strategies was applied to increase the speed of construction of utility lists and eliminate candidates. Then, they suggested an hash based method to update or insert new closed sets.

Previous studies show an idea approach for mining HUIs need obtain advantages including without generating candidates, without re-scanning database to compute the utility of itemsets. However, these studies still consume both time of CPU and memory because of processing all items in each transaction. Therefore, we propose a novel algorithm that obtains these advantages and overcome this limit.

IV. Our approach

Our idea is discovered from knowledge of the process of making business decision, the observation of recommendation systems, and properties of transaction databases. We see that decision makers plan business more effectively based on HUIs containing several items. HUIs with a few items is not enough information for decisions. HUIs including many items require decision makers to consider the importance of each items. Moreover, big recommendation systems usually show from 3 to 6 related items of the current item. Additionally, most HUIs in transaction databases contain several items and a few HUIs have many items. Therefore, we propose a fast effective algorithm for mining k-item HUIs with small number of items. The proposed algorithm is called kHUI-Miner (mining k-item HUIs). We introduce a simple list to store k-itemsets containing in databases. This list is called Items-Utility list (IU list). Our algorithm includes two main steps. Firstly, a vertical segmentation of the current database is performed to form sub-partitions (SPs) based on the value of *k*. Each row in a SP contains *k* items. Next, our

algorithm mines k-itemsets from each SP and stores them in a global IU list. Then, k-item HUIs are extracted from the IU list based on the utility of itemsets. We explain our approach more detail in next subsections.

A. *Structure of the IU list*

We propose the IU list to store k-itemsets appearing from the current database. Structure of this list includes items, utility of itemsets. Table 3 shows an example of the IU list.

Items	Utility
2, 3, 5, 6	404
1, 3, 4, 5	176

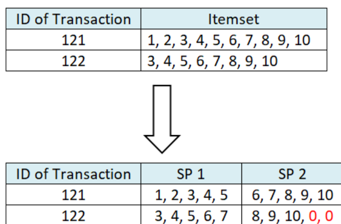
Table 3. An example of the IU list with k = 4.

The IU list is added or updated during browsing transactions. When a new itemset appears, a new row is added. If an itemset exists then updating its utility. Structure of the IU list is simple and only contains two most important information for mining HUIs. Therefore, the IU list uses a small capacity of memory and helps to decrease time for searching existing itemsets.

B. *The proposed algorithm for mining k-item HUIs*

Problem Statement: Given a DB and a utility threshold *minutil*, mining k-item HUIs from DB discovers all k-itemsets whose utilities are greater than or equal to *minutil*.

Firstly, we perform a vertical segmentation for the current database. This database is divided into SPs whose rows contain *k* items. Therefore, we need choose a proper value of *k*. We observe big recommendation systems in retail. We see that an item usually links to from 3 to 6 related items. Therefore, we recommend value of *k* that is from 4 to 7. As a result, items in a transaction belong to many SPs. *k* first items are in SP 1 and *k* next items are in SP 2. Repeat until the last item are assigned to a SP. If number of items of the last SP is not enough *k* items then adding default items. For example, adding item 0 to last SP. Figure 1 shows an example for segmentation of a database with k = 5.



ID of Transaction	Itemset
121	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
122	3, 4, 5, 6, 7, 8, 9, 10

ID of Transaction	SP 1	SP 2
121	1, 2, 3, 4, 5	6, 7, 8, 9, 10
122	3, 4, 5, 6, 7	8, 9, 10, 0, 0

Figure 1. Segmentation of a small database with k = 5.

Exploring transactions in each SP is performed based on the top-down strategy. kHUI-Miner finds an itemset that appears in adjacent transactions. Assuming, a k-itemset X appears in transactions from *m* to *n*. We formulate the utility of X in transactions from *m* to *n* and store in an available. After browsing a row, this available is added to the utility of X in the previous row. Then, searching X from the IU list. If X exists then updating the utility of X. Otherwise, adding a new row to the IU list to store items and utility of X. Repeat this process until browsing the last row of each SP. We only use an IU list. As a result, a global IU list stores itemsets appearing in all SPs.

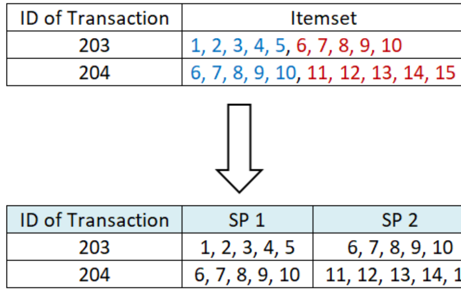
When adding new transactions from an incremental database, the IU list is continuously added or updated. Meaning, we combine data of both the original database and incremental databases in an IU list. Therefore, kHUI-Miner still finds new HUIs without re-scanning the original database. Moreover, the proposed algorithm still outputs HUIs without rescanning mined databases when changing *minutil*. Similarly, mining HUIs from a incremental database is done. kHUI-Miner is applied for both a original database and incremental databases. Steps of the proposed algorithm are described as follow

Algorithm 1: *Mining k-item HUIs*

Input: Given a database DB
minutil - the threshold of utility
k - the number of items of an itemset
P - a sub-partition, L - the IU list, T - a transaction
X - an itemset appearing in adjacent transactions
Y - an itemset appearing in the current row
Z - an itemset appearing in L
Output: k-item HUIs.
1: Divide DB into SPs with the parameter *k*
2: for each P in DB do
3: Set X to the itemset in the first transaction of P
4: for each $T \in P$ do
5: if X = Y then
6: Update the utility of X
7: end if
8: if $X \neq Y$ then
9: if $X \notin L$ then
7: Add X to L
8: else
9: Update the utility of X in L.
10: end if
11: X=Y
12: for each $Z \in L$ do
13: if Z.Utility $\geq$ *minutil* then
14: Z is a k-item HUI.
15: end if

We mine all k-item HUIs by one of two following ways:

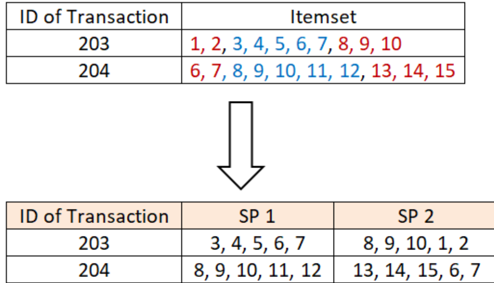
- Choose a proper value of *k* to obtain the highest ratio of number of k-item HUIs and number of k-itemsets. Additionally, both time of CPU and memory are accepted for real applications. We explain more detail in experiments.
- Perform two times of this algorithm on two ways of vertical segmentation of databases when giving a value of *k*. Assuming that we choose k=5 for an example database. The first way for segmenting database is done in Fig. 2 and Fig. 3 shows the second one of segmenting database. As a result, k-itemsets appearing between two adjacent SPs of the first way are mined.



ID of Transaction	Itemset
203	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
204	6, 7, 8, 9, 10, 11, 12, 13, 14, 15

ID of Transaction	SP 1	SP 2
203	1, 2, 3, 4, 5	6, 7, 8, 9, 10
204	6, 7, 8, 9, 10	11, 12, 13, 14, 15

Figure 2. The first way for segmenting database with k = 5.



ID of Transaction	Itemset
203	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
204	6, 7, 8, 9, 10, 11, 12, 13, 14, 15

ID of Transaction	SP 1	SP 2
203	3, 4, 5, 6, 7	8, 9, 10, 1, 2
204	8, 9, 10, 11, 12	13, 14, 15, 6, 7

Figure 3. The second way for segmenting database with k = 5.

C. *Discussion*

Our approach obtains following advantages

- Mine on a small part of databases to drop memory and avoid generation of candidate itemsets.
- Mine new HUIs without re-scanning mined databases when changing *minutil*.
- Propose a simple list to decrease memory consumption and time for searching.
- Apply for both a original database and incremental databases.
- Be simple to understand and perform.

As a result, kHUI-Miner significantly decreases both time and memory for computing. Moreover, the proposed algorithm is suitable for real applications which require online analytical processing. We conduct experiments to prove the effectiveness of our algorithm.

V. Experiments

Experiments are conducted on databases with varying *minutil* value to find a proper value of *k*. Influence of the insertion ratio and scalability tests are done to evaluate the performance of our algorithm. kHUI-Miner is compared to state-of-the-art algorithms including EFIM-Closed [9], and CHUIMiner [10]. EFIM-Closed presents three pruning strategies, and uses novel utility upper bounds to mine closed HUIs. CHUIMiner adopts the divide-and-conquer methodology to mine the complete set of closed HUIs without generating candidates.

A. *Experiment setup*

We performed experiments on benchmark databases having various characteristics. Three dense databases were used in experiments including *mushroom, chess, connect*. They were downloaded from FIMI Repository [14]. Table 4 presents characteristics of these datasets, where #Trans, #Item, Length indicate the number of transactions, the number of distinct items, and the transaction length respective.

Database	#Trans	#Item	Length
Mushroom	8124	119	23
Chess	3196	75	37
Connect	67557	129	43

Table 4. Statistical information about dense databases.

Most databases do not provide item utility (external utility) and item count for each transaction (internal utility). Like some previous studies [8], external utilities for items are generated between 0.01 and 10 by using a log-normal distribution and internal utilities for items are randomly generated from 1 to 10. All algorithms were implemented in Java, where EFIMClosed, CHUIMiner were obtained from SPMF [15]. We conducted experiments on a computer equipped with a 64 bit, Core i3 (2GHz x 2GHz) Intel Processor, 4GB of main memory, and running Windows 10. Each data is an average of 5 experiment results. Time of CPU consists of time for inputting transactions and mining HUIs.

B. *Finding a proper value of k for each database*

We select *k* from 4 to 7 to run experiments. Time of CPU and memory consumption of kHUI-Miner are measured to determine the proper value of *k* for each database. For *mushroom* and *chess*, we use all transactions. For *connect*, we use transactions from 50000 to 67557 to mine the newest transactions.

1) *Results on mushroom*

Table 5, 6, and 7 show number of k-item HUIs, CPU's time, and memory of experiments on *mushroom*. The last row of Table 5 presents the number of k-itemsets at each value of *k*.

minutil	k=4	k=5	k=6	k=7
500	264	295	164	56
400	280	305	167	56
300	289	313	168	56
200	296	319	169	56
100	305	335	3353	56
#k-itemset	316	341	8291	56

Table 5. Number of k-item HUIs of *mushroom*.

minutil	k=4	k=5	k=6	k=7
500	1499.4	1892.8	29561.4	578.0
400	1781.0	1849.6	29514.8	574.6
300	1462.0	1843.2	29609.8	543.4
200	1478.0	1852.4	29558.6	518.8
100	1487.4	1908.8	29365.2	528.0

Table 6. Time of CPU (ms) of *mushroom*.

We see that *k*=5 is the proper value because the number of k-item HUIs is 335 which is of 341 k-itemsets at the lowest value of *minutil*. Moreover, its time of CPU is less than 2000 (ms) and its memory consumption is less than 0.36 MB. With *k*=4, time for computing is less than about 400 (ms) but the number of k-item HUIs is lower. As a result, some k-item HUIs is not mined. With *k*=6, generating many k-itemsets and CPU's time is significantly higher (about 30000 ms). With

k=7, obtaining a small number of k-item HUIs.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
500	0.35268	0.35326	0.35279	0.35271
400	0.35288	0.35276	0.35274	0.35268
300	0.35309	0.35290	0.35296	0.35271
200	0.35274	0.35268	0.35314	0.35268
100	0.35268	0.35288	0.35295	0.35271

Table 7. Memory consumption (MB) of *mushroom*.

2) Results on chess

Table 8, 9, and 10 show data of experiments on *chess*. We see that *k*=7 is the best value because the number of k-item HUIs is 231 which is of 231 k-itemsets at the lowest value of *minutil*. Moreover, both its time of CPU and its memory consumption are lower than other values of *k*.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
250	88	121	115	186
200	93	123	118	196
150	95	129	127	206
100	101	136	208	219
50	106	142	2813	231
#k-itemset	108	143	3242	231

Table 8. Number of k-item HUIs of *chess*.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
250	606.4	634.2	4964.6	575.0
200	718.8	696.8	4561.4	587.4
150	637.2	672.0	4798.4	584.4
100	674.8	624.8	4708.4	615.6
50	678.2	565.4	4571.2	584.2

Table 9. Time of CPU (ms) of *chess*.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
250	1.13604	1.39722	1.39083	0.35267
200	1.65831	1.65834	0.61221	0.35270
150	1.65827	1.65833	0.87181	0.35267
100	1.39716	1.13608	0.87175	0.35267
50	1.65827	0.87496	0.61221	0.35267

Table 10. Memory consumption (MB) of *chess*.

3) Results on connect

Table 11, 12, and 13 show data of experiments on *connect*.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
300	229	298	224	461
250	233	310	227	480
200	236	316	238	511
150	245	335	391	540
100	252	347	7159	565
#k-itemset	264	368	17811	605

Table 11. Number of k-item HUIs of *connect*.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
300	1965.4	2809.0	129392.8	4118.0
250	1930.8	2887.0	129599.6	4096.0
200	1946.8	2740.0	130592.8	4070.8
150	1977.6	2805.4	129778.2	4084.0
100	1949.8	2808.6	130612.0	4039.6

Table 12. Time of CPU (ms) of *connect*.

minutil	**k=4**	**k=5**	**k=6**	**k=7**
300	26.83866	26.83872	0.35337	27.21266
250	26.83864	26.83867	0.35331	27.21263
200	26.83861	26.83870	0.35331	21.84076
150	26.83864	26.83867	0.35331	27.21263
100	26.83861	26.83867	0.35329	27.21263

Table 13. Memory consumption (MB) of *connect*.

We see that *k*=4 might be the proper value because the number of k-item HUIs is 252 which is of 264 k-itemsets at the lowest value of *minutil*. Moreover, its time of CPU is the lowest and its memory consumption is higher than results of *k*=6. With *k*=6, generating two many k-itemsets (17811) and CPU's time is sharply higher (126000 ms) than *k*=5. However, recommendation systems usually introduce from 5 to 7 related items. Therefore, *k*=5 is the best value because the number of k-item HUIs is 347 which is of 368 k-itemsets at the lowest value of minutil. Its time of CPU is slightly higher (800ms) than *k*=4 and its memory consumption is equal to results of *k*=4.

C. Influence of the insertion ratio

We setup *minutil*=10 and use the best value of *k* on each database to conduct all experiments. Selected insertion ratios are 10%, 15 %, 20%, and 25%. Data of each insertion ratio are different because we select 10% first transactions, 15% next ones, 20% next ones, and 25% next ones. Meaning, 70% first transactions are used to run experiments. For *mushroom*, 800 transactions are 10% of transactions. Similarly, 300 transactions are 10% of *chess* and 2000 transactions are 10% of *connect* (running from the 50000-th transaction). Table 14, 15, and 16 present results of *mushroom, chess*, and *connect*. Table 17 gives more information of results on databases at 25% transactions. For small databases, *mushroom* uses about 572 ms and 0.35 MB and *chess* uses about 522 ms and 1.66 MB. For larger database, *connect* uses about 2324 ms and 27.2 MB. We see that both the maximum of time and the maximum of memory are small. Therefore, kHUI-Miner is effective for application that need online analytical processing.

Insertation ratio	**#k-item HUI**	**Time (ms)**	**Memory (MB)**
10%	110	374.8	0.35275
15%	128	425.2	0.35295
20%	125	440.2	0.35278
25%	183	571.8	0.35275

Table 14. Influence of the insertion ratio on *mushroom*.

Insertation ratio	**#k-item HUI**	**Time (ms)**	**Memory (MB)**

10%	46	321.6	1.39737
15%	70	346.6	1.65855
20%	96	406.0	1.65858
25%	152	521.8	1.65858

Table 15. Influence of the insertion ratio on *chess*.

Insertation ratio	#k-item HUI	Time (ms)	Memory (MB)
10%	212	1515.2	16.47384
15%	214	1671.2	11.09711
20%	308	1715.4	5.72522
25%	443	2324.4	27.21266

Table 16. Influence of the insertion ratio on *connect*.

Database	#Trans	#Item	Length	Time	Memory
Mushroom	8124	119	23	571.8	0.35275
Chess	3196	75	37	521.8	1.65858
Connect	67557	129	43	2324.4	27.21266

Table 17. Results of 25% transactions of each database.

D. Scalability tests

We setup *minutil*=10 and use the best value of *k* on each database for all experiments. Number of transactions are 20%, 40%, 60%, and 80%. For *mushroom*, data are selected from the first transaction to 1600, 3200, 4800, 6400. For *chess*, selected transactions are from 1 to 600, 1200, 1800, 2400. For *connect*, they are 50001..54000, 50001..58000, 50001..62000, 50001..66000. Table 18, 19, and 20 show scalability results of *mushroom, chess,* and *connect*.

Scalability	#k-item HUI	Time (ms)	Memory (MB)
20%	121	450.0	0.35297
40%	161	668.8	0.35281
60%	226	990.4	0.35295
80%	308	1655.8	0.35289

Table 18. Scalability tests of *mushroom*.

Scalability	#k-item HUI	Time (ms)	Memory (MB)
20%	71	418.8	1.39745
40%	109	503.2	1.65860
60%	146	631.2	0.61388
80%	185	702.6	0.35274

Table 19. Scalability tests of *chess*.

For *mushroom*, 80% transactions use about 1656 ms and 0.35 MB. CPU’s time increases slightly (from 450 to 1666) and memory is about 0.35 MB. For *chess,* the maximum of time is 702 ms and the maximum of memory is 1.66 MB. For *connect*, the maximum of time is 4415 ms and the maximum of memory is 27.2 MB. We also see that both the maximum of time and the maximum of memory are much smaller than hardware and performance of modern computers. Therefore, the proposed algorithm is suitable for online analytical processing.

Scalability	#k-item HUI	Time (ms)	Memory (MB)
20%	273	1721.6	0.35334
40%	337	2439.0	27.21269
60%	468	3313.6	27.21266
80%	573	4415.0	27.21269

Table 20. Scalability tests of *connect*.

E. Compare to state-of-the-art algorithms

We choose *minutil* for kHUI-Miner. Values of *minutil* are sharply smaller to find number of k-item HUIs that is approximately equal to number of compact HUIs of EFIM-Closed and CHUIMiner. Experiments are conducted on *mushroom* to limit the running time. Our algorithm uses *minutil*=10 while EFIM-Closed and CHUIMiner setup *minutil*=1000. Meaning, *minuitl* in kHUI-Miner is lower 100 times than compared algorithms.

1) Results of influence of the insertion ratio

Table 21 and 22 present CPU’s time and memory consumption of algorithms to measure the influence of the insertion ratio. Data from Table 21 show that kHUI-Miner is the best algorithm. CPU’s time of it is lower 4 times than EFIMClosed (the second algorithm) at 25% transactions. Similarly, Table 22 shows the proposed algorithm is better than compared algorithms. kHUI-Miner’s memory is lower about 50 times than CHUI-Miner (the second one) at 25% transactions.

Insertation ratio	EFIM-Closed	CHUIMiner	kHUI-Miner
10%	725.2	472.0	**374.8**
15%	2764.6	2536.8	**425.2**
20%	1288.8	1624.6	**440.2**
25%	2605.8	2802.4	**571.8**

Table 21. Time of CPU when changing the insertion ratio of *mushroom*.

Insertation ratio	EFIM-Closed	CHUIMiner	kHUI-Miner
10%	42.86039	25.00209	**0.35275**
15%	74.50662	38.47234	**0.35295**
20%	55.99478	33.74635	**0.35278**
25%	68.96075	52.36323	**0.35275**

Table 22. Memory consumption when changing the insertion ratio of *mushroom*.

2) Results on scalability tests

Table 23 and 24 present CPU’s time and memory consumption of algorithms to evaluate scalability. Table 23 shows that our approach is the best algorithm and significantly decreases the time of CPU. Similarly, kHUI-Miner is better than compared methods in Table 24. Especially, CPU’s time of our algorithm sharply decreases.

Data from Table 21-24 show that kHUI-Miner is the better than state-of-the-art methods. Both time and memory for computing are smaller many times than both hardware and performance of modern computers. Therefore, our algorithm is effective for applications running online analytic processing.

Scalability	EFIM-Closed	CHUIMiner	kHUI-Miner

20%	1599.6	1146.2	**450.0**
40%	4108.4	3973.8	**668.8**
60%	7864.0	11437.6	**990.4**
80%	9554.0	19669.4	**1655.8**

Table 23. Time of CPU on scalability tests of *mushroom*.

Scalability	EFIM-Closed	CHUIMiner	kHUI-Miner
20%	67.03810	36.10468	**0.35297**
40%	134.23781	42.78154	**0.35281**
60%	160.74933	60.13595	**0.35281**
80%	143.89967	92.38238	**0.35289**

Table 24. Memory consumption on scalability tests of *mushroom*.

VI. Conclusion

In this paper, we propose a novel fast algorithm for mining k-item HUIs that meets the need of business activities and improves the performance of mining HUIs. We use a simple list structure to store information during browsing transactions. The proposed algorithm for mining k-itemHUIs consists of two main steps. First, the current database is vertically segmented into subpartitions. Then, k-item HUIs are mined from each sub-partition. Our approach obtains advantages including without candidate generation, without re-scanning database when changing the threshold of utility. Experiments are conducted on dense benchmark datasets including *mushroom, chess*, and *connect*. Experiment results show that our algorithm is better than state-of-the-art methods.

We will investigate to optimize the time for computing and conduct experiments on larger databases in the future.

References

[1] Y. Unil, H. Ryang, L. Gangin, H. Fujita. “An efficient algorithm for mining high utility patterns from incremental databases with one database scan”, *KnowledgeBased Systems*, 124, pp. 188-206, 2017.

[2] P. Fournier-Viger, L.C. Jerry, T. Gueniche, P. Barhate. ”Efficient Incremental High Utility Itemset Mining”. In *Proceedings of ASE International Conference on Big Data*, 2015.

[3] L.Jerry Chun-Wei, G. Wensheng, H. Tzung-Pei, P. Jeng-Shyang. “Incrementally Updating High-Utility Itemsets with Transaction Insertion”, in *Procedding of Advanced Data Mining and Applications*, pp. 44-56, 2014.

[4] L. Judae, Y. Unil Yun, L. Gangin , Y. Eunchul. “Efficient incremental high utility pattern mining based on pre-large concept”, *Engineering Applications of Artificial Intelligence*, 72, pp. 111–123, 2018.

[5] R. Heungmo, Y. Unil. “High utility pattern mining over data streams with sliding window technique”, *Expert Systems With Applications*, 57, pp. 214–231, 2016.

[6] Y. Unil, H. Ryang. “Incremental high utility pattern mining with static and dynamic databases”, *Applied Intelligence*, 42, pp. 323–352, 2015.

[7] P. Fournier-Viger, L. Jerry Chun-Wei, N. Roger, Bay Vo, V. S. Tseng. “Mining Compact High Utility Itemsets Without Candidate Generation”, *High-Utility Pattern Mining: Theory, Algorithms and Applications*, 51, pp. 282-307, 2019.

[8] D. Thu-Lan, R. Heri, N. Kjetil, D. Quang-Huy, “Towards efficiently mining closed high utility itemsets from incremental databases”, *Knowledge-Based Systems*, vol. 165, pp. 13-29, 2019.

[9] P. Fournier-Viger, S. Zida, J. C.W. Lin, C.W. Wu, V. S. Tseng. ”EFIM-Closed: Fast and memory efficient discovery of closed high-utility itemsets”. In *Proceedings of Machine Learning and Data Mining in Pattern Recognition*, pp. 199-213, 2016.

[10] C. W. Wu, P. Fournier-Viger, J. Y. Gu, V. S. Tseng. “Mining closed+ high utility itemsets without candidate generation”, In *Proceedings of Conference on Technologies and Applications of Artificial Intelligence*, pp. 187-194, 2015.

[11] N.T.T. Loan, V.D. Bao, N.D.D. Trinh, V. Bay. ”Mining Maximal High Utility Itemsets on Dynamic Profit Databases”, *Cybernetics and Systems*, 51 (2), pp. 140- 160, 2020.

[12] L. Chun-Wei, G. Wensheng, H. Tzung-Pei, Z. Binbin. ”An Incremental High-Utility Mining Algorithm with Transaction Insertion”, *The Scientific World Journal*, 2015.

[13] L. Chun-Wei, H. Tzung-Pei, L. Guo-Cheng, W. JiaWei, L. Wen-Yang. ”Incrementally mining high utility patterns based on pre-large concept”, *Applied Intelligence*, 40, pp. 343–357, 2014.

[14] Frequent Itemset Mining Dataset Repository. http:// fimi.ua.ac.be/, 2012.

[15] P. Fournier-Viger, A. Gomariz, A. Soltani, H. Lam, T. Gueniche. ”Spmf: Open-source data mining platform”, http://www.philippe-fournier-viger.com/spmf, 2014.

Author Biographies

Nong Thi Hoa
Nong Thi Hoa received her BS degrees in Information Technology from University of Engineering and Technology, Vietnam National University, Ha Noi, Vietnam in 2000. She received her MS degrees in Computer Science from Thai Nguyen University of Information and Communication Technology, Vietnam in 2006. She received her PhD degree in Computer Science from University of Engineering and Technology, Vietnam National University, Ha Noi, Vietnam in 2015. Her major research interests include pattern recognition, classification, clustering, and mining high utility itemset.

Nguyen Van Tao
Nguyen Van Tao received his BS degrees in Math from Thai Nguyen University of Education, Vietnam in 1994. He received his MS degrees in Computer Science from University of Science, Vietnam National University, Ha Noi, Vietnam in 1999. He received his PhD degree in Guarantee of mathematics for computers and calculation systems from Institute of Information Technology, Vietnam Academy of Science and Technology, Vietnam in 2009. His major research interests include image processing, safety and security of information.