Acta Cybernetica **20** (2011) 331–346.

Mining High Utility Itemsets in Massive Transactional Datasets*

Vu Duc Thi[†] and Nguyen Huy Duc[‡]

Abstract

Mining High Utility Itemsets from a transaction database is to find itemsets that have utility beyond an user-specified threshold. Existing High Utility Itemsets mining algorithms suffer from many problems when being applied to massive transactional datasets. One major problem is the high memory dependency: the gigantic data structure built is assumed to fit in the computer main memory. This paper proposes a new disk-based High Utility Itemsets mining algorithm, which achieves its efficiency by applying three new ideas. First, transactional data is converted into a new database layout called Transactional Array that prevents multiple scanning of the database during the mining phase. Second, for each frequent item, a relatively small independent tree is built for summarizing co-occurrences. Finally, a simple and non-recursive mining process reduces the memory requirements as minimum candidacy generation and counting is needed. We have tested our algorithm on several very large transactional databases and the results show that our algorithm works efficiently.

Keywords: High Utility Itemset Mining, COUI-tree

1 Introduction

A framework for high utility itemset mining was proposed recently by Yao et al (H. Yao and H. J. Hamilton, 2006) [6]. In this, the value of one item is a number (the quantity of the sold item, we can call it an objective value), otherwise, it has a utility table that contains utility of all items in the dataset (we can call it a subjective value, determined by manager). Utility of a itemset is the sum of all utility of all items in that itemset. The high utility itemset mining problem is to

*This work was funded by the Vietnam's National Foundation for Science and Technology Development (NAFOSTED) via a research grant for fundamental sciences, grant number: 102.01-2010.09

[†]Institute of Information Technology. Viet Namese Academy of Science and Technology., E-mail: vdthi@ioit.ac.vn

[‡]Faculty of Information and Computer, National Training College for Teachers, Ha Noi, Viet Nam., E-mail: ducnghuy@yahoo.com

DOI: 10.14232/actacyb.20.2.2011.6

find all itemsets that have utility larger than a user specified value of minimum utility.

In [6], H. Yao and H. J. Hamilton proposed a mining method and described pruning strategies based on the mathematical properties of utility constraints. They also developed an algorithm named Umining and another heuristic based algorithm Umining_H to discover high utility itemsets.

Recent research has focused on efficient high utility mining algorithms using intermediate anti-monotone measures for pruning the search space. In [7], Liu et al. (Y. Liu, Liao, & Choudhary, 2005) propose a two phase algorithm to mine high utility itemsets. They use a transaction weighted utility (TWU) measure in the first phase to find the supersets of high utility itemsets, followed by a rescan of the database to determine the actual high utility itemsets among them. However, their algorithm is based on the candidate generation-and-test approach and so suffers from poor performance when mining dense datasets and long patterns in the same way as the Apriori algorithm for frequent pattern mining.

In this paper, we propose an efficient algorithm for utility mining in massive datasets. This algorithm rearranges database and saves it the external memory, in mining process only a small part of data is put into the internal memory and mining is based on the idea of COFI-tree algorithm by Mohammad El-Hajj and Osmar R. Zaiane presented in 2003. After all data is rearranged and stored in the external memory, we can mine high utility itemsets with a different threshold without reorganizing the database.

The rest of the paper is organized as follows: In Section 2, we define the relevant terms. Section 3 summarizes the COFI-tree algorithm used in mining frequent patterns. Section 4 describes our new algorithm for mining high utility itemsets in large datasets. The performance studies of the algorithm are given in Section 5. Section 6 contains the conclusions of the paper.

2 High Utility Itemset Mining

In this Section, we give the basic notations and the definitions of terms to describe high utility itemset mining, based on (H. Yao and H. J. Hamilton, 2006) [6]. Let $I = \{i_1, \ldots, i_n\}$ be a set of items. A transaction T is a subset of $I, T \subseteq I$. $DB = \{T_1, \ldots, T_m\}$ is a transaction database. Each transaction is assigned by an ID called TID. A subset $X \subseteq I$ which contains k different items is called k-itemset. Transaction T contains X if $X \subseteq T$.

Definition 1. *The value of item i_p in transaction T_q (at column i_p row T_q of database) is an objective value denoted as (i_p, T_q).*

Definition 2. *Calling value, which is assigned by manager for item i_p in database, based on estimating utility gaining from one unit of that item is a called subjective value denoted as $s(i_p)$.*

Normally, the subjective value is given in a table called the utility table. For example, in Table 1 and 2, the objective value of item B at transaction T_2 is $o(B, T_2) = 12$, the subjective value of item B is $s(B) = 5$.

Definition 3. *Let x be an objective value and y be a subjective value of one item. Function $f(x, y) : R \times R \longrightarrow R$ is called the utility function calculated as follow: $f(x, y) = x \times y$.*

Table 1: Transactional database

TID	A	B	C	D	E
$T1$	0	12	2	0	2
$T2$	0	12	0	2	1
$T3$	2	0	1	0	1
$T4$	1	0	0	2	1
$T5$	0	0	4	0	2
$T6$	1	2	0	0	0
$T7$	0	20	0	2	1
$T8$	3	0	25	6	1
$T9$	1	2	0	0	0
$T10$	0	0	16	0	1

Table 2: Utility table

Item	Profit ($/unit)
A	3
B	5
C	1
D	3
E	5

Definition 4. *Let $f(x, y)$ be an utility function. The utility of item i_p in transaction T_q (denoted as $u(i_p, T_q)$) is the value of $f(x, y)$ at $o(i_p, T_q)$ and $s(i_p)$, that is $u(i_p, T_q) = f(o(i_p, T_q), s(i_p))$.*

Definition 5. *Let X be an itemset in transaction T_q. Utility of X in transaction T_q, denoted as $u(X, T_q)$, is defined as: $u(X, T_q) = \sum_{i_p \in X \subseteq T_q} u(i_p, T_q)$.*

An itemset X has an associated set of transactions in DB, denoted as db_X , where $db_X = \{T_q : X \subseteq T_q, T_q \in DB\}$.

Definition 6. *Utility of itemset X in database DB, denoted as $u(X)$, is utility sum of X itemset at all transactions of db_X, that is: $u(X) = \sum_{T_q \in db_X} u(X, T_q)$ $= \sum_{T_q \in db_X} \sum_{i_p \in X} u(i_p, T_q)$.*

For example, in Table 1 and 2, $u(B,T_2) = 12 \cdot 5 = 60$. Consider $X = \{B, D\}$, $u(X,T_2) = u(B,T_2)+u(D,T_2) = 12\cdot 5+2\cdot 3 = 66$, there are two transactions T_2 and T_7 which contain itemset X, so $db_X = \{T_2,T_7\}, u(X) = u(X,T_2) + u(X,T_7) = 172$.

Definition 7. *Given a minutil (> 0) and a itemset X. X is called high utility itemset if $u(X) \geq minutil$; otherwise, X is called low utility itemset.*

Definition 8. *Given a transaction database DB and a minutil. The problem of mining high utility itemsets is to find HU set such that it contains all high utility itemsets, i.e.:*

$$HU = \{X : X \subseteq I, u(X) \geq minutil\}.$$

The problem of mining frequent itemsets can be seen as a special case of mining high utility itemsets when all items have the objective value of 0 or 1 and subjective value of 1. The main property used for mining frequent itemsets is Apriori. The Apriori property states that all nonempty subsets of a frequent itemset must also be frequent. It is not hard to see that this property is not correct in the case of utility. For example, in database of Table 1, we have, $u(BC) = 62 < 72 = u(BCE)$, while $u(BC) = 62 > 0 = u(BCD)$. The following section will present the fundamental idea of IM algorithm [9] for mining frequent itemsets using COFI-tree structure.

3 Mining frequent itemsets based on the structure of COFI-tree.

In 2003, Mohammad El-Haj and Osmar R. Zaiane in Department of Computing Science University of Alberta Edmonton, AB, Canada proposed IM (Inverted Matrix) algorithm [9] for mining frequent itemsets in large databases.

IM algorithm can be divided into two phases:

Phase 1: (pre-processing) It rearranges data into matrix and saves this matrix in the external memory.

Phase 2: This phase is mining matrix by using COFI-tree (Co-Occurrence Frequent Item Tree) for each item [8].

In the first phase, the Inverted Matrix is a disk-based data layout made of two parts: the index and the transactional arrays. The index contains the items and their respective frequencies. The transactional array is a set of rows in which each row is associated with one item in the index part. Each row is made of a pairs of pointers holding following information: the physical address in the index part of the next item in the same transaction, and the physical address in the row of the next item in the same transaction. Building the Inverted Matrix is accomplished in two passes of the database during the pre-processing phase. The first pass scans the whole database to find the frequency of each item. The item list is then ordered in ascending order according to their frequency. The second pass reads each transaction from the database and also orders it into ascending order based on the frequency of each item. In the index part, the location of the first item in the transaction is sought and an entry to its transactional array is added that holds the location of the next item in this transaction. For the second item, the same process is applied, in which an entry in the transactional table of the second item is added to hold the location of the third item in the transaction. The process is then repeated for all items in this transaction. The following transaction is read next and the same applies to all of its items. This process repeats for all transactions in the database.

In the second phase, it mines the data matrix (transactional array) by using the structure of COFI-tree. It traverses the index part and ignores all non-frequent items, with each frequent item, it reads all transactions that contain the items and build a COFI-tree for this item, after that it mines all frequent itemsets in this tree. The trees are discarded as soon as mining ends and exactly the same process is repeated for other items.

COFI-tree of one item is a tree constructed by this item and all the others that have frequencies equal or greater than of that item. Each tree has a header table which contains a collection of frequent items, these items in header table are also sorted in ascending order of their frequency. Each entry in the header table have three data fields: item's name, local frequency in the COFI-tree and a pointer pointing to the first and the same item in the tree. A link list is maintained between all positions of those items in the tree. Each node of COFI-tree contains 4 data fields: item's name, S (it's support), P (it's participation, this field keeps track of how many times this item participates in a candidate generation), pointers pointing to the next same label node or null if not. More details of the algorithm could found in [8, 9].

4 Mining high utility itemsets in large dataset

Liu et al. (Y. Liu, Liao, & Choudhary, 2005) proposed the concepts of Transaction Utility (TU) and Transaction Weighted Utility (TWU) to prune the search space for mining high utility itemsets. Transaction Utility of a transaction, denoted $tu(T_q)$ is the sum of the utilities of all items in T_q, $tu(T_q) = \sum_{i_p \in T_q} u(i_p, T_q)$. Transaction Weighted Utility of an itemset X, denoted as $twu(X)$ is the sum of the transaction utilities of all the transactions containing X, $twu(X) = \sum_{T_q \in DB \wedge X \subseteq T_q} tu(T_q)$.

For example, in Table 1 and 2, $tu(T_2) = 12 \cdot 5 + 2 \cdot 3 + 1 \cdot 5 = 71$.

Let $X = DE$, $db_X = \{T_2, T_4, T_7, T_8\}, twu(X) = tu(T_2) + tu(T_4) + tu(T_7) + tu(T_8) = 253$.

Note: $u(X, T_q) \leq tu(T_q)$

$\Rightarrow u(X) = \sum_{T_q \in DB \wedge X \subseteq T_q} u(X, T_q) \leq \sum_{T_q \in DB \wedge X \subseteq T_q} tu(T_q) = twu(X)$.

Consider $twu(X)$ as the upper bound of $u(X)$. If X is a high utility itemset for a threshold $minutil$, then X is also a high utility TWU because $twu(X) \geq u(X) > minutil$. Vice versa, if X is not a high utility TWU then X is also not a high utility itemset.

TWU-utility constraint has anti-monotone property [7], i.e.: All itemsets that contain a low utility TWU itemset ($twu(X) < minutil$) is a low utility itemset. So, if X is a low utility TWU itemset ($twu(X) < minutil$), X and all itemsets that contain it are low utility itemsets, and could be removed while mining high utility itemsets.

Based on this idea, we propose a new COUI-Mine algorithm (Co-Occurrence Utility Item Mine) for mining high utility itemsets in large datasets. This algorithm can be divided into 2 phases:

Phase 1: Construct transactional array and saves it in the external memory.

Phase 2: Mining high utility itemsets by using the structure of COUI-tree.

4.1 Construct transactional array:

The algorithm separates disk-based data into two parts: the index and the transactional array. Each entry in index part contains 5 data fields: item's name, quantity, profit of one unit, frequency and its TWU. In this part, items are sorted in ascending order of their frequencies. The transactional array is a set of rows in which each row is associated with one item in the index part. Each element in the transactional array stores 4 data fields : quantity in transaction, TU of transaction and location [row, column] of the next item in this transaction. If that item is the last element in a transaction, its location should be empty.

In the first scanning of the database, we calculate transaction utility, total quantity, frequency, TWU of each item. Sort the items in ascending order of frequencies and build the index part.

During the second scanning of the database, we sort each transaction, in ascending order of item's frequency and put it into transactional array as follows:

Based on the Index part we determine the position of the first item. Then in the item's row, we find the first empty place (cell) and save item's information here. Then the location of next item is determined in exactly the same way as it was done for the first item (note that this location will be stored in the cell of the first item). We repeat this for all items in the transaction, in the cell of the last item the location fields are empty.

This transactional array is constructed and saved in the external memory.

The Tables below are used for the demonstration of our algorithm.

To give an example, suppose we have a database in Table 1 and 2, threshold = 30% (of total utility), and *minutil* = 30% x 398 = 119,4.

The algorithm scans the database for the first time, calculates the transaction utility (in Table 3), total quantity, frequency, and TWU of each item (in Table 4). It then sorts items in ascending order of frequencies and builds the index part (Table 5).

The algorithm then scans the database for the second time, and for each transaction, it sorts the items into ascending order of frequencies and put them into the transactional array. Table 6 illustrates the sorted transactions. In case of transaction $T1 = (B : 12, C : 2, E : 2)$ the search in the index part gives that B is in position 3, C is in position 4 and E is in position 5, so the 3 blocks which are used to save the information of these items are in row 3, row 4 and row 5 of the transactional array. First we find the first empty block in row 3 and we have [3,1], this block will save B's information and the address (or location) of C as well. The first

Table 3: Utility of transactions of database in table 1 and 2

TID	A	B	C	D	E	*tu*
$T1$	0	12	2	0	2	*72*
$T2$	0	12	0	2	1	*71*
$T3$	2	0	1	0	1	*12*
$T4$	1	0	0	2	1	*14*
$T5$	0	0	4	0	2	*14*
$T6$	1	2	0	0	0	*13*
$T7$	0	20	0	2	1	*111*
$T8$	3	0	25	6	1	*57*
$T9$	1	2	0	0	0	*13*
$T10$	0	0	16	0	1	*21*
Sum	8	48	48	12	10	398

Table 4: Quantity, twu and Frequency of items

Item	Quantity	Frequency	twu
A	8	5	109
B	48	5	280
C	48	5	176
D	12	4	253
E	10	8	372

Table 5: Index part of transactional array

Pos	Item	Quantity	Profit/Unit	Frequency	Twu
1	D	12	3	4	253
2	A	8	3	5	109
3	B	48	5	5	280
4	C	48	1	5	176
5	E	10	5	8	372

empty block in row 4 is [4,1] so [3,1] will contain the following parts: 12 (quantity of B), 72 (Transaction Utility), [4,1] (Address of next item in transaction). Then the same process is applied to C and E. Since E is the last item, the location field of E will be empty. We repeat that for all other transactions to obtain the final transactional array given in Table 7.

All needed information from Table 1 and 2 has been transformed into the trans-

Table 6: Sorted transaction in order of frequency.

TID	D	A	B	C	E	tu
$T1$	0	0	12	2	2	*72*
$T2$	2	0	12	0	1	*71*
$T3$	0	2	0	1	1	*12*
$T4$	2	1	0	0	1	*14*
$T5$	0	0	0	4	2	*14*
$T6$	0	1	2	0	0	*13*
$T7$	2	0	20	0	1	*111*
$T8$	6	3	0	25	1	*57*
$T9$	0	1	2	0	0	*13*
$T10$	0	0	0	16	1	*21*

actional array so that we can use this array for mining high utility itemsets (even with different threshold).

Table 7: Transactional array of table 1 and 2

	Transactional Array									
Pos	Index	1	2	3	4	5	6	7	8	9
1	D, 12, 3 4, 253	2, 71 [3,2]	2, 14 [2,2]	2, 111 [3,4]	6, 57 [2,4]					
2	A, 8, 3 5, 109	2, 12 [4,2]	1,14 [5,4]	1,13 [3,3]	3, 57 [4,4]	1, 13 [3,5]				
3	B, 48, 5 5, 280	12, 72 [4,1]	12, 71 [5,2]	2, 13 [∅, ∅]	20,111 [5,6]	2, 13 [∅, ∅]				
4	C, 48, 1 5, 176	2, 72 [5,1]	1, 12 [5,3]	4, 14 [5,5]	25, 57 [5,7]	16, 21 [5,8]				
5	E, 10, 5 8, 372	2, 72 [∅, ∅]	1, 71 [∅, ∅]	1, 12 [∅, ∅]	1,14 [∅, ∅]	2, 14 [∅, ∅]	1, 111 [∅, ∅]	1, 57 [∅, ∅]	1, 21 [∅, ∅]	

The following is our algorithm for building transactional array:

Algorithm 1. (Build transactional array).

Input: Database DB.

Output: Transactional array in external memory.

Method:

```
1. for each $T \in DB$ // First time scanning database
2. begin
3.     - Calculate transaction utility $tu(T)$;
4.     - Calculate frequency, quantity, TWU of each item;
5. end;
6. Sort all items in ascending order of frequency;
```

```
Based on sorted items list, build index part of transactional array;
for each $T \in DB$ // *Second time scanning database*
begin
    Sort items in $T$ in order of index part, we have following list:
            $Tlist = (A_1 : s_1, A_2 : s_2, ..., A_k : s_k)$ ;
              // $s_i$ *is quantity of item in transaction* $T$.
    Determine address $[d_1, c_1]$ to save information of item $A_1$ in
     transactional array;
    for $i$:=1 to $k - 1$ do // *With each item in TList* .
      begin
         - Determine address $[d_{i+1}, c_{i+1}]$ where save information of $A_{i+1}$;
         - Save at $[d_i, c_i]$ :Quantity $s_i$ ,Transaction Utility $tu(T)$,
           Address $[d_{i+1}, c_{i+1}]$;
      end;
    Save at $[d_k, c_k]$: Quantity $s_k$ , Transaction Utility $tu(T)$ , empty
     address $[\emptyset, \emptyset]$;
end;
```

4.2 Mining transactional array

Consider all items of the index part of transactional array (top down). For each item i_p, if $TWU(i_p) \geq minutil$ the algorithm gets all transactions from the transactional array that contains that item. From these transactions, it builds the COUI-tree for that item and mines that tree for high utility itemset. It then discards the tree as soon as it has been mined and moves to the next item. COUI-tree of item x must have x as its root. Each COUI-tree has a header table that contains three data fields: item's name, TWU and pointer (pointing to the first and same item in COUI-tree). Each node of COUI-tree includes 4 data fields: item's name, TWU (Utility of transaction that it's inside), an array of quantity of all items from this node up to the root, pointers pointing to the next same label node or null if not. Each transaction is read and inserted into COUI-tree as follow:

Let $[x \,|\, L]$ be a transaction, where x is the first item and L is the rest of the transactions. The algorithm checks whether item x is one of child nodes of the root. If it is, then update the information for that node correspondingly, otherwise, add a new node as a child of root and labels it x. Consider the present node as the root, repeat the process on the next item in L if it is not empty. When adding a new node, an update of horizontal link of the corresponding item in header table is needed.

The COUI-tree building process is illustrated by an example with the transactional array in Table 7.

To mine the high utility itemsets on the transactional array in Table 7 we need to build COUI-tree for the items: D, B and C. We call COUI-tree corresponding for each item as D-COUI-tree, B-COUI-tree and C-COUI-tree respectively. It is not necessary to build A-COUI-tree since $twu(A) = 109 < minutil$, not for E-COUI-tree because there is only one node (E as root) in this tree. D-COUI-tree contains all items co-occurring with D in the transactions. B-COUI-tree contains all items co-occuring with B in the transactions except for D and A. C-COUI-tree contains all items co-occuring with C in the transactions except for D, B and A.

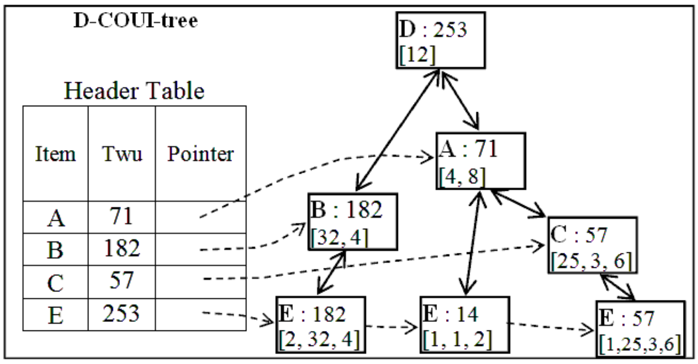
- Building process of D-COUI-tree:

From the index part we know that D's frequency is 4, so there are 4 transactions that contains D inside. Start at the first block in row 1 of transactional array, read information in this block and the address saved in the Location field to reach the next item. Here we get the following sequence.

Starting at [1, 1] we get item D with quantity of 2, this block refers to [3, 2]. At [3, 2] we get item B with quantity of 12, this block refers to [5, 2]. At [5, 2] we get item E with quantity of 1 and an empty Location field so the algorithm stops at this point.

At this first link we get the first transaction of D, $T_1 = (D : 2, B : 12, E : 1)$ and $tu(T_1) = 71$. Likewise, we could get all D's transactions and have: $T_4 = (D : 2, A : 1, E : 1)$ with $tu(T_4) = 14$, $T_7 = (D : 2, B : 20, E : 1)$ with $tu(T_7) = 111$, and $T_8 = (D : 6, A : 3, C : 25, E : 1)$ with $tu(T_8) = 57$. Each transaction is read and inserted into D-COUI-tree. It is noted that twu of the header table needs to be adjusted correctly. Figure 1 shows the D-COUI-tree.

Figure 1: D-COUI-tree



Item	Twu	Pointer
A	71	
B	182	
C	57	
E	253	

- Mining D-COUI-tree:

Mining D-COUI-tree is to find all high utility itemsets that contain D inside. In D-COUI-tree, twu of item A and C is smaller than minutil, so the itemsets that contain them cannot be high utility itemsets and in the candidate generating process we do not generate candidates containing these items.

In turn, consider all items in the header table but this time we do it bottom up, therefore, E will be the first item we encounter. From the pointer in the header table of item E we find 3 nodes in D-COUI-tree labeled E. In the path from the first E to the root we will have (E:2, B:32, D:4) with twu = 182; Push it and all its subsets plus D into $D - list$ (a list contains all high utility candidates containing D) and we will have:

$D - List = \{(E:2, B:32, D:4):182; (E:2, D:4):182; (B:32, D:4):182\}$.

Adjust twu and the array of quantity of each node E, B and D on that path. Twu is subtracted to 182 and the array of quantity is subtracted corresponding (step 1).

The path to root of the second E (E:1, A:1, D:2) with twu = 14 does not generate any candidate that contains A, so only (E:1, D:2) is pushed into $D - List$. In $D - List$, (E:1, D:2) has twu = 182 so this value will be adjusted to 196; Adjust all the values for items E, A and D (step 2).

Likewise, for the third E and we add (E:1, D:6):57 into $D - List$. At this point we have done with item E and move to the next item in the header table. The next items in the header table are C, B and A but all of them have twu = 0 so there is no need to generate any candidates from them. Figure 2 shows this process.

Finish mining D-COUI-tree we have a candidate list in $D - List$. Traverses all candidates and with each $X \in D - List$, we calculate actual utility, if $u(X) \geq minutil$ then X is a high utility itemset. With D-COUI-tree we find $HU = \{EBD(182), \mathrm{BD}(172)\}$. Repeat this process for the next items, in the end, the result will be:

$$HU = \{EBD(182), \mathrm{BD}(172), EB(240), B(240)\}.$$

We can describe the algorithm for building and mining COUI-tree as follows

Algorithm 2 (Build and mine COUI-tree).

Input: Transactional array, utility function, threshold *minutil*.

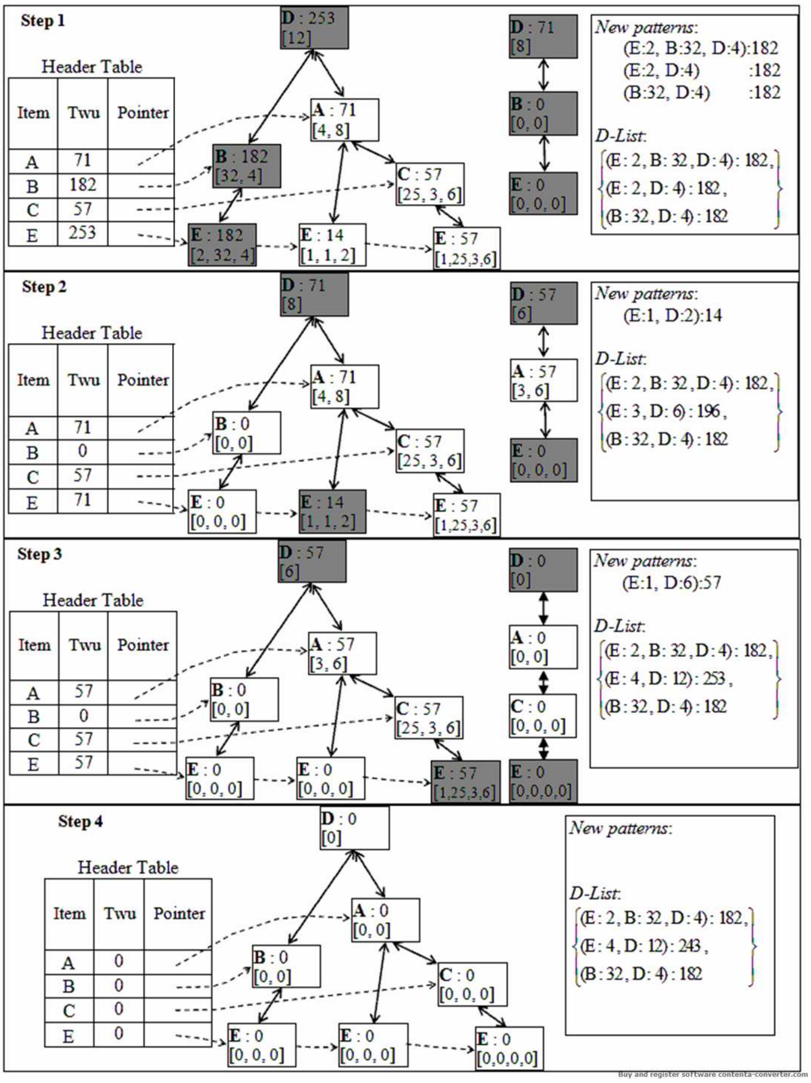
Output: *HU* set contains all high utility itemsets of database *DB*.

Method:

```
1.  From top down, A = first item in index part satisfy $twu(A) \geq minutil$;
2.  repeat
3.      if $twu(A) < minutil$ then goto 15;
4.      Calculate utility of A; // base on quantity and unit utility.
5.      if $u(A) > minutil$ then $HU := HU \cup \{A\}$;
6.      Read frequency s and location of row d that contains A in transaction;
7.      Create root R with label A of (A)-COUI-tree, assign twu = 0, quantity = 0;
8.      for i:=1 to s do //traverse all s blocks in row d of transactional array;
9.          begin
10.             - Start at [d, i], determine $T = (A_1 : s_1, A_2 : s_2, , A_k : s_k)$ and
                  transaction utility $tu(T)$; //$A_1$ is item A.
11.             - Call $insert\_tree(T, R)$ function to insert T into (A)-COUI-tree;
12.         end;
13.     Call MineCOUI-tree (A); // a function mines (A)-COUI-tree.
14.     Free (A)-COUI-tree;
15.     A = next item in index part;
16. Until (A is the last one in index part);
17. Calculate utility of A;
18. if $u(A) > minutil$ then $HU := HU \cup \{A\}$;
```

Here is MineCOUI-Tree function.

Figure 2: 4 steps to mining D-COUI-tree



Step 1

Header Table

Item	Twu	Pointer
A	71	
B	182	
C	57	
E	253	

New patterns:
(E:2, B:32, D:4):182
(E:2, D:4) :182
(B:32, D:4) :182

D-List:
{(E : 2, B : 32, D : 4) : 182,
(E : 2, D : 4) : 182,
(B : 32, D : 4) : 182}

Step 2

Header Table

Item	Twu	Pointer
A	71	
B	0	
C	57	
E	71	

New patterns:
(E:1, D:2):14

D-List:
{(E : 2, B : 32, D : 4) : 182,
(E : 3, D : 6) : 196,
(B : 32, D : 4) : 182}

Step 3

Header Table

Item	Twu	Pointer
A	57	
B	0	
C	57	
E	57	

New patterns:
(E:1, D:6):57

D-List:
{(E : 2, B : 32, D : 4) : 182,
(E : 4, D : 12) : 253,
(B : 32, D : 4) : 182}

Step 4

Header Table

Item	Twu	Pointer
A	0	
B	0	
C	0	
E	0	

New patterns:

D-List:
{(E : 2, B : 32, D : 4) : 182,
(E : 4, D : 12) : 243,
(B : 32, D : 4) : 182}

```
Function: MineCOUI-tree (A);
Method:
$(A) - List := \emptyset$ //Initialize the empty candidate list.
for each (item B in header table of (A)-COUI-tree) //bottom up.
    for each (node N on (A)-COUI-tree that labeled B)
                                        //follow the pointer from header table.
        begin
```

```
            - Read twu and quantity array of all items of node N;
            - Determine pattern $X$ in the path from N up to the root;
            - Generate subset of $X$ that contains A //*discard all low utility TWU items.*
            - Push all subsets generated above into $(A) - List$;
            - Adjust twu and array of quantity of all items on the path of N to the root;
        **end**; //*Finish mining* $(A) - COUI - tree$.
**for each** $Y \in (A) - List$ // *Traverse all candidates in* $(A) - List$.
    **begin**
        - Calculate utility $u(Y)$ of candidate Y;
        - **if** $u(Y) > minutil$ **then** $HU := HU \cup \{Y\}$;
    **end**;
Return $HU$;
```

5 Algorithm Evaluation and Performance Study

5.1 Algorithm Evaluation

a) Algorithm 1: Transaction Array construction

+ Pass I:

- Calculation of transaction utility $tu(T)$, calculation of frequency, quantity, TWU of each item. Hence, the total time complexity of this step is $O(n)$.

- Sorting of all items in ascending order of frequency costs $O(n.logn)$ in time.

- Based on the sorted item list, the building index part of transactional array has time complexity of $O(n)$.

+ Pass II: For each T of DB, we need to identify $Tlist = (A_1 : S_1, A_2 : S_2, ..., A_k : S_k)$, S_i is number of A_i item in transaction T. With each item in $TList$, address $[d_{i+1}, c_{i+1}]$ where save information of A_{i+1} needs to be determined, that makes the total time complexity of $O(n^2)$. In summary, the time complexity for algorithm 1 is $O(n^2)$.

b) Algorithm 2: Mining the transaction array.

+ Building the COUI-tree: At the turn of a top-down data items, time complexity for tree construction of algorithm is $O(n^2)$. Since there are n data items, the total time complexity to build all trees COUI-tree is $O(n^3)$.

+ Minning the COUI-tree: Algorithms considers in turn each data item in the header table, with each B data item to browse nodes in COUI-tree labeled B. Suppose the height of the tree is h, to generates any candidate patterns then generated sub patterns , need a running time complexity of 2^{h-1}.

The greatest height of COUI-trees is equal to the length of the longest transaction in the database transaction: $max(h) = max\{|T|, T \in DB\}, 1 \leq h \leq n$. In the worst case, the database have transactions that include all items, $max\{|T|, T \in DB\} = n$, so $max(h) = n$. In that case, time complexity to mine highest COUI-tree is $O(2^{n-1})$, thefore, the time complexiyty for algorithm 2 is $O(2^n)$.

Algorithm time complexity is the total time complexity of algorithm 1 and algorithm 2 making it as $O(2^n)$ (n is the number of data items).

Although in theory the worst case time complexity of the algorithm is $O(2^n)$, in reality, transactions databases are often extremely sparse, the height h of the tree COUI-tree could be very small compared to n, so the practical running of the algorithm often does not suffer from combinatorial explosion.

5.2 Performance Study

The algorithm was written in Microsoft Visual C ++ 6.0, running on a PC with a Pentium dual core 2.0 GHz CPU, 1 GB of RAM, using Windows XP Professional operating system. The program reads data from files and outputs to a data file. The algorithm was experimented on several real and synthetic data sets. Retail is a market basket dataset from a Belgian supermarket (Brijs, Goethals, Swinnen, Vanhoof, & Wets, 1999). Retail transaction file contains 88,162 transactions, 16,470 items and the average length of transactions is 10.31 [5]

We generated two synthetic datasets using our own program and IBM Quest data generator [10]: (a) T10I500D100K, the average length of transactions is 10.74, with 500 items and the number of transactions is 100K, (b) 10I1000D100K, the average length of transactions is 10.10, with 1000 items and the number of transactions is 100K.

Table 8 shows the characteristics of the datasets. Since all these datasets are normally used for testing traditional frequent itemset mining algorithms, we added quantity and item utility values to the dataset. We generated a utility table based on lognormal distribution with the utility values ranging from 0.1 to 10. The quantities of items were generated randomly in the range of 1 to 10. Test results are shown in Figure 3.

Transactional data is converted into a new database layout set in the external memory, so the algorithm can mine very large datasets. Running time of COUI-Mine algorithm includes data conversion time for the transaction array and the mining of transaction array. Once data has been converted into a new database layout, it can be mined with different utility thresholds without converting the data, hence, running time of the algorithm is reduced to the time to mine the transaction array only. On the dataset Retail, the data conversion time was 4744 seconds. Table 9 shows the running time of the algorithm on dataset Retail with different utility thresholds.

6 Conclusion

Based on the results of the experiments and analyses of the algorithm, some conclusions could be drawn as follow:

+ It needs to be scanned database twice to build transactional array and this array contains enough information for mining high utility itemsets. This transactional array is stored in the external memory, so the algorithm can mine very large

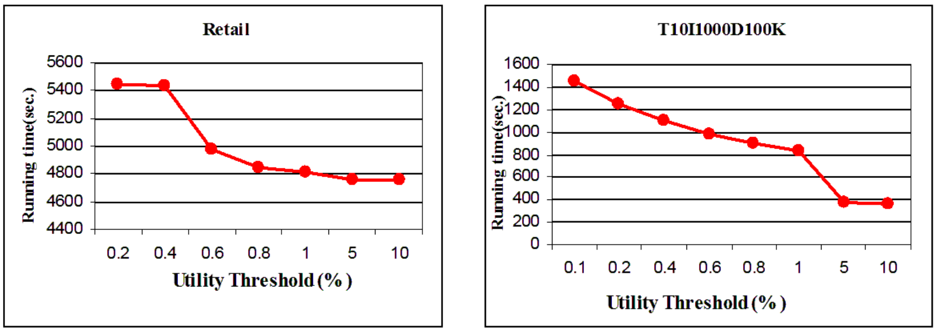
Table 8: Characteristics of Datasets

Dataset	**Number of transactions**	**Number of Items**	**Average Length**
Retail	88.162	16.470	10,31
T10I500D100K	100.000	500	10,74
T10I1000D100K	100.000	1000	10,10

Table 9: Execution time on dataset Retail

Utility Threshold	**COUI-Mine**		
	Phase 1	Phase 2	Total
0,2	4744	702	5446
0,4		697	5411
0,6		237	4981
0,8		99	4843
1		76	4820
5		12	4756
10		11	4755

Figure 3: Execution time with varying minimum utility thresholds on real and synthetic datasets



databases.

+ Mining transactional array is based on small structure of COUI-tree. At each time, only one tree is in the memory, it means that we only store in the memory a small part of the data. Otherwise, mining COUI-tree is using non-recursive algorithm so it reduces time and memory needed in the mining process.

+ After the transactional array is built, the algorithm can mine with arbitrary thresholds.

+ The algorithm avoids massive computations because it does not need to generate candidates and check for constraints like in some other approaches.

+ The algorithm also uses the concept of TWU effectively to reduce the time complexity to generate candidates.

In conclusion, COUI-Mine is an effective algorithm for mining high utility itemsets in large datasets.

References

[1] Nguyen Huy Duc, *"Mining Association Rule in Large Databases"*, In Proceeding of the First National Symposium Fundamental and Applied Information Technology Research (FAIR), Ha Noi, 2003.

[2] Nguyen Thanh Tung, *"Mining High Utility Itemsets in Databases"*. Journal of Computer Science and Cybernetics, Viet Nam, vol. 23, no. 4, pp. 364-373, 2007.

[3] Vu Duc Thi and Nguyen Huy Duc, *"Efficient Algorithm for Mining High Utility Itemsets Based On Prefix-trees"*, Journal of Computer Science and Cybernetics, Viet Nam, vol. 24, no. 3, pp. 204-216, 2008.

[4] R. Agrawal and R. Srikant, *"Fast algorithms for mining association rules"*. In pro-ceedings of 20^{th} International Conference on Very Large Databases, Santiago, Chile, 1994.

[5] Frequent Itemset Mining Implementations Repository, 2003. `http://fimi.cs.helsinki.fi/data/`

[6] H. Yao and H. J. Hamilton, *"Mining itemset utilities from transaction databases"*. Data & Knowledge Engineering, vol. 59, pp. 603- 626, 2006.

[7] Y. Liu, W.-K. Liao, and A. Choudhary, *"A Fast High Utility Itemsets Mining Algorithm"*, Proc. UBDM'05, Chicago Illinois, 2005.

[8] M. El-Hajj and Osmar R. Zaiane. *"COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation"*. In Proc. 2003 Intl Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD), August 2003.

[9] M. El-Hajj and Osmar R. Zaiane. *"Inverted matrix: Efcient discovery of frequent items in large datasets in the context of interactive mining"*. In Proc. 2003 Intl Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD), pp. 109-118, August 2003.

[10] IBM Synthetic Data Generator, `http://www.almaden.ibm.com/software/quest/resources/index.html`

Received 18th May 2009