Mining Closed$^+$ High Utility Itemsets without Candidate Generation

Cheng-Wei Wu[1], Philippe Fournier-Viger[2], Jia-Yuan Gu[3], Vincent S. Tseng[1]*

[1]Department of Computer Science, Chiao Tung University, Hsinchu, Taiwan
[2]Department of Computer Science, University of Moncton, Moncton, Canada
[3]Department of Computer Science and Information Engineering, Cheng Kung University, Tainan, Taiwan

silvemoonfox@gmail.com, philippe.fournier-viger@umoncton.ca, lester13.gu@gmail.com,
Correspondence: vtseng@cs.nctu.edu.tw

***Abstract*—High utility itemsets (*HUIs*) mining refers to discovering sets of items that not only co-occur but also carry high utilities (e.g., high profits). HUI mining receives extensive attentions in recent years due to the wide applications in various domains like commerce and biomedicine. However, huge number of HUIs might be produced to users, which degrades the efficiency of the mining process. A promising solution to this problem is to mine *closed$^+$ high utility itemset* (*CHUI*), a compact and lossless representation of HUIs. Nevertheless, existing algorithms incur the problem of producing a large amount of candidates, which degrades the mining performance in terms of time and space. In this paper, a novel algorithm named *CHUI-Miner* (*Closed$^+$ High Utility Itemset mining without candidates*) for mining CHUIs is proposed, which directly computes the utility of itemsets without producing candidates. To our best knowledge, this is the first work addressing the issue of mining CHUIs without candidate generation. Experimental results show that *CHUI-Miner* is several orders of magnitude faster than the state-of-the-art algorithms.**

Keywords—utility mining, closed itemset mining, closed$^+$ high utility itemset, compact and lossless representation

I. INTRODUCTION

High utility itemset mining has emerged as an important research topic in data mining and has received a lot of attention in recent years. In utility mining, each item in a database has an *external utility* (e.g., weight / unit profit / interestingness) and an item in a record has an *internal utility* (e.g., purchase quantity / occurrence count / importance). The *utility of an itemset* is measured by its external utility and internal utility. An itemset is called *high utility itemset* (abbreviated as *HUI*) if its utility is no less than a user-specified *minimum utility threshold*. Mining high utility itemsets in a database consists of identifying the sets of items that co-occur in a database with high utilities (e.g., high profits). HUI mining has been considered in a wide range of applications such as *cross marketing* [2, 5, 14], *user behavior analysis* [12] and *biomedical applications* [3]. Although HUI mining is essential to many applications, mining HUIs is not as easy as *mining frequent itemsets* due to the absence of *downward closure property* [1, 4]. The search space for mining HUIs cannot be directly reduced as it is done in frequent itemset mining because a superset of a low utility itemset can be a HUI. Many efficient algorithms were proposed for mining HUIs, such as *Two-Phase* [7], *IHUP* [2], *IIDS* [18] *UP-Growth* [13] and *UP-Growth$^+$*[14]. These algorithms use the concept of *transaction-weighted downward closure* property [7] to find HUIs and consist of two phases named *Phase I* and *Phase II*. In Phase I, they generate candidates for HUIs by overestimating the utility of itemsets. In Phase II, HUIs are identified from the candidates by scanning the original database once.

Recently, a novel algorithm named HUI-Miner [9] was proposed, which relies on a novel structure named *utility-list* to discover HUIs by using a single phase. The utility-list structure allows HUI-Miner to directly compute the utility of generated itemsets in main memory without scanning the original database. Besides, by using this structure, a tighter upper bound on the utility of supersets of an itemset can be obtained so that a larger part of the search space can be effectively pruned. To the best of our knowledge, HUI-Miner is the state-of-the-art algorithm for mining HUIs [9].

Although the aforementioned algorithms represent state-of-the-art techniques for HUI mining, they often present a large number of HUIs to users. A large number of HUIs and candidates cause the mining tasks to suffer from long execution time and huge memory consumption. The more itemsets and candidates are generated by the algorithms, the more resources they require. When the system resources are limited (memory, disk space or processing power), it is often impractical to generate the entire set of HUIs. Moreover, a large amount of HUIs is hard to be utilized by users.

To address this problem, Tseng et al. proposed a compact and lossless representation of HUIs [15] called *closed$^+$ high utility itemset* (abbreviated as *CHUI*). An itemset is called *closed high utility itemset* if its utility is no less than a user-specified *minimum utility threshold* and it has no proper supersets having the same support. Although the set of closed HUIs is a compact representation of HUIs, it is not lossless. To tackle this problem, each closed HUI is annotated with a special structure called *utility unit array* such that the resulting itemset is called a *closed$^+$ high utility itemset*. The idea of utility unit array makes the set of CHUIs lossless because HUIs and their utilities can be derived from this set without accessing the original database. Besides, it was shown that the set of CHUIs can be several orders of magnitude smaller than the set of all HUIs, especially for dense databases. Moreover, they proposed the first algorithm named *CHUD* to discover CHUIs in databases and an efficient method named *DAHU* to recover all the HUIs from the complete set of CHUIs. To our best knowledge, CHUD is the current best algorithm for mining CHUIs.

Although CHUI mining is desirable and useful in many applications, it is still a challenging data mining task that has not been deeply explored. To our best knowledge, only the CHUD algorithm has been proposed so far for efficiently mining CHUIs. However, the CHUD algorithm has not been compared with the most recent algorithms. Therefore the answer to the question "*Is it still faster to discover CHUIs than HUIs?*" is unknown. Besides, another critical problem with the CHUD algorithm is that it relies on the two phase model and overestimates too many low utility itemsets as candidates, which degrades its overall performance in terms of execution time and memory usage. In view of these problems, an interesting question is raised: "*Could we design a one-phase algorithm for efficiently mining CHUIs without producing candidates?*".

In fact, there exist several challenges to develop such an algorithm. First, designing a CHUI mining algorithm requires to correctly integrate techniques from closed itemset mining [6, 10, 16] with those for HUI mining such that the complete set of CHUIs can be captured with no miss. Second, developing an algorithm for discovering CHUIs in one phase would require a special data structure that allows computing the utility of CHUIs in memory without producing candidates and performing a costly database scan to calculate their utilities and utility unit arrays. Lastly, the resulting algorithm is required to be more efficient in terms of execution time and memory than state-of-the-art algorithms for mining CHUI and HUIs. Lastly, the resulting algorithm may be slower or less memory efficient than current state-of-the-art algorithms for mining CHUI and HUIs.

In this paper, we address all the above challenges by proposing a new framework for mining CHUIs without candidate generation, a topic that has not been explored so far. The contributions of this work are summarized as follows.

First, we propose a novel structure named *EU-List* (*Extended Utility-List*) to maintain the utility information of itemsets in transactions, which allows to efficiently calculate the utility and utility unit arrays of itemsets in memory without scanning the original database.

Second, a novel algorithm named *CHUI-Miner* (*Closed$^+$ High Utility Itemset mining without candidates*) incorporating the *EU-List* structure is proposed, which adopts the divide-and-conquer methodology to mine the complete set of CHUIs in databases without producing candidates.

Third, we perform an extensive experimental study on both real and synthetic datasets to evaluate the performance of the proposed algorithm. Experimental results show that CHUI-Miner outperforms the current best algorithm for mining CHUIs substantially, especially for dense datasets. Moreover, the combination of CHUI-Miner and DAHU is also faster than state-of-the-art algorithms for mining HUIs even on sparse datasets. More importantly, CHUI-Miner is more than two orders of magnitude faster than the compared algorithms on dense datasets.

The rest of this paper is organized as follows. Section 2 presents preliminaries and defines the problem of closed$^+$ high utility itemset mining. Section 3 describes the CHUI-Miner algorithm in detail. An extensive performance evaluation is presented in Section 4. Section 5 draws the conclusion.

TABLE 1. AN EXAMPLE DATABASE

TID	Transaction	TU
T_1	A(3), B(3), C(2), E(3)	26
T_2	B(2), D(7)	20
T_3	A(2), C(5), D(5), E(2)	25
T_4	A(1), C(6), D(5), E(1), F(2)	29

TABLE 2. UNIT PROFITS OF ITEMS

Item	A	B	C	D	E	F
Unit Profit	4	3	1	2	1	4

TABLE 3. TWU VALUES OF ITEMS

Item	A	C	E	D	B	F
TWU	80	80	80	74	46	29

TABLE 4. REORGNIZED TRANSACTIONS WHEN MIN_UTIL = 40

TID	Reorganized Transaction	RTU
T_1	A[12], B[9], C[2], E[3]	26
T_2	B[6], D[14]	20
T_3	A[8], C[5], D[10], E[2]	25
T_4	A[4], C[6], D[10], E[1]	21

II. PROBLEM DEFINITION

Let be a finite set of distinct *items* $I^* = \{I_1, I_2, \ldots, I_N\}$, such that each item $I_i \in I^*$ is associated with a positive number $p(I_i, D)$, called its *external utility* (e.g. unit profit). A *transactional database* $D = \{T_1, T_2, \ldots, T_M\}$ is a set of transactions, where each transaction $T_r \in D$ $(1 \leq r \leq M)$ is a subset of I^* and has an unique identifier r, called its *Tid*. In a transaction T_r, each item $I_i \in I^*$ is associated with a positive number $q(I_i, T_r)$, called its *internal utility* in T_r (e.g. purchase quantity). An itemset $X = \{I_1, I_2, \ldots, I_k\}$ is a set of k distinct items, where $I_i \in I^*$ $(1 \leq i \leq k)$ and k is the length of X. A k-itemset is an itemset of length k. An itemset X is said to be *contained* in a transaction T_r if $X \subseteq T_r$.

For example, TABLE 1 shows an example database containing five transactions. Each row in TABLE 1 represents a transaction, in which each letter represents an item and has a purchase quantity (internal utility). The unit profit (external utility) of each item is shown in TABLE 2.

Definition 1 (Superset and subset). Let be non-empty itemsets X and Y. X is *a subset of* Y, or equivalently Y is a *superset of* X if $X \subseteq Y$. If $X \subset Y$, X is a *proper subset of* Y and Y is a *proper superset of* X.

Definition 2 (Support count and Tidset). The *support count* of an itemset X is the number of transactions containing X in D and is denoted as $SC(X)$. The *Tidset* of an itemset X is denoted as $TidSet(X)$ and defined as the set of Tids of transactions containing X. The support count of an itemset X is expressed in terms of $TidSet(X)$ as $SC(X) = |TidSet(X)|$.

For example, $TidSet(\{ACE\}) = \{1, 3, 4\}$ and $SC(\{ACE\}) = |TidSet(\{ACE\})| = 3$.

Property 1. For a k-itemset $X = \{I_1, I_2, \ldots, I_k\}$, $SC(X) = |TidSet(I_1) \cap TidSet(I_2) \cap \ldots \cap TidSet(I_k)|$.

Property 2. Let itemset Y be a proper superset of X. Then, $TidSet(Y) \subseteq TidSet(X)$.

Definition 3 (Utility of an item in a transaction). The *utility of an item* $I_i \in I^*$ *in a transaction* T_r is denoted as $u(I_i, T_r)$ and defined as $p(I_i, D) \times q(I_i, T_r)$.

Definition 4 (Utility of an itemset in a transaction). The *utility of a k-itemset* $X = \{I_1, I_2, \ldots, I_k\}$ *in a transaction* T_r is denoted $u(X, T_r)$ and defined $\sum_{I_i \in X} u(I_i, T_r)$.

For example, the utility of {A} in T_1 is $u(\{A\}, T_1) = p(\{A\}, D) \times q(\{A\}, T_1) = 4 \times 3 = 12$. The utility of {ACE} in T_1 is $u(\{ACE\}, T_1) = u(\{A\}, T_1) + u(\{C\}, T_1) + u(\{E\}, T_1) = 12 + 2 + 3 = 17$.

Definition 5 (Transaction utility). The *transaction utility* (abbreviated as *TU*) of a transaction T_r is denoted as $TU(T_r)$ and defined as $u(T_r, T_r)$.

Definition 6 (Total utility). The *total utility of a database D* is denoted as $TotalU_D$ and defined as $\sum_{T_r \in D} TU(T_r, T_r)$.

For example, the transaction utility of T_1 is $TU(T_1) = u(\{ABCE\}, T_1) = 26$. The total utility of the database in TABLE 1 is $TotalU_D = TU(T_1) + TU(T_2) + TU(T_3) + TU(T_4) = 26 + 20 + 25 + 29 = 100$.

Definition 7 (Utility and relative utility of an itemset). The *utility of an itemset X* in the database *D* is defined as $u(X) = \sum_{r \in TidSet(X)} u(X, T_r)$. The *relative utility of X* is defined as $ru(X) = u(X)/TotalU_D$.

For example, the utility of {AC} is $u(\{AC\}) = u(\{AC\}, T_1) + u(\{AC\}, T_3) + u(\{AC\}, T_4) = 17 + 15 + 11 = 43$. The relative utility of {AC} is $ru(\{AC\}) = 43/100 = 43\%$.

Definition 8 (High utility itemset). An itemset *X* is called *high utility itemset* (abbreviated as *HUI*) if $u(X)$ is no less than a user-specified *minimum utility threshold min_util* ($0 < min\_util \le TotalU_D$). Otherwise, the itemset is *low utility*. Let $min\_util/TotalU_D$ be the *relative minimum utility threshold rmin_util.* An equivalent definition is that *X* is a HUI iff $ru(X) \ge rmin\_util$. The complete set of HUIs in *D* is denoted as *H*.

For example, if $min\_util = 30$, the set of HUIs in TABLE 1 is H = {{D}:34, {AC}:37, {AD}:32, {AE}:30, {CD}:31, {ACD}:43, {ACE}:43, {CDE}:34, {ADE}: 35, {ACDE}:46}, where the number beside each itemset is its utility.

Definition 9 (Closed itemset). An itemset *X* is called *closed itemset* if there exists no proper superset $Y \supset X$ in *D* such that $SC(X) = SC(Y)$. The complete set of closed itemset is denoted as *C*.

For example, the complete set of closed itemset in TABLE 1 is C = {{B}:2, {D}:3, {BD}:1, {ACE}:3, {ABCE}:1, {ACED}:2, {ACDEF}:1}, in which the number beside each itemset is its support count.

Definition 10 (Closure of an itemset). Let *Y* be a superset of an itemset *X*. *Y* is called the *closure* of *X* if *Y* is closed and $SC(Y) = SC(X)$. The closure of *X* is defined as $closure(X) = \cap_{r \in TidSet(X)} T_r$.

For example, the closure of {AC} is $closure(\{AC\}) = T_1 \cap T_3 \cap T_4$ = {ABCE} ∩ {ACDE} ∩ {ACDEF} = {ACE}.

Property 3. For any itemset *X*, the support count of *X* is equal to that of its closure. Equivalently, $SC(X) = max\{SC(Y) \mid Y$ is closed and $X \subseteq Y\}$.

For example, the support count of {AC} is equal to that of its closure $SC(closure(\{AC\})) = 3$. Besides, $SC(\{AC\}) = max\{SC(\{ACE\}), SC(\{ABCE\}), SC(\{ACED\}), SC(\{ACDEF\})\} = 3$.

Definition 11 (Closed high utility itemset). An itemset *X* is called *closed high utility itemset* iff *X* is closed and $u(X) \ge min\_util$. The complete set of closed HUIs is defined as $CH = H \cap C$.

For example, if $min\_util = 30$, the complete set of closed HUIs in TABLE 1 is $CH = H \cap C$ = {{D}, {ACE}, {ACDE}}.

Definition 12 (Utility unit array). For a *k*-itemset $X = \{I_1, I_2, \ldots, I_k\}$, its *utility unit array* is denoted as $V(X) = [v_1, v_2, \ldots, v_k]$, where the *k*-th utility value in $V(X)$ is defined as $V(X, I_i) = \sum_{r \in TidSet(X) \wedge I_i \in T_r} u(I_i, T_r)$. Besides, the utility value of *X* can be expressed as $u(X) = \sum_{i=1}^{k} V(X, I_i)$.

For example, the second utility value of the utility unit array of {ACE} is $V(\{ACE\},\{C\}) = u(\{C\}, T_1) + u(\{C\}, T_3) + v(\{C\}, T_4) = 2 + 5 + 6 = 13$. The utility unit array of {ACE} is $V(\{ACE\}) = [24, 13, 6]$ and $u(\{ACE\}) = V(\{ACE\}, \{A\}) + V(\{ACE\}, \{C\}) + V(\{ACE\}, \{E\}) = 43$.

Definition 13 (Closed$^+$ high utility itemset). A closed high utility itemset *X* is called *closed$^+$ high utility itemset* (abbreviated as *CHUI*) iff *X* is attached with its utility unit array. The complete set of CHUIs is denoted as CH^+.

For example, CH^+ = {({D}, [34]), ({ACE}, [24, 13, 6]), {ACDE}, [12, 11, 20, 3]}.

Property 4. For any high utility itemset *X*, there exists a CHUI itemset *Y* such that $Y = closure(X)$ and $u(Y) \ge u(X)$.

Property 5. For any itemset *X* that is not a CHUI, all its subsets are low utility.

Property 6. The complete set of CHUIs is a lossless representation of all HUIs. For any *k*-itemset $X = \{I_1, I_2, \ldots, I_k\}$, if $closure(X) \in CH^+$, $u(X)$ can be calculated as $\sum_{I_i \in X} V(closure(X), I_i)$ by using the utility unit array of its closure without accessing the original database.

Problem Statement. Given a database *D* with internal and external utility of items, and a user-specified minimum utility threshold *min_util*, the problem statement is to discover from *D* all closed itemsets whose utilities are no less than *min_util* and their utility unit arrays.

```
Algorithm: CHUI-Miner
   Input:  D, min_util;
  Output:  The complete set of closed$^+$ high utility itemsets;
      01.  Scan D once to find promising items by calculating TWU of items;
      02.  Scan D again to construct EU-List of promising items;
      03.  GEN-CHUI($\emptyset$, $\emptyset$, P, min_util);
```

Fig. 1. The pseudo code of the CHUI-Miner algorithm

III. The Proposed Method: CHUI-Miner

In this section, we propose an efficient algorithm for discovering CHUIs without producing candidates, namely *CHUI-Miner* (*Closed$^+$ High Utility Itemset mining without candidates*). The CHUI-Miner algorithm adopts the proposed structure *EU-List* (*Extended Utility-List*) to maintain the utility information of itemsets in transactions. The pseudo code of CHUI-Miner is shown in Fig. 1. The algorithm has two input parameters: (1) a database D and (2) a user specified minimum utility threshold *min_util.* It outputs the complete set of CHUIs in D. The proposed algorithm consists of two parts: (1) construction of *EU-Lists* of promising items and (2) generation of CHUIs by using *EU-Lists*.

3.1 Construction of EU-Lists of promising items

To efficiently mine CHUIs in a database without producing candidates and avoid repeatedly scanning the original database, the information of itemsets in transactions are maintained in *EU-Lists*. The construction of the initial *EU-Lists* can be performed with two database scans (line 1 of Fig. 1). In the first database scan, the transaction utility of each transaction and *TWU* of items are calculated.

Definition 14 (TWU of an itemset). The *TWU* (*Transaction-Weighted Utilization*) of an itemset X is the sum of the transaction utilities of all the transactions containing X, which is defined as $TWU(X) = \sum_{r \in TidSet(X)} TU(T_r)$. For example, TABLE 3 shows the TWU of items.

Definition 15 (TWDC property). The *TWDC* (*Transaction-Weighted Utilization Downward Closure*) property [2, 7] states that for any itemset X if its $TWU(X)$ is less than *min_util*, all supersets of X are low utility.

After scanning the database once, items and their TWU values are obtained. By the *TWDC* property, if the *TWU* of an item is less than *min_util*, all its supersets are not CHUIs. These items are called *unpromising items*. Definition 16 gives a formal definition for *unpromising items* and *promising items* [13, 14, 15].

Definition 16 (Promising and unpromising items). An item $I_i \in I^*$ ($1 \leq i \leq N$) is a *promising item* if $TWU(I_i) \geq min\_util$. Otherwise, the item is called an *unpromising item*.

During the second database scan, when a transaction T_j ($1 \leq j \leq |D|$) is retrieved, unpromising items are removed from T_j and their utilities are eliminated from the transaction utility of T_j since only supersets of promising items can be CHUIs. The promising items in T_j are sorted in TWU descending order. Note that other orders can be used. In this paper, we use the TWU descending order for the running example. A transaction resulting from the above process is called a *reorganized transaction* and its transaction utility is called *RTU* (*Reorganized Transaction Utility*). The *RTU* of a reorganized transaction T_r is denoted as $RTU(T_r)$, where $1 \leq r \leq |D|$. For example, TABLE 4 shows reorganized transactions for the database in TABLE 1, where the number beside each item is the multiplication of its internal and external utilities.

{A}		
Tid	EU	RU
1	12	14
3	8	17
4	4	17

{C}		
Tid	EU	RU
1	2	12
3	5	12
4	6	11

{E}		
Tid	EU	RU
1	3	9
3	2	10
4	1	10

{D}		
Tid	EU	RU
2	14	6
3	10	0
4	10	0

{B}		
Tid	EU	RU
1	9	0
2	6	0

Fig. 2. Utility-lists of promising items

3.2 Generation of CHUIs by using EU-Lists

After scanning the database twice, the *EU-List* of each promising item is constructed. In the proposed algorithm, each item(set) is associated with an *EU-List*. The *EU-List* of an item(set) X includes a *utility-list* [9], support count of X, utility unit array of X, and two ordered sets named $PrevSet(X)$ and $PostSet(X)$. The *utility-list* of X consists of several tuples. Each tuple in the *utility-list* of X represents the utility information of X in the reorganized transaction T_r and has three fields: *Tid*, *EU* and *RU*. Fields *Tid* and *EU* respectively indicate the identifier of T_r and the exact utility of X in T_r. Field *RU* indicates the *remaining utility of X in T_r*. The concept of remaining utility is based on the following definitions.

Definition 17 (Precede and succeed). Given a set of items $\{I_1, I_2, \ldots, I_N\}$ and a total order relation R: $I_1 \prec I_2 \prec \ldots \prec I_N$ among items. An item I_i *precedes* an item I_j if $I_i \prec I_j$ ($1 \leq i < j \leq N$). Otherwise, I_i *succeeds* I_j. In the rest of the paper, items in itemsets are sorted according to R.

Definition 18 (Remaining utility of an itemset in a transaction). The *remaining utility of a k-itemset $X = \{I_1, I_2, \ldots, I_k\}$, in a reorganized transaction T_r* is denoted as $RU(X, T_r)$ and defined as $\sum_{I_k \in T_r \wedge I_k \prec I_i} u(I_i, T_r)$.

Definition 19 (Remaining utility of an itemset in a database). The *remaining utility of a k-itemset $X = \{I_1, I_2, \ldots, I_k\}$ in a database D* is denoted as $RU(X)$ and defined as $\sum_{r \in TidSet(I_i)} RU(I_i, T_r)$.

Definition 20 (Utility-list structure). The *utility-list of an itemset X* is denoted as $UL(X)$, which is a list containing $|TidSet(X)|$ tuples of the form $<Tid, EU, RU>$. Each tuple is called an *element* and maintains the utility information of X in the transaction T_r, where $r \in TidSet(X)$. An element *associated with T_r in utility-list of X* is denoted as $UL(X, T_r)$, which contains the information $<r, EU(X, T_r), RU(X, T_r)>$. The *EU* and *RU* of the element are respectively denoted as $UL(Y, Tr).EU$ and $UL(Y, Tr).RU$. For example, Fig. 2 shows the *utility-lists* of promising items for the database of Table 1.

Definition 21 (Extension of an itemset). Let $X = \{x_1, x_2, \ldots, x_u\}$ ($x_i \in I^*$, $1 \leq i \leq u$) and $Y = \{y_1, y_2, \ldots, y_v\}$ ($y_j \in I^*$, $1 \leq j \leq v$) be two itemsets. Y is an extension of X if $X \subset Y$ and each item $y_j \notin X$ succeeds all items in X according to the total order R.

For example, if R is A≺C≺E≺D≺B≺F, {ACE} is not an extension of {AE} but it is an extension of {AC}.

Definition 22. In the *utility-list* of an itemset X, the sums of all its *EU* and *RU* values are denoted as $SumEU(X)$ and $SumRU(X)$, respectively.

```
Procedure: GEN-CHUI
   Input:  An itemset X, PostSet(X), PreSet(X), min_util;
  Output:  The complete set of closed+ high utility itemsets;
  01.  While PostSet(X) ≠ Ø;
  02.      I ← min≺(PostSet(X));
  03.      Remove I from PostSet(X);
  04.      Y ← X ∪ I;
  05.      Construct EU-List of Y;
  06.      PrevSet(Y) ← PrevSet(X);
  07.      If(SumEU(Y) + SumRU(Y)) ≥ min_util;
  08.          If(IsSubsumedCheck(Y, PrevSet(Y)) = false)
  09.              EUL(Y_C) ← EUL(Y);
  10.              PrevSet(Y_C) ← PrevSet(Y);
  11.              PostSet(Y_C) ← Ø;
  12.                 Y_C ← ClosureComputation(Y, PostSet(Y_C), PostSet(X));
  13.              V(Y_C) ← UpdateUtilityUnitArray(Y_C, V(Y_C));
  14.              If(SumEU(Y_C) ≥ min_util
  15.                  Output Y_C, SC(Y_C) and V(Y_C);
  16.              GEN-CHUI(Y_C, PostSet(Y_C), PrevSet(Y_C), min_util);
  17.              PrevSet(X) ← PrevSet(X) ∪ I;
```

Fig. 3. The pseudo code of the GEN-CHUI procedure

```
Procedure: IsSubsumedCheck
   Input:  An itemset Y, PrevSet(Y);
  Result:  Return true if Y is subsumed by already mined closed+ high utility item-
           sets. Otherwise, return false.
  For each item J∈PrevSet(Y) do
      If(TidSet(Y) ⊆ TidSet(J))
          Return true;
      Return false;
```

Fig. 4. The pseudo code of the IsSubsumedCheck procedure

Property 7. If *SumEU*(*X*) + *SumRU*(*X*) is less than *min_util*, all extensions of *X* are low utility.

The procedure *GEN-CHUI*(Ø, Ø, *P*, *min_util*) is then called to generate CHUIs by using *EU-Lists* of promising items. The procedure is an extension of an efficient algorithm named *DCI-Closed* [6] for mining closed itemsets. For each closed itemset *X* discovered by the algorithm, the algorithm constructs its *EU-List* to calculate its utility to determine whether it is a CHUI and uses property of remaining utility to prune the search space. The procedure has four input parameters: (1) an itemset *X*, (2) *PrevSet*(*X*), (3) *PostSet*(*X*) and (4) the minimum utility threshold *min_util*. It explores the search space of CHUIs that are supersets of *X* by appending items from *PostSet*(*X*) to *X*.

The pseudo code of GEN-CHUI is shown in Fig. 3, which is performed as follows. While *PostSet*(*X*) is not null, the procedure selects an item *I* having the smallest order in *PostSet*(*X*) to create an itemset $Y = X \cup I$ and remove *I* from *PostSet*(*X*) (line 3 to 5 of Fig. 3). Then, the *EU-List* of *Y* is constructed as follows. First, *TidSet*(*Y*) is obtained by intersecting *TidSet*(*X*) and *TidSet*(*I*) and the support count of *Y* is set to |*TidSet*(*Y*)|. Then, the *utility-list* of *Y* is obtained by the following process. For each transaction $T_r \in$ *TidSet*(*X*)∩*TidSet*(*I*), the element *UL*(*Y*, T_r) is created in *UL*(*Y*), where *UL*(*Y*, T_r).*EU* is set to the sum of *UL*(*X*, T_r).*EU* and *UL*(*I*, T_r).*EU*, and where *UL*(*Y*, T_r).*RU* is set to *UL*(*I*, T_r).*RU*. Finally, *PrevSet*(*Y*) is initialized to *PrevSet*(*X*). In other word, *UL*(*Y*, T_r).*EU* = *UL*(*X*, T_r).*EU* + *UL*(*I*, T_r).*EU* and *UL*(*Y*, T_r).*RU* = *UL*(*X*, T_r).*RU* – *UL*(*I*, T_r).*EU*. Finally, *PrevSet*(*Y*) is initialized to *PrevSet*(*X*).

```
Procedure: ClosureComputation
   Input:  An itemset Y, PostSet(Y_C), PostSet(X);
  Result:  Return Y's closure Y_C. Update PostSet(Y_C) and EU-List of Y_C;
  Y_C ← Y;
  PostSet(Y_C) ← Ø;
  For each item Z∈PostSet(X) do
      If(TidSet(Y_C) ⊆ TidSet(X))
          Y_C ← Y_C ∪ Z;
          For each Tid r∈TidSet(Y_C) do
              EL(Y_C, T_r).EU ← EL(Y_C, T_r).EU + EL(Z, T_r).EU;
              EL(Y_C, T_r).PU ← EL(Y_C, T_r).PU – EL(Z, T_r).PU;
      else PostSet(Y_C) ← PostSet(Y_C) ∪ Z;
  Return Y_C;
```

Fig. 5. The pseudo code of the ClosureComputation procedure

If the sum of *sumEU*(*Y*) and *sumRU*(*Y*) is less than *min_util*, the combination of *Y* with any item *I*∈*PostSet*(*Y*) is low utility. Otherwise, the algorithm calls procedure *IsSubsumedCheck*(*Y*, *PreSet*(*Y*)) to check whether *Y is subsumed* by an already mined CHUI.

Definition 15 (Subsume). The itemset *Y is subsumed* by the itemset *S* if $Y \subset S$ and *SC*(*Y*) = *SC*(*S*). For example, {C} is subsumed by {ACE} because {C} ⊂ {ACE} and *SC*({C}) = *SC*({ACE}).

Property 8. Given two itemsets *Y* and *S*, if $Y \subset S$ and *SC*(*Y*) = *SC*(*S*) then *closure*(*X*) = *closure*(*Y*).

Property 9. Given an itemset *X* and an item $I_i \in I^*$ $(1 \le i \le N)$, *TidSet*(*X*) ⊆ *TidSet* (I_i) ⇔ $I_i \in$ *closure*(*X*).

By Definition 15, if we can find an already mined CHUI *S* that subsumes *Y*, we can conclude that *Y* is not closed and *closure*(*S*) = *closure*(*Y*). Hence, we can safely prune the itemset *Y* and stop exploring the search space of succeeding supersets of *Y*. Otherwise, the procedure returns true.

The pseudo code of the *IsSubsumedCheck* procedure is shown in Fig. 4. The procedure has two input parameters: (1) an itemset *Y* and (2) *PrevSet*(*Y*) and it is performed as follows. For each item *J* in *PrevSet*(*Y*), if *TidSet*(*Y*) is contained in *TidSet*(*J*), the procedure returns *true* to indicate that *Y* is subsumed by an already mined CHUI and thus that *Y* is not a CHUI (Property 8 and 9). If *TidSet*(*Y*) is not contained in the *TidSet* of any item in *PostSet*(*Y*), the procedure returns *false* to indicate that the closure of *Y* is closed.

If *Y* passes the subsumption check, the procedure *ClosureComputation*(*Y*, *PostSet*(Y_C), *PostSet*(*X*)) is called to compute the closure of *Y* and construct its *EU-List* by updating the *EU-List* of *Y*. The pseudo code of the *ClosureComputation* procedure is shown in Fig. 5, and it is performed as follows. Initially, a variable Y_C for storing the closure of *Y* is set to Ø and *EL*(Y_C) is set to *EL*(*Y*). Then, for each item *Z* in *PostSet*(*X*), we check if *TidSet*(*Y*) is contained in *Tidset*(*Z*). If it is not contained in *Tidset*(*Z*), *Z* is added to *PostSet*(*Y*). Otherwise, *Z* is added to Y_C because *Z* is contained in the closure of *Y* by Property 9.

The *EU-List* of Y_C is updated by the following process. For each Tid $r \in$ *TidSet*(Y_C) ∩ *TidSet*(*Z*), the *EL*(Y_C, T_r) is updated to <*r*, v_1, v_2>, where v_1 is *EL*(Y_C, T_r).*EU* + *EL*(*Z*, T_r).*EU* and v_2 is *EL*(Y_C, T_r).*PU* – *EL*(*Z*, T_r).*PU*. After processing all the items in *PostSet*(*X*), the *EU-List* of Y_C is updated and Y_C captures the closure of *Y*. Then, the procedure returns *Y*'s closure Y_C.

After calling the *ClosureComputation* procedure, the *UpdateUtilityArray*(Y_C) procedure is then called to calculate the utility unit array of Y_C. If $Y_C = \{I_1, I_2, ..., I_k\}$, the utility unit array $V(Y_C)$ of Y_C is calculated as follows. For each item $I_i \in Y_C$, we calculate $V(Y_C, I_i)$ (Definition 12) by scanning $UL(I_i)$ once to sum up the utilities of I_i in all elements whose Tids are in $Tidset(Y_C)$. Note that in our implementation for any item $I \in Y_C$ such that $I \notin Y$, the utility value $V(I, Y_C)$ is calculated during the closure computation by the *ClosureComputation* procedure. For the sake of simplicity, this optimization is not shown in the pseudo code of the *ClosureComputation* procedure due to the page limitation.

Then the algorithm outputs Y_C if $SumEU(Y_C)$ is no less than *min_util* because Y_C meets the criteria of being a closed itemset, a high utility itemset, and is associated with its utility unit array. In other word, Y_C is a CHUI. The algorithm then calls the GEN-CHUI procedure to further explore the search space and find the CHUIs that are succeeding supersets of Y_C. Then, the item I is added into *PrevSet*(X). When the *CHUI-Miner* algorithm terminates, all the CHUIs in the database are obtained.

IV. Experimental Evaluations

In this section, we compare the performance of CHUI-Miner with two algorithms, CHUD [15] and HUI-Miner [9]. CHUD mines closed$^+$ high utility itemsets and HUI-Miner mines high utility itemsets. These algorithms are to our knowledge the best algorithms for their respective mining tasks. Moreover, to compare CHUI-Miner with HUI-Miner which produces different results, we have added a version of CHUI-Miner combining CHUI-Miner and DAHU, called CHUI-Miner+DAHU, for mining high utility itemsets. Experiments are performed on a computer with an Intel Core i7-2600 CPU @ 3.40 GHz, 8 GB of RAM, and Windows 7 SP1, 64-bit version. All algorithms are written in Java, and run on JVM 1.7.0_45-b18. Both the initial and maximum heap memory sizes of the JVM are set to 2,048 MB.

To analyze the performance of the algorithms in different situations, we test the algorithms with six different datasets. Their characteristics are shown in Table 5. The datasets *Mushroom*, *Connect*, *Chess* and *Retail* are obtained from the FIMI Repository [20], *Foodmart* is obtained from the Microsoft foodmart 2000 database [21] and *Chain Store* is obtained from NUMineBench 2.0 [11]. Except for *Foodmart* and *Chain Store*, these datasets do not provide the external and internal utilities of items. As in the performance evaluation of previous studies [9, 13, 14], the external utilities of items are generated between 0.01 and 10 using a log-normal distribution and the internal utilities of items are generated randomly ranging from 1 to 5.

Table 5. Statistical information about datasets

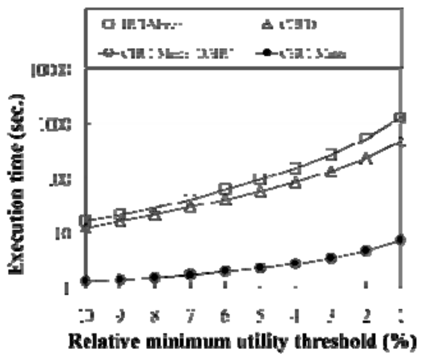
Dataset	#Item	#Transaction	AvgLen	MaxLen	Type
Mushroom	119	8,124	23	23	Dense
Connect	129	67,557	43	43	Dense
Chess	75	3,196	37	37	Dense
Foodmart	1,559	4,141	4.4	14	Sparse
Retail	16,470	88,162	10.3	76	Sparse
Chain Store	46,086	1,112,949	7.3	170	Sparse

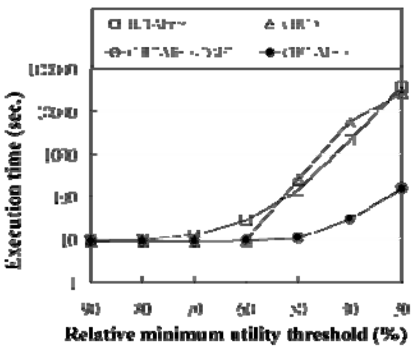
4.1 Experiments on dense datasets

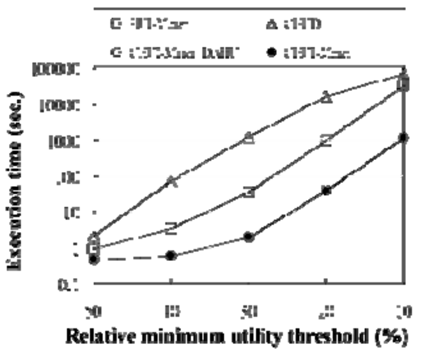
The first experiment consists of running the algorithms on dense datasets *Mushroom*, *Connect* and *Chess* while varying the *min_util* parameter. Execution times of the compared algorithms are shown in Fig 6. (a), (b), (c). The results show that CHUI-Miner and CHUI-Miner/DAHU outperform both CHUD and HUI-Miner for their respective tasks. For example, when *min_util* = 1% for the *Mushroom* dataset, CHUI-Miner and CHUI-Miner/DAHU are more than 100 times faster than CHUD and HUI-Miner. The reason why the former algorithms performs so well is that CHUD and HUI-Miner respectively produce a large number of candidates and HUIs, as shown in Table 6 and Fig 7. (a). Another insightful observation is that mining CHUIs is not always faster than mining HUIs (it depends on the algorithm design). For example, CHUD is slower than HUI-Miner on *Connect* for low *min_util* values. The reason is that CHUD produces too many candidates. For example, for *Chess* and *min_util* = 10%, CHUD produces more than 50 GB of candidates and cause our computer to run out of disk space.

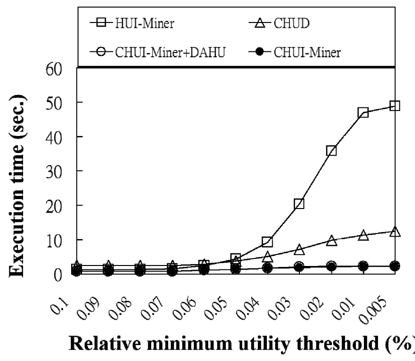
4.2 Experiments on sparse datasets

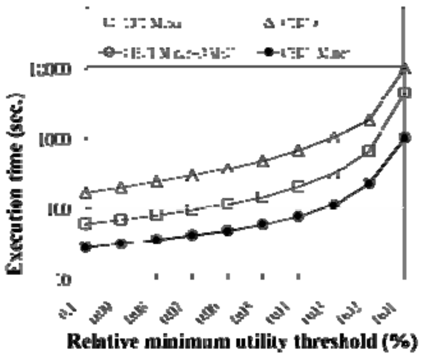
The second experiment consists of running the algorithms on sparse datasets. Execution times of the algorithms are shown in Fig 6. (d), (e), (f) and the number of HUIs, candidates and CHUIs are given in Table 6 and Fig 7. (b), (c). Results show that CHUI-Miner and CHUI-Miner/DAHU outperform both CHUD and HUI-Miner again, despite that the performance gap is smaller than for dense datasets. For example, when *min_util* = 0.005% for the *Foodmart* dataset, CHUI-Miner and CHUI-Miner/DAHU are about 6 times faster than CHUD and 21 times faster than HUI-Miner. Because the number of closed$^+$ high utility itemset becomes very small compared to the number of high utility itemsets for the *Foodmart* dataset when *min_util* is lower than 0.06%, CHUD becomes faster than HUI-Miner. But still, it can be observed that CHUD is slower than HUI-Miner on the *Retail* and *ChainStore* datasets for low *min_util* values, and CHUI-Miner and CHUI-Miner/DAHU remain faster than both of them. The reasons why these latter are still faster than HUI-Miner is that (1) CHUI-Miner performs a two way search to build utility lists unlike HUI-Miner which performs a three-way search and (2) the closure of itemsets are constructed directly by CHUI-Miner, thus avoiding calculating utility lists of several low utility or non-closed itemsets. This result is interesting since it is contrary to the observation in the field of frequent pattern mining that mining closed itemsets is slower than mining all frequent itemsets for sparse datasets. Moreover, as show in Fig 6, CHUI-Miner has good scalability even on large datasets.


(a) Mushroom


(b) Connect


(c) Chess


(d) Foodmart


(e) Retail

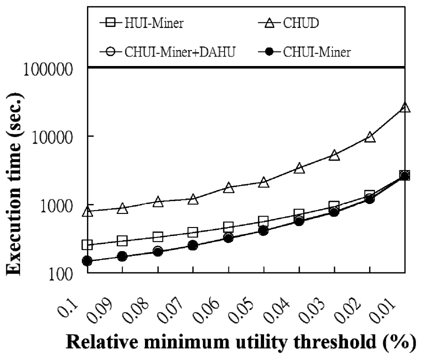
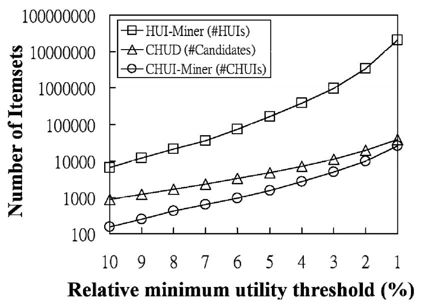
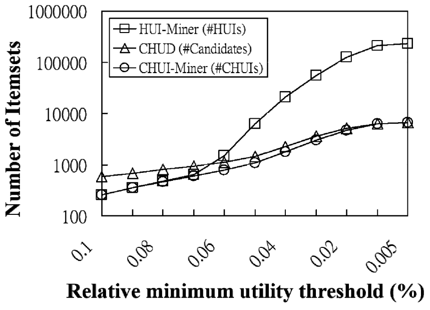

(f) ChainStore

Fig. 6. Execution time on different datasets


(a) Mushroom


(b) Foodmart

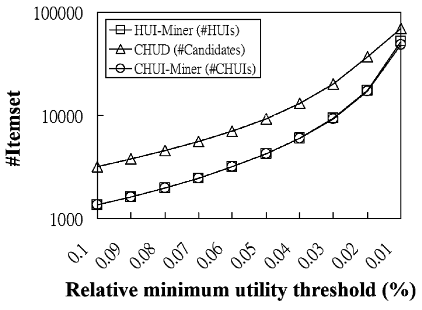
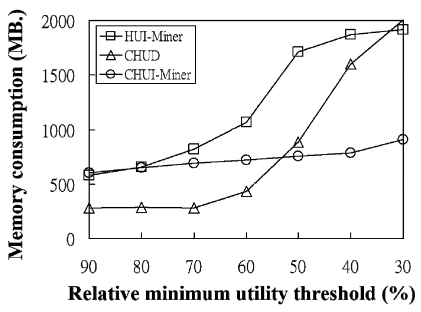
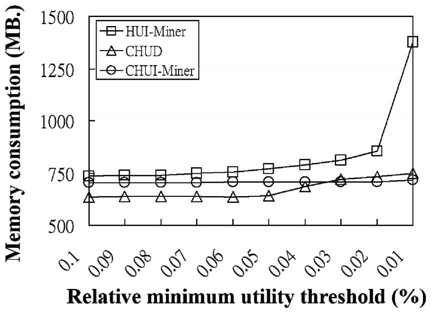

(c) Retail

Fig. 7. Number of itemsets on different datasets


(a) Connect


(b) Retail

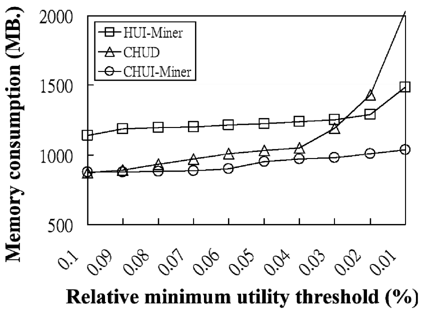

(c) ChainStore

Fig. 8. Memory consumption on different datasets

TABLE 6. NUMBERS OF HIGH UTILITY ITEMSETS, CANDIDATES PRODUCED BY CHUD AND CLOSED HIGH UTILITY ITEMSETS ON DIFFERENT DATASETS

Dataset	*min_util*	HUI-Miner	CHUD Algo.	CHUI-Miner	Reduction ratio
		#HUIs	#Candidates	#CHUIs	#HUIs/#CHUIs
Mushroom	3%	970,638	11,294	4,911	197.646
	2%	3,381,719	19,362	10,046	336.623
	1%	20,392,064	39,522	25,611	796.223
Connect	50%	0	711	0	N/A
	40%	0	21,667	0	N/A
	30%	91232	106146	3332	27.381
Chess	30%	0	111,813	0	N/A
	20%	252,069	1,844,542	49,410	5.102
	10%	58,482,852	*	5,204,820	11.236
Foodmart	0.06%	1483	1113	770	1.926
	0.04%	20751	2208	1762	11.777
	0.02%	124,589	5,107	4,704	26.486
Retail	0.06%	3187	7026	3171	1.005
	0.04%	6014	13055	5967	1.008
	0.02%	17,403	36,827	17,076	1.019
ChainStore	0.03%	593	6268	593	1.000
	0.02%	1,165	12206	1,165	1.000
	0.01%	3,884	40486	3,870	1.004

4.3 Memory usage comparisons

Fig. 8 shows the memory consumption of the algorithms on three datasets carrying different characteristics, including a dense dataset (i.e., Connect), a sparse dataset (i.e., Retail) and a large scale dataset (i.e., ChainStore). In Fig. 8, we can see that CHUI-Miner uses less memory than CHUD to find CHUIs, especially for dense dataset. The reason is that the number of CHUIs is considerably smaller than the total number of HUIs for low *min_util* values. Besides, we have found that CHUD consumes less memory than CHUI-Miner in several cases because the local TU table, the main data structure of CHUD, stores less information than utility lists/EU-lists, the main structures used by HUI-Miner and CHUI-Miner. But for the ChainStore dataset, the advantage provided by mining CHUIs is small, so the memory usage of CHUI-Miner and CHUD is similar to HUI-Miner. We can also observe that the memory usage of CHUI-Miner is more stable than the memory usage of HUI-Miner and CHUD.

V. CONCLUSION

In this paper, we have proposed a new framework for mining CHUIs without candidate generation. A novel structure named *EU-List* (*Extended Utility-List*) is proposed to maintain the utility information of itemsets in transactions, which allows to efficiently calculate the utility and utility unit arrays of itemsets in one phase. Moreover, a novel algorithm named *CHUI-Miner* (*Closed$^+$ High Utility Itemset mining without candidates*) incorporating the *EU-List* structure is proposed, which discovers the complete set of CHUIs in databases without producing candidates. Experimental results on real-life datasets show that CHUI-Miner can be more than two orders of magnitude faster than the state-of-the-art CHUI mining algorithm [15]. The implementation of CHUI-Miner has been incorporated into the open-source high utility pattern mining toolbox *UP-Miner* [16]. The source codes of CHUI-Miner and the compared algorithms [9, 15] are public available at: http://bigdatalab.cs.nctu.edu.tw/software.htm.

References

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," Proc. of the 20th Int'l Conf. on Very Large Data Bases, pp. 487-499, 1994.

[2] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong, and Y.-K. Lee, "Efficient tree structures for high utility pattern mining in incremental databases," IEEE Transactions on Knowledge and Data Engineering, vol. 21, issue 12, pp. 1708-1721, 2009.

[3] R. Chan, Q. Yang, and Y. Shen, "Mining high utility itemsets," Proc. of IEEE Int'l Conf. on Data Mining (ICDM), pp. 19-26, Nov., 2003.

[4] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 1-12, 2000.

[5] H.-F. Li, H.-Y. Huang, Y.-C. Chen , Y.-J. Liu, and S.-Y. Lee, "Fast and memory efficient mining of high utility itemsets in data streams," Proc. of the IEEE Int'l Conf. on Data Mining (ICDM), pp. 881-886, 2008.

[6] C. Lucchese, S. Orlando, and R. Perego, "Fast and memory efficient mining of frequent closed itemsets," IEEE Transactions on Knowledge and Data Engineering, vol. 18, issue 1, pp. 21-36, 2006.

[7] Y. Liu, W. Liao, and A. Choudhary, "A fast high utility itemsets mining algorithm," Proc. of the Utility-Based Data Mining Workshop, pp. 90-99, 2005.

[8] Y.-C. Li, J.-S. Yeh, and C.-C. Chang, "Isolated items discarding strategy for discovering high utility itemsets," Data & Knowledge Engineering, vol. 64, issue 1, pp. 198-217, Jan., 2008.

[9] M. Liu and J. Qu, "Mining High Utility Itemsets without Candidate Generation," Proc. of ACM Int'l Conf. on Information and Knowledge Management (CIKM), pp. 55-64, 2012.

[10] N. Pasquier, T. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," Proc. of the Int'l Conf. on Database Theory (ICDT), pp. 398–416, 1999.

[11] J. Pisharath, Y. Liu, B. Ozisikyilmaz, R. Narayanan, W. K. Liao, A. Choudhary, and G. Memik, "NU-MineBench version 2.0 dataset and technical report,"
http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html

[12] B.-E. Shie, P. S. Yu, and V. S. Tseng, "Mining interesting user behavior patterns in mobile commerce environments," Applied Intelligence, vol. 38, issue 3, pp. 418-435, 2013.

[13] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "UP-Growth: an efficient algorithm for high utility itemset mining," Proc. of ACM SIGKDD, pp. 253–262, 2010.

[14] V. S. Tseng, B.-E. Shie, C.-W. Wu, and P. S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," IEEE Transactions on Knowledge and Data Engineering, vol. 25, issue 8, pp. 1772-1786, 2013.

[15] V. S. Tseng, C.-W. Wu, P. Fournier-Viger and P. S. Yu, "Efficient Algorithms for Mining the Concise and Lossless Representation of High Utility Itemsets," IEEE Transactions on Knowledge and Data Engineering, vol. 27, issue 3, pp. 726-739, 2015.

[16] V. S. Tseng, C.-W. Wu, J.-H. Lin and P. Fournier-Viger, , "UP-Miner: A Utility Pattern Mining Toolbox, accepted and to appear in Proc. of IEEE Int'l Conf. on Data Mining (ICDM), 2015. http://idb.csie.ncku.edu.tw/tsengsm/software.htm

[17] C.-W. Wu, B.-E. Shie, V. S. Tseng, and P. S. Yu, "Mining top-k high utility itemsets," Proc. of ACM SIGKDD, pp. 78-86, 2012.

[18] H. Yao, H. J. Hamilton, and L. Geng, "A unified framework for utility-based measures for mining itemsets," Proc. of ACM SIGKDD 2nd Workshop on Utility-Based Data Mining, pp. 28-37, 2006.

[19] M. J. Zaki and C. J. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," IEEE Transactions on Knowledge and Data Engineering, vol. 17, issue 4, pp. 462–478, 2005.

[20] Frequent Itemset Mining Implementations Repository, http://fimi.cs.helsinki.fi/

[21] FoodMart2000, Microsoft Developer Network (MSDN) http://msdn.microsoft.com/enus/library/aa217032(v=sql.80).asp