Mining High Utility Itemsets without Candidate Generation

Mengchi Liu[†‡] Junfeng Qu[†*]
[†]State Key Lab of Software Engineering, School of Computer, Wuhan University, Wuhan 430072, China
[‡]School of Computer Science, Carleton University, Ottawa K1S 5B6, Canada
mengchi@scs.carleton.ca cocoqjf@gmail.com

ABSTRACT

High utility itemsets refer to the sets of items with high utility like profit in a database, and efficient mining of high utility itemsets plays a crucial role in many real-life applications and is an important research issue in data mining area. To identify high utility itemsets, most existing algorithms first generate candidate itemsets by overestimating their utilities, and subsequently compute the exact utilities of these candidates. These algorithms incur the problem that a very large number of candidates are generated, but most of the candidates are found out to be not high utility after their exact utilities are computed. In this paper, we propose an algorithm, called HUI-Miner (High Utility Itemset Miner), for high utility itemset mining. HUI-Miner uses a novel structure, called utility-list, to store both the utility information about an itemset and the heuristic information for pruning the search space of HUI-Miner. By avoiding the costly generation and utility computation of numerous candidate itemsets, HUI-Miner can efficiently mine high utility itemsets from the utility-lists constructed from a mined database. We compared HUI-Miner with the state-of-the-art algorithms on various databases, and experimental results show that HUI-Miner outperforms these algorithms in terms of both running time and memory consumption.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining

General Terms

Algorithms, Experimentation, Performance

Keywords

High utility itemset, mining algorithm

*Corresponding author.

1. INTRODUCTION

The rapid development of database techniques facilitates the storage and usage of massive data from business corporations, governments, and scientific organizations. How to obtain valuable information from various databases has received considerable attention, which results in the sharp rise of related research topics. Among the topics, the high utility itemset mining problem is one of the most important, and it derives from the famous frequent itemset mining problem [7, 8].

Mining frequent itemsets is to identify the sets of items that appear frequently in transactions in a database. The frequency of an itemset is measured with the support of the itemset, i.e., the number of transactions containing the itemset. If the support of an itemset exceeds a user-specified minimum support threshold, the itemset is considered as frequent. Most frequent itemset mining algorithms employ the downward closure property of itemsets [4]. That is, all supersets of an infrequent itemset are infrequent, and all subsets of a frequent itemset are frequent. The property provides the algorithms with a powerful pruning strategy. In the process of mining frequent itemsets, once an infrequent itemset is identified, the algorithms no longer check all supersets of the itemset. For example, for a database with n items, after the algorithms identify an infrequent itemset containing k items, there is no need to check all of its supersets, i.e., $2^{(n-k)} - 1$ itemsets.

Mining of frequent itemsets only takes the presence and absence of items into account. Other information about items is not considered, such as the independent utility of an item and the context utility of an item in a transaction. Typically, in a supermarket database, each item has a distinct price/profit, and each item in a transaction is associated with a distinct count which means the quantity of the item one bought. Consider the database in Fig. 1. There are seven items in the utility table and seven transactions in the transaction table in the database. To calculate support, an algorithm only makes use of the information of the first two columns in the transaction table, the information of both the utility table and the other columns in the transaction table are discarded. However, an itemset with high support may have low utility, or vice versa. For example, the support and utility of itemset {bc} appearing in T1, T2, and T6 are 3 and 18 respectively(See Section 2.1 for utility computation), and those of itemset {de}

Item	a	b	c	d	e	f	g
Utility	1	2	1	5	4	3	1

(a) Utility table

Tid	Transaction	Count
T1	{ b, c, d, g }	{ 1, 2, 1, 1 }
T2	{ a, b, c, d, e }	{ 4, 1, 3, 1, 1 }
T3	{ a, c, d }	{ 4, 2, 1 }
T4	{ c, e, f }	{ 2, 1, 1 }
T5	{ a, b, d, e }	{ 5, 2, 1, 2 }
T6	{ a, b, c, f }	{ 3, 4, 1, 2 }
T7	{ d, g }	{ 1, 5 }

(b) Transaction table

Figure 1: Database

appearing in T2 and T5 are 2 and 22. In some applications, such as market analysis, one may be more interested in the utility rather than support of itemsets. Traditional frequent itemset mining algorithms cannot evaluate the utility information about itemsets.

Like frequent itemsets, itemsets with utilities not less than a user-specified minimum utility threshold are generally valuable and interesting, and they are called "high utility itemsets". To mine all high utility itemsets from a database is very intractable, because the downward closure property of itemsets no longer holds for high utility itemsets. When items are appended to an itemset one by one, the support of the itemset monotonously decreases or remains unchanged, but the utility of the itemset varies irregularly. For example, for the database in Fig. 1, the supports of {a}, {ab}, {abc}, and {abcd} are 4, 3, 2, and 1, but the utilities of these itemsets are 16, 26, 21, and 14, respectively. Suppose the threshold is 20, and then high utility {abc} contains both high utility {ab} and low utility {a}. Therefore, the pruning strategy used in the frequent itemset mining algorithms becomes invalid.

Recently, a number of high utility itemset mining algorithms have been proposed [25, 18, 14, 5, 23, 22]. Most of the algorithms adopt a similar framework: firstly, generate candidate high utility itemsets from a database; secondly, compute the exact utilities of the candidates by scanning the database to identify high utility itemsets. However, the algorithms often generate a very large number of candidate itemsets and thus are confronted with two problems: (1) excessive memory requirement for storing candidate itemsets; (2) a large amount of running time for generating candidates and computing their exact utilities. When the number of candidates is so large that they cannot be stored in memory, the algorithms will fail or their performance will be degraded due to thrashing.

To solve these problems, we propose in this paper an algorithm for high utility itemset mining. The contributions of the paper are as follows:

1. A novel structure, called *utility-list*, is proposed. A utility-list stores not only the utility information about an itemset but also the heuristic information about whether the itemset should be pruned or not.

2. An efficient algorithm, called *HUI-Miner* (*High Utility Itemset Miner*), is developed. Different from previous algorithms, HUI-Miner does not generate candidate high utility itemsets. After constructing the initial utility-lists from a mined database, HUI-Miner can mine high utility itemsets from these utility-lists.

3. Extensive experiments on various databases were performed to compare HUI-Miner with the state-of-the-art algorithms. Experimental results that show HUI-Miner outperforms these algorithms are reported.

After the related background is stated in Section 2, the paper is organized according to the three points aforementioned in Section 3, 4, and 5. Our work is summarized in Section 6.

2. BACKGROUND

In the section, we first give the formal description of the high utility itemset mining problem and subsequently introduce the previous solutions to the problem.

2.1 Problem Definition

Let $\mathcal{I}$=$\{i_1, i_2, i_3, \ldots, i_n\}$ be a set of items and DB be a database composed of a utility table and a transaction table. Each item in $\mathcal{I}$ 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 $\mathcal{I}$, in which each item is associated with a count value. An itemset is a subset of $\mathcal{I}$ and is called a k-itemset if it contains k items.

Definition 1. *The external utility of item i, denoted as eu(i), is the utility value of i in the utility table of DB.*

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.*

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, in Fig. 1, eu(e) = 4, iu(e, T5) = 2, and u(e, T5)= iu(e, T5) × eu(e) = 2 × 4 = 8.

Definition 4. *The utility of itemset X in transaction T, denoted as u(X, T), is the sum of the utilities of all the items in X in T in which X is contained, where* $u(X, T) = \sum_{i \in X \wedge X \subseteq T} u(i, T)$.

Definition 5. *The utility of itemset X, denoted as u(X), is the sum of the utilities of X in all the transactions containing X in DB, where* $u(X) = \sum_{T \in DB \wedge X \subseteq T} u(X, T)$.

For example, in Fig. 1, u({ae}, T2) = u(a, T2) + u(e, T2) = 4 × 1 + 1 × 4 = 8, and u({ae}) = u({ae}, T2) + u({ae}, T5) = 8 + 13 = 21.

Definition 6. *The utility of transaction T, denoted as tu(T), is the sum of the utilities of all the items in T, where* $tu(T) = \sum_{i \in T} u(i, T)$*, and the total utility of DB is the sum of the utilities of all the transactions in DB.*

Fig. 2 shows the utility of each transaction, for example, tu(T1) = u(b, T1) + u(c, T1) + u(d, T1) + u(g, T1) = 2 + 2 + 5 + 1 = 10. The total utility of the database in Fig. 1 is 98. An itemset X is *high utility* if u(X) is not less than a user-specified minimum utility threshold denoted as *minutil*,

Tid	T1	T2	T3	T4	T5	T6	T7
TU	10	18	11	9	22	18	10

Figure 2: Transaction Utility

or the product of a minutil and the total utility of a mined database if the minutil is a percentage. Given a database and a minutil, the high utility itemset mining problem is to discover from the database all the itemsets whose utilities are not less than the minutil.

2.2 Related Work

Before the high utility itemset mining problem was formally proposed [25] as above, a variation of the problem had been studied, namely the problem of extracting share frequent itemsets [6, 13, 12] that invariably defines the external utility of each item as 1. The ZP [6], ZSP [6], FSH [13], ShFSH [12], and DCG [11] algorithms for share frequent itemset mining can also be used to mine high utility itemsets. Since the downward closure property cannot be directly applied, Liu et al. proposed an important property [17] for pruning the search space of the high utility itemset mining problem.

Definition 7. *The transaction-weighted utility of itemset X in DB, denoted as twu(X), is the sum of the utilities of all the transactions containing X in DB, where* $twu(X) = \sum_{T \in DB \wedge X \subseteq T} tu(T)$.

Property 1. *If twu(X) is less than a given "minutil", all supersets of X are not high utility.*
Rationale. If $X \subseteq X'$, *then* $u(X') \leq twu(X') \leq twu(X) <$ *minutil.*

Itemset	{a}	{b}	{c}	{d}	{e}	{f}	{g}
TWU	69	68	66	71	49	27	10

Figure 3: Transaction-Weighted Utility

Fig. 3 shows the transaction-weighted utilities of all 1-itemsets. For example, itemset {f} is contained in T4 and T6, and thus twu({f}) = tu(T4) + tu(T6) = 9 + 18 = 27. If a minutil is equal to 30, all supersets of {f} are not high utility according to Property 1. The Two-Phase algorithm [18, 17] first adopts Property 1 to prune the search space. Afterwards, the isolated items discarding strategy (IIDS) is proposed [14], and the strategy can be incorporated in the above algorithms to improve their performance, for example, the FUM [14] and DCG+ [14] algorithms outperform ShFSH and DCG, respectively.

ZP, ZSP, FSH, ShFSH, DCG, Two-Phase, FUM, and DCG+ mine high utility itemsets as the famous Apriori algorithm [4] mines frequent itemsets. Given a database, firstly, all 1-itemsets are candidate high utility itemsets. After scanning the database, the algorithms eliminate unpromising 1-itemsets and generate 2-itemsets from the remaining 1-itemsets as candidate high itemsets. After the second scan over the database, unpromising 2-itemsets are eliminated and 3-itemsets as candidates are generated from the remaining 2-itemsets.. The procedure is performed repeatedly until there is no generated candidate itemset. Finally, these algorithms, except for DCG and DCG+, compute the exact utilities of all remaining candidates by an additional database scan to identify high utility itemsets (DCG and DCG+ compute exact utility in each database scan.). Besides the two problems mentioned in Section 1, these algorithms suffer from the level-wise mining problems as well, e.g., repeated database scans.

The algorithms based on the FP-Growth algorithm [9] show better performance. These algorithms include IHUPTWU [5], UP-Growth [23], and UP-Growth+ [22]. Firstly, they transform a mined database into a prefix-tree, and the tree maintains the utility information about itemsets. Secondly, for each item of the tree, if it is estimated to be valuable, namely there is likely to be high utility itemsets containing the item, the algorithms construct a conditional prefix-tree for the item. Thirdly, the algorithms recursively process all conditional prefix-trees to generate candidate high utility itemsets. Finally, the algorithms scan the database again to compute the exact utilities of all candidates for identifying high utility itemsets. Reducing the numbers of both database scans and candidate itemsets, these algorithms outperform the Apriori-based algorithms. Even so, compared with the number of resultant high utility itemsets, these algorithms still generate a large number of candidate itemsets in most cases, and it is very costly to both generate these candidates and compute their exact utilities.

There are also a number of studies that focus on the problem of mining an approximate set of all high utility itemsets [10, 24] or a condensed set of all high utility itemsets [20, 21]. In this study, the problem of mining the complete set of all high utility itemsets from a database is discussed.

3. UTILITY-LIST STRUCTURE

To mine high utility itemsets, many previous algorithms directly perform on an original database. Although FP-Growth-based algorithms generate candidate itemsets from prefix-trees, they have to compute the exact utilities of candidates by scanning the database. In the section, we propose a utility-list structure to maintain the utility information about a database.

3.1 Initial Utility-Lists

In our HUI-Miner algorithm, each itemset holds a utility-list. Initial utility-lists storing the utility information about a mined database can be constructed by two scans of the database. Firstly, the transaction-weighted utilities of all items are accumulated by a database scan. If the transaction-weighted utility of an item is less than a given minutil, the item is no longer considered according to Property 1 in the subsequent mining process. For the items whose transaction-weighted utilities exceed the minutil, they are sorted in transaction-weighted-utility-ascending order. For the database in Fig. 1, suppose the minutil is 30, and then the algorithm no longer takes items f and g into consideration after the first database scan. The remaining items are sorted: e<c<b<a<d.

Definition 8. *A transaction is considered as "revised" after (1) all the items whose transaction-weighted utilities are less than a given minutil are deleted from the transaction; (2) the remaining items are sorted in transaction-weighted-utility-ascending order.*

When scanning the database again, the algorithm revises

Tid	Item	Util.	Item	Util.	item	Util.	item	Util.	item	Util.	TU
T1	c	2	b	2	d	5					9
T2	e	4	c	3	b	2	a	4	d	5	18
T3	c	2	a	4	d	5					11
T4	e	4	c	2							6
T5	e	8	b	4	a	5	d	5			22
T6	c	1	b	8	a	3					12
T7	d	5									5

Figure 4: Database View

each transaction for constructing initial utility-lists. The database view in Fig. 4 lists all revised transactions derived from the database in Fig. 1. From here on, the following convention holds in the remainder of this paper:

Convention 1. *A transaction is considered as revised, and all the items in an itemset are sorted in transaction-weighted-utility-ascending order, when mentioned.*

Definition 9. *Given an itemset X and a transaction (or itemset) T with X⊆T, the set of all the items after X in T is denoted as T/X.*

For example, consider the view in Fig. 4, T2/{eb} = {ad} and T2/{c} = {bad}.

Definition 10. *The remaining utility of itemset X in transaction T, denoted as ru(X, T), is the sum of the utilities of all the items in T/X in T, where ru(X, T) =* $\sum_{i \in (T/X)} u(i, T)$.

Each element in the utility-list of itemset X contains three fields: *tid*, *iutil*, and *rutil*.

- Field tid indicates a transaction T containing X.
- Field iutil is the utility of X in T, i.e., u(X, T).
- Field rutil is the remaining utility of X in T, i.e., ru(X, T).

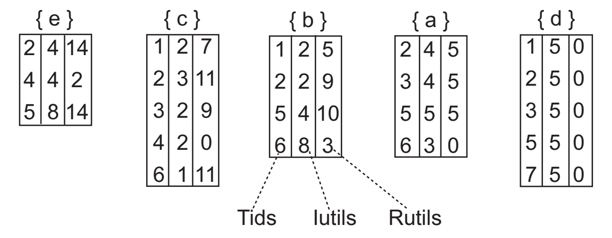

Figure 5: Initial Utility-Lists

During the second database scan, the algorithm constructs the initial utility-lists showed in Fig. 5. For example, consider the utility-list of itemset {c}. In T1, u({c}, T1) = 2, ru({c}, T1) = u(b, T1) + u(d, T1) = 2 + 5 = 7, and thus element <1, 2, 7> is in the utility-list of {c} (<x, y, z> means <tid, iutil, rutil>, and 1 represents T1 for simplicity.). In T2, u({c}, T2) = 3, ru({c}, T2) = u(b, T2) + u(a, T2) + u(d, T2) = 2 + 4 + 5 = 11, and thus element <2, 3, 11> belongs to the utility-list of {c} as well. The rest can be figured out in the same manner.

3.2 Utility-Lists of 2-Itemsets

No need for database scan, the utility-list of 2-itemset $\{xy\}$ can be constructed by the intersection of the utility-list of $\{x\}$ and that of $\{y\}$. The algorithm identifies common transactions by comparing the tids in the two utility-lists. Suppose the lengths of the utility-lists are m and n respectively, and then $(m + n)$ comparisons at most are enough for identifying common transactions, because all tids in a utility-list are ordered. The identification process is actually a 2-way comparison. For example, the tid comparison between the utility-lists of itemsets {e} and {c} in Fig. 5 is demonstrated in Fig. 6(a).

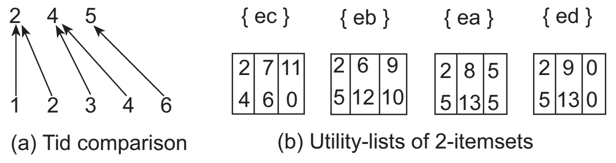

Figure 6: Constructing Utility-Lists of 2-Itemsets

For each common transaction t, the algorithm will generate an element E and append it to the utility-list of $\{xy\}$. The tid field of E is the tid of t. The iutil of E is the sum of the iutils associated with t in the utility-lists of $\{x\}$ and $\{y\}$. Suppose x is before y, and then the rutil of E is assigned as the rutil associated with t in the utility-list of $\{y\}$.

Fig. 6(b) depicts the utility-lists of all the 2-itemsets with itemset {e} as prefix. For example, to construct the utility-list of itemset {eb}, the algorithm intersects the utility-list of {e}, i.e., {<2, 4, 14>, <4, 4, 2>, <5, 8, 14>}, and that of {b}, i.e., {<1, 2, 5>, <2, 2, 9>, <5, 4, 10>, <6, 8, 3>}, which results in {<2, 6, 9>, <5, 12, 10>}. One can observe from the database view in Fig. 4 that itemset {eb} only appears in T2 and T5. In T2, u({eb}, T2) = u(e, T2) + u(b, T2) = 2 + 4 = 6, and ru({eb}, T2) = u(a, T2) + u(d, T2) = 4 + 5 = 9. Similarly, in T5, the utility of {eb} is 8 + 4 = 12, and the remaining utility of {eb} is 5 + 5 = 10.

3.3 Utility-Lists of k-Itemsets (k≥3)

To construct the utility-list of k-itemset $\{i_1 \cdots i_{(k-1)} i_k\}$ (k≥3), we can directly intersect the utility-list of $\{i_1 \cdots i_{(k-2)} i_{(k-1)}\}$ and that of $\{i_1 \cdots i_{(k-2)} i_k\}$ as we do to construct the utility-list of a 2-itemset. For example, to construct the utility-list of {eba}, we can intersect the utility-list of {eb} and that of {ea} in Fig. 6(b), and the resultant utility-list is depicted in Fig. 7(a). Itemset {eba} does appear in T2 and T5 in the database view in Fig. 4, and however the utilities of the itemset in T2 and T5 are 10 and 17 rather than 14 and 25, respectively.

The reason for miscalculating the utility of {eba} in T2 is that the sum of the utilities of both {eb} and

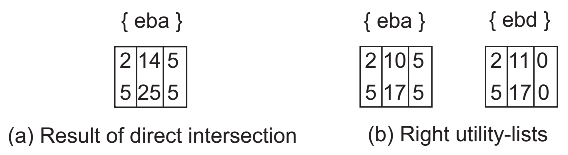

Figure 7: Utility-Lists of 3-Itemsets

Algorithm 1: *Construct* Algorithm

```
   Input: $P.UL$, the utility-list of itemset $P$;
          $Px.UL$, the utility-list of itemset $Px$;
          $Py.UL$, the utility-list of itemset $Py$.
   Output: $Pxy.UL$, the utility-list of itemset $Pxy$.
1  $Pxy.UL = NULL$;
2  foreach element $Ex \in Px.UL$ do
3      if $\exists Ey \in Py.UL$ and $Ex.tid == Ey.tid$ then
4          if $P.UL$ is not empty then
5              search such element $E \in P.UL$ that
               $E.tid == Ex.tid$;
6              $Exy = <Ex.tid, Ex.iutil + Ey.iutil - E.iutil,
               Ey.rutil>$;
7          else
8              $Exy = <Ex.tid, Ex.iutil + Ey.iutil, Ey.rutil>$;
9          end
10         append $Exy$ to $Pxy.UL$;
11     end
12 end
13 return $Pxy.UL$;
```

{ea} in T2 contains the utility of {e} in T2 twofold. Generally, to calculate the utility of $\{i_1 \cdots i_{(k-2)} i_{(k-1)} i_k\}$ in T, the following formula holds: u($\{i_1 \cdots i_{(k-2)} i_{(k-1)} i_k\}$, T) = u($\{i_1 \cdots i_{(k-2)} i_{(k-1)}\}$, T) + u($\{i_1 \cdots i_{(k-2)} i_k\}$, T) - u($\{i_1 \cdots i_{(k-2)}\}$, T).

Therefore, the iutil of the element associated with T2 in the utility-list of {eba} is: u({eba}, T2) = u({eb}, T2) + u({ea}, T2) - u({e}, T2) = 6 + 8 - 4 = 10. That associated with T5 is: u({eba}, T5) = u({eb}, T5) + u({ea}, T5) - u({e}, T5) = 12 + 13 - 8 = 17. The values of u({eb}, T), u({ea}, T), and u({e}, T) can be accessed from the utility-lists of {eb}, {ea}, and {e}, respectively.

Suppose itemsets *Px* and *Py* are the combinations of itemset *P* with items *x* and *y* (*x* is before *y*.), respectively, and *P.UL*, *Px.UL*, and *Py.UL* are the utility-lists of itemsets *P*, *Px*, and *Py*. Algorithm 1 shows how to construct the utility-list of itemset *Pxy*. The utility-list of a 2-itemset is constructed when *P.UL* is empty, namely when *P* is empty, and the utility-list of a k-itemset (k≥3) is constructed when *P.UL* is not empty. Note that element *E* in line 5 can always be found out when *P.UL* is not empty, because the tid sets in both *Px.UL* and *Py.UL* are subsets of the tid set in *P.UL*. The utility-lists of all the itemsets with {eb} as prefix constructed by Algorithm 1 are showed in Fig. 7(b).

Thus far, we have illustrated how to construct the utility-list of an itemset. When does HUI-Miner construct the utility-list of an itemset and how does HUI-Miner judge whether or not the utility-list of an itemset should be constructed, which will be illuminated in the next section.

4. HIGH UTILITY ITEMSET MINER

After constructing the initial utility-lists from a database, the HUI-Miner algorithm can efficiently mine all high utility itemsets from the utility-lists as the Eclat algorithm mines frequent itemsets [26]. In the section, the search space of HUI-Miner is first introduced, and subsequently we propose a pruning strategy for the algorithm. Finally, the HUI-Miner algorithm and a number of implementation details are presented.

4.1 Search Space

The search space of the high utility itemset mining problem can be represented as a set-enumeration tree [19]. Given a set of items $\mathcal{I} = \{i_1, i_2, i_3, \ldots, i_n\}$ and a total order on all items (suppose $i_1 < i_2 < \cdots < i_n$), a set-enumeration tree representing all itemsets can be constructed as follows. Firstly, the root of the tree is created; secondly, the n child nodes of the root representing n 1-itemsets are created, respectively; thirdly, for a node representing itemset $\{i_s \cdots i_e\}$ $(1 \le s \le e < n)$, the $(n-e)$ child nodes of the node representing itemsets $\{i_s \cdots i_e i_{(e+1)}\}$, $\{i_s \cdots i_e i_{(e+2)}\}$, ..., $\{i_s \cdots i_e i_n\}$ are created. The third step is done repeatedly until all leaf nodes are created. For example, given $\mathcal{I}$ = {e, c, b, a, d} and $e < c < b < a < d$, a set-enumeration tree representing all itemsets of $\mathcal{I}$ is depicted in Fig. 8.

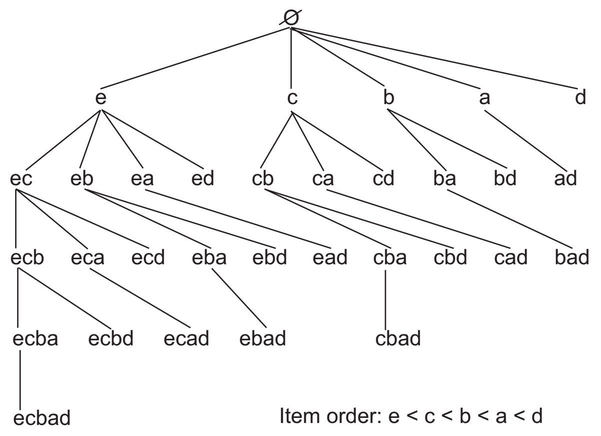

Figure 8: Set-Enumeration Tree

Definition 11. *Given a set-enumeration tree, an itemset represented by a node is called an extension of an itemset represented by an ancestor node of the node. For an itemset containing k items, its extension containing (k+i) items is called an i-extension of the itemset.*

Property 2. *If X' is an extension of X, (X'−X)=(X'/X).*
Rationale. Any extension of X is a combination of X with the item(s) after X.

For example, in Fig. 8, itemsets {eba} and {ebd} are the 1-extensions of {eb}, and {ebad} is the 2-extension of {eb}. Starting from the root of a set-enumeration tree, for an itemset, HUI-Miner first checks all of its 1-extensions by constructing their utility-lists. After identifying and outputting high utility itemsets from the extensions, HUI-Miner recursively processes promising extensions one by one and gives up the others. The question is: what are "promising" extensions?

4.2 Pruning Strategy

Exhaustive search can discover all high utility itemsets but is excessively time-consuming, because the numbers of items are large for many databases. For a database with n items, exhaustive search has to check 2^n itemsets.

To reduce the search space, we can exploit the iutils and rutils in the utility-list of an itemset. The sum of all the iutils in the utility-list of an itemset is the utility of the itemset according to Definition 5, and thus the itemset is high utility if the sum exceeds a given minutil. The sum of all the iutils and rutils in the utility-list provides HUI-Miner with the key information about whether the itemset should be pruned or not.

LEMMA 1. *Given the utility-list of itemset X, if the sum of all the iutils and rutils in the utility-list is less than a given "minutil", any extension X' of X is not high utility.*

PROOF. For $\forall$ *transaction* $t \supseteq X'$:

$$\because X' \text{ is an extension of } X \implies (X' - X) = (X'/X)$$
$$X \subset X' \subseteq t \implies (X'/X) \subseteq (t/X)$$

$$\begin{aligned}\therefore u(X',t) &= u(X,t) + u((X'-X),t)\\ &= u(X,t) + u((X'/X),t)\\ &= u(X,t) + \sum_{i\in(X'/X)} u(i,t)\\ &\le u(X,t) + \sum_{i\in(t/X)} u(i,t)\\ &= u(X,t) + ru(X,t),\end{aligned}$$

suppose $id(t)$ denotes the tid of transaction *t*, *X.tids* denotes the tid set in the utility-list of *X*, and *X'.tids* that in *X'*, then:

$$\begin{aligned}\because X \subset X' &\implies X'.tids \subseteq X.tids\\ \therefore\ u(X') &= \sum_{id(t)\in X'.tids} u(X',t)\\ &\le \sum_{id(t)\in X'.tids} (u(X,t) + ru(X,t))\\ &\le \sum_{id(t)\in X.tids} (u(X,t) + ru(X,t))\\ &< minutil.\end{aligned}$$

□

For example, consider the utility-lists in Fig. 6(b). Itemset {ec} should be pruned because the sum of all the iutils and rutils in its utility-list, i.e., 24, is less than the minutil, i.e., 30. Therefore, there is no need to check the 7 extensions of itemset {ec} (see Fig. 8).

4.3 HUI-Miner Algorithm

Algorithm 2 shows the pseudo-code of HUI-Miner. For each utility-list *X* in *ULs* (the second parameter), if the sum of all the iutils in *X* exceeds *minutil*, and then the extension associated with *X* is high utility and outputted. According to Lemma 1, only when the sum of all the iutils and rutils in *X* exceeds *minutil* should it be processed further. When the initial utility-lists are constructed from a database, they are sorted and processed in transaction-weighted-utility-ascending order (see Section 3.1). Therefore, all the utility-lists in *ULs* are ordered as the initial utility-lists are. To explore the search space, the algorithm intersects *X* and each utility-list *Y* after *X* in *ULs*. Suppose *X* is the utility-list of itemset *Px* and *Y* that of itemset *Py*, and then *construct*(*P.UL*, *X*, *Y*) in line 8 is to construct the utility-list of itemset *Pxy* as stated in Algorithm 1. Finally, the set of utility-lists of all the 1-extensions of itemset *Px* is recursively processed. Given a database and a minutil,

Algorithm 2: *HUI-Miner* Algorithm

```
Input: P.UL, the utility-list of itemset P, initially
       empty;
       ULs, the set of utility-lists of all P's
       1-extensions;
       minutil, the minimum utility threshold.
Output: all the high utility itemsets with P as prefix.
1  foreach utility-list X in ULs do
2      if SUM(X.iutils) ≥ minutil then
3          output the extension associated with X;
4      end
5      if SUM(X.iutils)+SUM(X.rutils) ≥ minutil then
6          exULs = NULL;
7          foreach utility-list Y after X in ULs do
8              exULs = exULs+Construct(P.UL, X, Y);
9          end
10         HUI-Miner(X, exULs, minutil);
11     end
12 end
```

after the initial utility-lists *IULs* are constructed, *HUI-Miner*(∅, IULs, minutil) can mine all high utility itemsets.

4.4 Implementation Details

The sums of the iutils and rutils in the utility-list of an itemset can be computed by scanning the utility-list. To avoid utility-list scan, in the process of constructing a utility-list, HUI-Miner simultaneously accumulates the iutils and rutils in the utility-list. In addition, there is also no need to bind each itemset to its utility-list. The itemsets represented by all child nodes of a node in a set-enumeration tree have the same prefix itemset. Therefore, for a 1-extension, its extended item can be separated from its prefix itemset. We slightly modify the utility-list structure when implementing HUI-Miner. For example, the utility-lists in Fig. 7(b) are implemented as those showed in Fig. 9. The first line in a utility-list stores the extended item and the sums of the iutils and rutils, and the prefix itemset is stored independently.

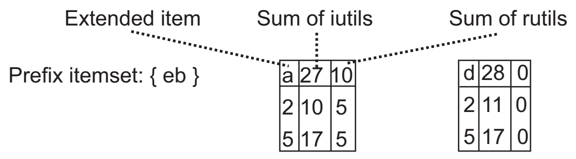

Figure 9: Utility-List Implementation

Another important detail is the processing order of items. In previous algorithms, such as IHUPTWU and UP-Growth, items are sorted in transaction-weighted-utility-descending order, which can reduce the size of prefix-trees used in these algorithms. However, IHUPTWU and UP-Growth process items in transaction-weighted-utility-ascending order. The processing order of items can result in the decrease in the explored scope of the search space and thus speed an algorithm up [15]. HUI-Miner adopts utility-lists as data structure, and the size of utility-lists is constant, no matter what order items are sorted in. Therefore, in HUI-Miner, items are sorted in transaction-weighted-utility-ascending order, and more important, processed in the same order.

Database	Size(kB)	#Trans	#Items	AvgLen	MaxLen
Accidents	59663	340183	468	33.8	51
Chain	63573	1112949	46086	7.3	170
Chess	591	3196	75	37	37
Kosarak	49859	990002	41270	8.1	2498
Mushroom	963	8124	119	23	23
Retail	6076	88162	16470	10.3	76
T10I4D100K	6144	100000	870	10.1	29
T40I10D100K	23796	100000	942	39.6	77

Figure 10: Statistical Information about Databases

5. EXPERIMENTS

To evaluate the performance of HUI-Miner, we have done extensive experiments on various databases, in which HUI-Miner is compared with the state-of-the-art mining algorithms. In this section, experimental results are reported and discussed.

5.1 Experimental Setup

Besides HUI-Miner, our experiments include the following algorithms: IHUPTWU (the fastest one among the algorithms proposed in [5]), UP-Growth [23], and UP-Growth+ [22]. The main procedure of IHUPTWU has been introduced in Section 2.2. On the basis of IHUPTWU, UP-Growth incorporates four strategies to lessen the estimated utilities of candidate itemsets and thus reduces the number of candidates. UP-Growth+, an improved UP-Growth algorithm, can generate fewer candidate itemsets than UP-Growth for a mining task. The less the number of candidate itemsets is, the less the costs of the generation and utility computation of candidates are. The three state-of-the-art algorithms had been proven to be superior to other algorithms, such as Two-Phase [18], ShFSM [12], DCG [11], FUM [14], and DCG+ [14]. Further, we optimized the three algorithms by transforming a mined database into a database view similar to that in Fig. 4. The view is implemented in memory, which can not only reduce the size of the database but also speed utility computation up.

The four algorithms were implemented in C++ language, used the same libraries, and were compiled using g++ (version 4.3.2). The experiments were performed on a 2.83GHz PC machine (Intel Core2 Q9500) with 4GB of memory, running on a Debian (Linux 2.6.26) operating system.

Eight databases were used in our experiments. Database *chain* was downloaded from NU-MineBench 2.0 [2], in which transaction records taken from a major grocery store chain in California are contained. The other databases were downloaded from FIMI Repository [1]. Databases *accidents*, *chess*, *kosarak*, *mushroom*, and *retail* are real. Synthetic databases *T10I4D100K* and *T40I10D100K* were generated by IBM Quest Synthetic Data Generation Code. Other than *chain*, the other databases do not provide item utility (external utility) and item count for each transaction (internal utility). Like the performance evaluation of previous algorithms [5, 23, 22], external utilities for items are generated between 0.01 and 10 using a log-normal distribution and internal utilities for items are generated randomly ranging from 1 to 10. Fig. 10 shows the statistical information about these databases, including the size on disk, the number of transactions, the number of distinct items, the average number of items in a transaction, and the maximal number of items in the longest transaction(s).

5.2 Running Time

The running time of the four algorithms on all databases is depicted in Fig. 11. Running time was recorded by the "time" command, and it contains input time, CPU time, and output time. The output results of the four algorithms are the same for a mining task, and they were written to "/dev/null". We terminated a mining task, once its running time exceeds 10000 seconds.

When measuring running time, we varied the minutil for each database. The lower the minutil is, the larger the number of high utility itemsets is, and thus the more the running time is. For example, for database *chain* in Fig. 11(b), when the minutils are 0.004% and 0.009%, the numbers of high utility itemsets are 18480 and 4578, and the running times of HUI-Miner are 580.9 seconds and

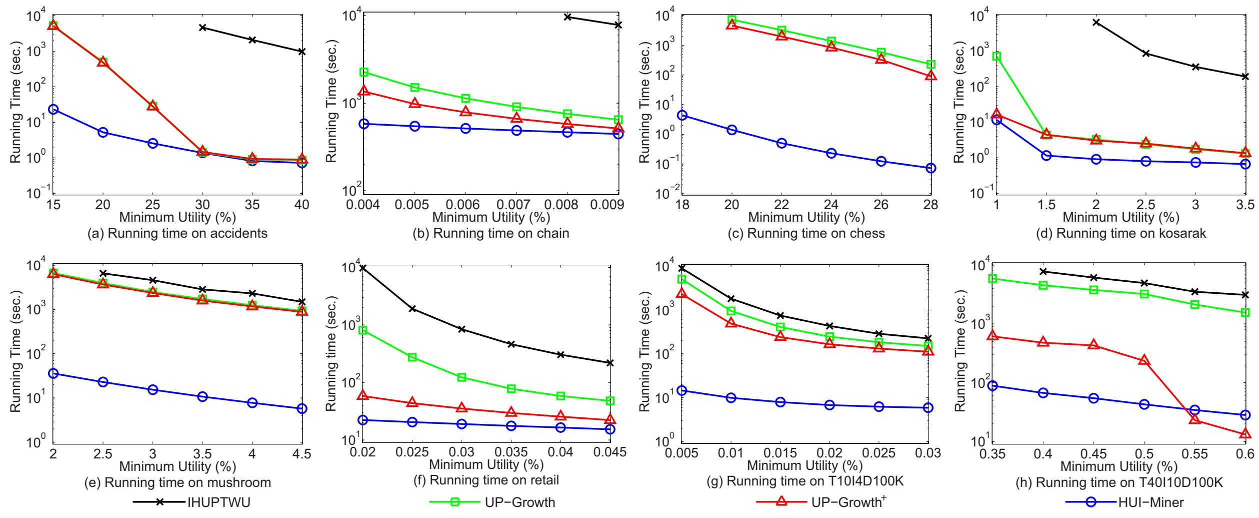

Figure 11: Running Time

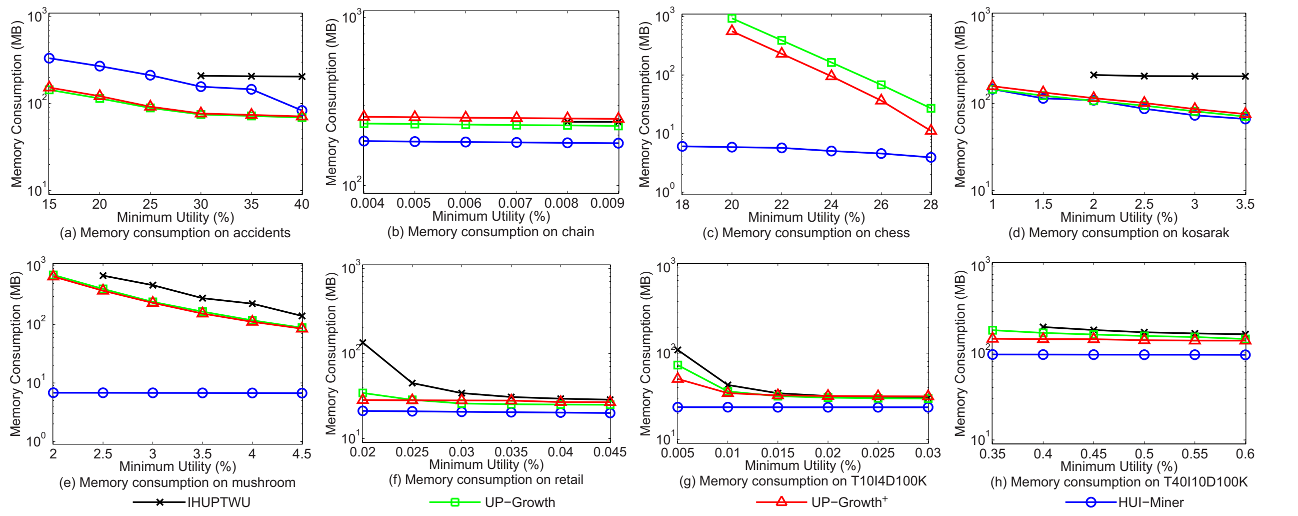

Figure 12: Memory Consumption

445.1 seconds, respectively. In addition, the curve for UP-Growth almost totally overlaps the curve for UP-Growth+ in Fig. 11(a); the running time of IHUPTWU for any minutil exceeds 10000 seconds for database *chess*, and thus there is no curve for IHUPTWU in Fig. 11(c).

For almost all databases and minutils, HUI-Miner performs the best. HUI-Miner is almost two orders of magnitude faster than the other algorithms for dense databases. For example, the running times of HUI-Miner and UP-Growth+ are 35.8 seconds and 6302.3 seconds for database *mushroom* in Fig. 11(e), when the minutil is 2%. In Fig. 11(h), HUI-Miner is slower than UP-Growth+ for high minutils, and we found out in this case that UP-Growth+ generates very few candidate itemsets (only 2007 candidates when the minutil is 0.6%); however, for low minutils, HUI-Miner is even an order of magnitude faster than UP-Growth+ (UP-Growth+ generates 178128 candidates when the minutil is 0.35%.). For most sparse databases, the performance superiority of HUI-Miner becomes very significant when the minutil decreases. For example, for *retail* in Fig. 11(f), the running times of HUI-Miner and IHUPTWU are 15.3 seconds and 219.1 seconds when the minutil is 0.045%, while their running times are 22.2 seconds and 9758.0 seconds when the minutil is reduced to 0.02%.

5.3 Memory Consumption

Fig. 12 shows the peak memory consumption of the four algorithms on all databases, in which each subfigure corresponds to a subfigure in Fig. 11. Peak memory consumption was recorded by the "massif" tool of the "valgrind" software [3].

Except for database *accidents* in Fig. 12(a), HUI-Miner always consumes less memory than the other algorithms. The reason is that these algorithms have to consume a very large amount of memory to store candidate high utility itemsets during their mining processes, while HUI-Miner does not. Generally, the memory consumption of these algorithms is proportional to the number of candidate itemsets they generate. For example, for database *T10I4D100K*, IHUPTWU generates 3826202 candidate itemsets and consumes 109.0 MB of memory while UP-Growth+ generates 1007150 candidate itemsets and consumes 50.22 MB of memory, when the minutil is 0.005%. The number of high utility itemsets is only 313509 for the mining task. HUI-Miner neither generates nor stores candidate itemsets, and thus it consumes only 23.62 MB of memory.

Another observation is that UP-Growth+ consumes more memory than UP-Growth in some cases, for example, in Fig. 12(b) and (d), although UP-Growth+ always generates fewer candidate itemsets than UP-Growth. It is because that each node in the prefix-trees used in UP-Growth+ holds more information than that in the prefix-trees used in UP-Growth [22]. When a database is sparse and large, the size of a corresponding prefix-tree is relatively large, while the number of candidate itemsets is relatively small. For example, the size of sparse database *kosarak* is 49859KB, but the numbers of candidate itemsets are only 80 and 74 for UP-Growth and UP-Growth+ when the minutil is 1.5%.

5.4 Processing Order of Items

The processing order of items significantly influences the performance of a high utility itemset mining algorithm [5]. As IHUPTWU, UP-Growth, and UP-Growth+ do, HUI-Miner processes items in transaction-weighted-utility-ascending order (see Section 4.4). To get the knowledge of the performance difference for different processing orders, we tested the running time of HUI-Miner on condition that items are processed in transaction-weighted-utility-

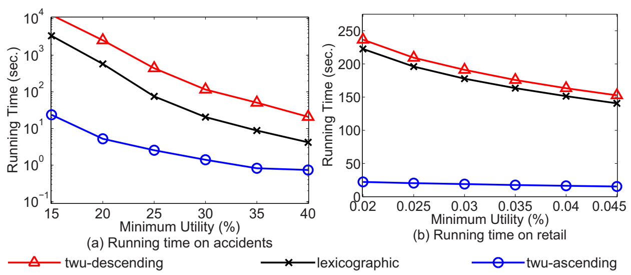

Figure 13: Different Item Orders

Accidents	15%	20%	25%	30%	35%	40%
IHUPTWU	2953047	978170	378955	163363	74144	35113
UP-Growth	184239	18761	1215	34	1	0
UP-Growth+	178732	18192	1193	34	1	0
HUI	280	0	0	0	0	0
Chain	0.004%	0.005%	0.006%	0.007%	0.008%	0.009%
IHUPTWU	43971025	9478232	739627	558254	429745	345648
UP-Growth	124509	82403	61215	48198	39637	33656
UP-Growth+	72560	51523	40725	33966	29274	25868
HUI	18480	12244	9040	6920	5585	4578
Chess	18%	20%	22%	24%	26%	28%
IHUPTWU	453506887	283147997	181541329	118826006	79065869	53468011
UP-Growth	50226806	22578752	9891125	4242057	1786383	702604
UP-Growth+	31670472	13725409	5795829	2464762	957940	273424
HUI	34870	4872	230	0	0	0
Kosarak	1%	1.5%	2%	2.5%	3%	3.5%
IHUPTWU	---	---	246648	35740	14244	6979
UP-Growth	27864	80	38	31	18	12
UP-Growth+	660	74	38	31	18	12
HUI	48	20	15	10	8	8
Mushroom	2%	2.5%	3%	3.5%	4%	4.5%
IHUPTWU	29593409	17342263	11985059	7396747	5981219	3741959
UP-Growth	17594549	10295610	6383808	4361711	3122163	2349568
UP-Growth+	16681744	9602355	6145020	4037680	2942177	2246587
HUI	3583596	1879818	1059350	640404	400137	256988
Retail	0.02%	0.025%	0.03%	0.035%	0.04%	0.045%
IHUPTWU	3280836	695675	308951	170144	111500	79647
UP-Growth	304128	112917	55037	35925	27208	22436
UP-Growth+	27917	21047	17006	14279	12329	10781
HUI	8723	6026	4377	3340	2676	2212
T10I4D100K	0.005%	0.01%	0.015%	0.02%	0.025%	0.03%
IHUPTWU	3826202	802793	335849	197697	133848	105685
UP-Growth	2155382	421526	188067	115329	85544	70147
UP-Growth+	1007150	226448	114948	78282	61451	51523
HUI	313509	81562	51457	40898	34092	29176
T40I10D100K	0.35%	0.4%	0.45%	0.5%	0.55%	0.6%
IHUPTWU	4214124	2229232	1752536	1410616	1240654	1105830
UP-Growth	1703398	1298224	1079166	912479	759925	541878
UP-Growth+	178128	141242	127305	71366	5342	2007
HUI	20448	4618	328	147	28	19

Figure 14: Number of Candidate Itemsets & Number of Resultant High Utility Itemsets

descending order, lexicographic order, and transaction-weighted-utility-ascending order, respectively. Fig. 13 shows the experimental results on databases *accidents* and *retail*.

As we can see, the transaction-weighted-utility-ascending order leads to the best performance. The reason is that the processing order of items is capable of reducing the number of sets of utility-lists for a mining task. To comprehend the reason in depth, one can consult the related work in [15, 16].

5.5 Discussion

From above experiments, we can observe that HUI-Miner outperforms the state-of-the-art algorithms.

To mine high utility itemsets, almost all existing algorithms first generate candidate high utility itemsets and subsequently compute the exact utility of each candidate to identify high utility itemsets. To improve performance, previous studies focus on how to reduce the number of candidates, which can lead to the decrease in the costs of both candidate generation and utility computation.

Fig. 14 shows the number of candidate itemsets the three algorithms generate and the number of resultant high utility itemsets. For database *kosarak*, when the minutils are 1% and 1.5%, the times of candidate generation are so much ($\gg$100000 seconds) that we had to terminate the two tests. From Fig. 11, Fig. 12, and Fig. 14, one can observe that the number of candidate itemsets generated by an algorithm is proportional to the running time and memory consumption of the algorithm. The state-of-the-art algorithms have been able to efficiently reduce the number of candidates. However, the number is still far larger than the number of resultant high utility itemsets in most cases. For example, IHUPTWU, UP-Growth, and UP-Growth+ generate 558254, 48198, and 33966 candidate itemsets, when the minutil is 0.007% for database *chain*, but the number of resultant high utility itemsets is only 6920.

Using the utility-list structure, the HUI-Miner algorithm can mine high utility itemsets without candidate generation. The distinct advantage of HUI-Miner is that it avoids the costly candidate generation and utility computation. For the above example, IHUPTWU, UP-Growth, and UP-Growth+ have to process 551334 ($= 558254 - 6920$), 41278 ($= 48198 - 6920$), and 27046 ($= 33966 - 6920$) candidate itemsets, respectively. These algorithms not only generate these itemsets but also compute their exact utilities on 1112949 transactions. However, these itemsets are discarded finally. The potential advantage of HUI-Miner is that a large amount of memory is saved. For example, the size of database *mushroom* is only 0.92MB, but UP-Growth and UP-Growth+ generate 17594549 and 16681744 candidate itemsets, and consume 699.9 MB and 658.7MB of memory, respectively (when the minutil is 2%.), and a large amount of memory is used to store candidate itemsets. Although the algorithms can be modified to swap candidate itemsets to disk, the disk space requirement is also considerable, and moreover, the algorithms' performance will be degraded.

6. CONCLUSION

In this paper, we have proposed a novel data structure, utility-list, and developed an efficient algorithm, HUI-Miner, for high utility itemset mining. Utility-lists provide not only utility information about itemsets but also important pruning information for HUI-Miner. Previous algorithms have to process a very large number of candidate itemsets during their mining processes. However, most candidate itemsets are not high utility and are discarded finally. HUI-Miner can mine high utility itemsets without candidate generation, which avoids the costly generation and utility computation of candidates. We have studied the performance of HUI-Miner in comparison with the state-of-the-art algorithms on various databases. Experimental results show that HUI-Miner gains significant performance improvement over these algorithms in terms of both running time and memory consumption.

7. ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers. Their helpful comments and suggestions have improved the quality of the paper.

8. REFERENCES

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

[2] NU-MineBench: A Data Mining Benchmark Suite. `http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html`, 2012.

[3] Valgrind: A GPL'd System for Debugging and Profiling Linux Programs. `http://valgrind.org/`, 2012.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. Int'l Conf. Very Large Data Bases*, pages 487–499, 1994.

[5] 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*, 21(12):1708–1721, 2009.

[6] B. Barber and H. J. Hamilton. Extracting share frequent itemsets with infrequent subsets. *Data Mining and Knowledge Discovery*, 7(2):153–185, 2003.

[7] A. Ceglar and J. F. Roddick. Association mining. *ACM Computing Surveys*, 38(2), 2006.

[8] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

[9] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach*. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[10] J. Hu and A. Mojsilovic. High-utility pattern mining: A method for discovery of high-utility item sets. *Pattern Recognition*, 40(11):3317 – 3324, 2007.

[11] Y.-C. Li, J.-S. Yeh, and C.-C. Chang. Direct candidates generation: A novel algorithm for discovering complete share-frequent itemsets. In *Proc. Fuzzy Systems and Knowledge Discovery*, pages 551–560, 2005.

[12] Y.-C. Li, J.-S. Yeh, and C.-C. Chang. Efficient algorithms for mining share-frequent itemsets. In *Proc. World Congress of Int'l. Fuzzy Systems Association*, pages 534–539, 2005.

[13] Y.-C. Li, J.-S. Yeh, and C.-C. Chang. A fast algorithm for mining share-frequent itemsets. In *Proc. Asia-Pacific Web Conf.*, pages 417–428, 2005.

[14] Y.-C. Li, J.-S. Yeh, and C.-C. Chang. Isolated items discarding strategy for discovering high utility itemsets. *Data & Knowledge Engineering*, 64(1):198–217, 2008.

[15] G. Liu, H. Lu, W. Lou, Y. Xu, and J. X. Yu. Efficient mining of frequent patterns using ascending frequency ordered prefix-tree. *Data Mining and Knowledge Discovery*, 9(3):249–274, 2004.

[16] G. Liu, H. Lu, J. X. Yu, W. Wang, and X. Xiao. Afopt: An efficient implementation of pattern growth approach. In *Proc. IEEE Int'l Conf. Data Mining Workshop Frequent Itemset Mining Implementations*, 2003.

[17] Y. Liu, W.-K. Liao, and A. Choudhary. A fast high utility itemsets mining algorithm. In *Proc. Utility-Based Data Mining Workshop*, pages 90–99, 2005.

[18] Y. Liu, W.-K. Liao, and A. N. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining*, pages 689–695, 2005.

[19] R. Rymon. Search through systematic set enumeration. In *Proc. Int'l Conf. Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.

[20] A. Soulet and B. Crémilleux. Adequate condensed representations of patterns. *Data Mining and Knowledge Discovery*, 17:94–110, 2008.

[21] A. Soulet, C. Raïssi, M. Plantevit, and B. Crémilleux. Mining dominant patterns in the sky. In *Proc. IEEE Int'l Conf. Data Mining*, pages 655 –664, 2011.

[22] 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*, 2012, doi: 10.1109/TKDE.2012.59.

[23] V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu. Upgrowth: An efficient algorithm for high utility itemset mining. In *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, pages 253–262, 2010.

[24] C. W. Wu, P. Fournier-Viger, P. S. Yu, and V. S. Tseng. Efficient mining of a concise and lossless representation of high utility itemsets. In *Proc. IEEE Int'l Conf. Data Mining*, pages 824 –833, 2011.

[25] H. Yao, H. J. Hamilton, and C. J. Butz. A foundational approach to mining itemset utilities from databases. In *Proc. SIAM Int'l Conf. Data Mining*, 2004.

[26] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.