# Efficient Algorithms for High Utility Itemset Mining Without Candidate Generation

**Chapter** · January 2019

DOI: 10.1007/978-3-030-04921-8_5



**3 authors**, including:

Mengchi Liu
Carleton University
**126** PUBLICATIONS **1,273** CITATIONS
SEE PROFILE

Philippe Fournier Viger
Harbin Institute of Technology (Shenzhen)
**320** PUBLICATIONS **6,454** CITATIONS
SEE PROFILE

# Efficient Algorithms for High Utility Itemset Mining Without Candidate Generation


**Jun-Feng Qu, Mengchi Liu and Philippe Fournier-Viger**

**Abstract** High utility itemsets are sets of items having a high utility or profit in a database. Efficiently discovering high utility itemsets plays a crucial role in real-life applications such as market analysis. Traditional high utility itemset mining algorithms generate candidate itemsets and subsequently compute the exact utilities of these candidates. These algorithms have the drawback of generating numerous candidates most of which are discarded for having a low utility. In this paper, we propose two algorithms, called HUI-Miner (High Utility Itemset Miner) and HUI-Miner*, for high utility itemset mining. HUI-Miner uses a novel utility-list structure to store both utility information about itemsets and heuristic information for search space pruning. The utility-list of items allows to directly derives the utility-lists of other itemsets and calculate their utilities without scanning the database. By avoiding candidate generation, HUI-Miner can efficiently mine high utility itemsets. To further speed up the construction of utility-lists, HUI-Miner* introduces an improved structure called utility-list* and an horizontal method to construct utility-lists*. Experimental results show that the proposed algorithms are several orders of magnitude faster than the state-of-the-art algorithms, reduce memory consumption, and that HUI-Miner* outperforms HUI-Miner especially for sparse databases.

J.-F. Qu (✉)
School of Computer Engineering, Hubei University of Arts and Science, Xiangyang 441053, China
e-mail: qmxwt@163.com

M. Liu
School of Computer Science, Carleton University, Ottawa, ON K1S 5B6, Canada
e-mail: mengchi@scs.carleton.ca

P. Fournier-Viger
School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China
e-mail: philfv8@yahoo.com



P. Fournier-Viger et al. (eds.), *High-Utility Pattern Mining*,
Studies in Big Data 51, https://doi.org/10.1007/978-3-030-04921-8_5

## 1 Introduction

Current database techniques facilitate the storage and usage of massive data from business corporations, scientific organizations, and governments. Research on methods for obtaining valuable information from various databases has received considerable attention and consequently many data mining problems were proposed. One of the most famous problems is frequent itemset mining [1–3].

A set of items appearing in a database is called an itemset, the frequency of which is measured by its support, i.e., the number of transactions containing the itemset in the database. If the support of an itemset exceeds a user-specified minimum support threshold, the itemset is considered frequent. Given a database and a threshold, the problem is to find the complete set of frequent itemsets from the database. Most frequent itemset mining algorithms employ a downward closure property [4], which states that all supersets of an infrequent itemset are infrequent and all subsets of a frequent itemset are frequent. This property provides a powerful pruning strategy to algorithms. Once an itemset mining algorithm identifies an infrequent itemset, its supersets no longer need to be considered. For example, for a database with $n$ items, after the algorithm identifies an infrequent itemset containing $k$ items, there is no need to check all of its $2^{(n-k)} - 1$ supersets.

Mining frequent itemsets takes the presence and absence of items into account, but 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 count indicating the purchase quantity of the item. Consider the sample database composed of Tables 1 and 2. There are seven items in the utility table and eight transactions in the transaction table. To compute the support of an itemset, an algorithm only uses the information of the first two columns in the transaction table, whereas the information of both the utility table and the last two columns in the transaction table is not considered. However, an itemset with a high support may have a low utility, or vice versa. For example, the support and utility of itemset {bd} appearing in T2, T3, and T6, are 3 and 16, respectively, and those of itemset {de} appearing in T3 and T6, are 2 and 17 (see Sect. 2.1 for an explanation of how utility is calculated). In some applications such as market analysis, one may be more interested in itemset utility rather than support. Frequent itemset mining algorithms cannot evaluate the utilities of itemsets.

Generally, itemsets with utilities no less than a user-specified minimum utility threshold are valuable, and they are called "high utility itemsets". Mining all high utility itemsets from a database is intractable, because the downward closure property does not hold 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 may increase, decrease or stay the same. For example, for the sample database, the supports of {a}, {ab}, {abc}, and {abcd} are 4, 3, 1, and 1, but the utilities of these itemsets are 32, 31, 15, and 19, respectively. If the minimum utility threshold is set to 18, the high utility itemset {abcd} contains both

the high utility {ab} and the low utility {abc}. Therefore, pruning strategies used in frequent itemset mining cannot be applied for high utility itemset mining.

Recently, a number of high utility itemset mining algorithms have been proposed [5–11]. Most of them adopt a similar framework: they first generate candidate high utility itemsets from a database, and then compute the exact utilities of the candidates to identify high utility itemsets. However, the algorithms often generate a very large number of candidate itemsets and thereby are confronted with two problems: (1) excessive running time for both candidate generation and exact utility computation; (2) a high memory requirement for storing candidates. Algorithms that generates too many candidates can fail to terminate due to a lack of memory, and their performance can deteriorate due to thrashing.

To solve the above problems, this paper proposes two algorithms for high utility itemset mining.[1] The contributions are as follows.

- A novel structure called *utility-list* is proposed. Utility-lists store not only utility information about itemsets but also heuristic information for search space pruning.
- An efficient algorithm called *HUI-Miner* (*High Utility Itemset Miner*) is developed. HUI-Miner uses utility-lists constructed from a database to mine high utility itemsets and, different from traditional high utility itemset mining algorithms, does not generate candidate itemsets.
- Furthermore, an improved algorithm called *HUI-Miner** is proposed, which uses a modified utility-list structure called *utility-list**. HUI-Miner and HUI-Miner* mine high utility itemsets by recursively constructing utility-lists and utility-lists*, respectively. However, utility-list* construction is more efficient than utility-list construction, especially for sparse databases.
- Extensive experiments on various databases were performed to compare HUI-Miner and HUI-Miner* with state-of-the-art algorithms. Experimental results are presented, which show that the proposed algorithms outperform these algorithms.

The rest of this paper is organized as follows. Section 2 introduces the background. Section 3 presents the proposed data structures and algorithms. Section 4 reports experimental results, which are then discussed in Sect. 5. Finally, concluding remarks are given in Sect. 6.

## 2 Background

This section gives a formal description of the search space for the high utility itemset mining problem and subsequently introduces previous solutions to the problem.

[1] This is an extension of the conference paper "Mining high utility itemsets without candidate generation" published in the proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012).

## 2.1 Preliminaries

Let $\mathscr{I} = \{i_1, i_2, i_3, \ldots, i_n\}$ be a set of items and *DB* be a database composed of a utility table and a transaction table. Each item in $\mathscr{I}$ has a utility value in the utility table (a positive number). Each transaction $T$ in the transaction table has a unique identifier (*Tid*) and is a subset of $\mathscr{I}$, and each item in $T$ is associated with a count value. An itemset is a subset of $\mathscr{I}$ and is called a $k$-itemset if it contains $k$ items.

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

**Definition 2** The internal utility of an item i in a 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 an item i in a transaction T, denoted as u(i, T), is the product of iu(i, T) and eu(i), that is u(i, T) = iu(i, T) × eu(i).

For example, in Table 1, eu(e) = 3, iu(e, T6) = 2, and u(e, T6) = iu(e, T6) × eu(e) = 2 × 3 = 6.

**Definition 4** The utility of an itemset X in a transaction T, denoted as u(X, T), is the sum of the utilities of items from X in T if T contains X, and 0 otherwise, that is u(X, T) = $\sum_{i \in X \wedge X \subseteq T} u(i, T)$.

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

For example, in Table 2, u({ae}, T3) = u(a, T3) + u(e, T3) = 4 × 2 + 1 × 3 = 11, and u({ae}) = u({ae}, T3) + u({ae}, T6) = 11 + 16 = 27.

**Table 1** A utility table

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

**Table 2** A transaction table

| TID | Transaction | Count | TU |
|---|---|---|---|
| T1 | { c, d } | { 2, 1 } | 8 |
| T2 | { b, d, g } | { 1, 1, 1 } | 6 |
| T3 | { a, b, c, d, e } | { 4, 1, 3, 1, 1 } | 22 |
| T4 | { c, e, f } | { 2, 1, 1 } | 9 |
| T5 | { d } | { 1 } | 4 |
| T6 | { a, b, d, e } | { 5, 2, 1, 2 } | 22 |
| T7 | { a, b, f } | { 3, 4, 2 } | 14 |
| T8 | { a, c } | { 4, 1 } | 10 |

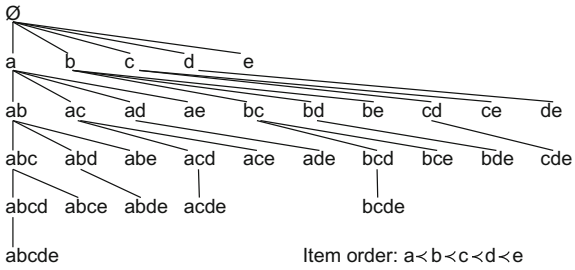



**Fig. 1** A set-enumeration tree

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

The last column of Table 2 indicates the utility of each transaction. For example, tu(T7) = u(a, T7) + u(b, T7) + u(f, T7) = 6 + 4 + 4 = 14. The total utility of the database is 95. Given a database and a user-specified minimum utility threshold denoted as *minutil*, an itemset $X$ is *a high utility itemset* if u($X$) exceeds minutil threshold. Note that the minutil threshold can equivalently be specified as a percentage of the utility of a database. The problem of finding the complete set of high utility itemsets from a database is called high utility itemset mining.

The problem's search space can be represented as a set-enumeration tree [12]. Given a set of items $\{i_1, i_2, \ldots, i_n\}$ and a total order on all items (suppose $i_1 \prec i_2 \prec \cdots \prec i_n$), a set-enumeration tree representing all itemsets can be constructed as follows. Firstly, the root of the tree representing the empty set is created; secondly, the $n$ child nodes of the root representing $n$ 1-itemsets are created, respectively; thirdly, for a node representing itemset $\{i_s \cdots i_e\}$ $(1 \leq s \leq e < n)$, 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 {a, b, c, d, e} and the lexicographical order, the set-enumeration tree representing all itemsets from $\mathscr{I}$ is depicted in Fig. 1.

**Definition 7** In a set-enumeration tree, each itemset is represented by a node. A node is said to be an extension of any of its ancestor nodes. For a $k$-itemset, its extension containing $(k + i)$ items is called an $i$-extension of the itemset.

For example, in Fig. 1, {abc} and {abd} are two 1-extensions of {ab}, and {abcd} is a 2-extension of {ab}.

## 2.2 Related Work

Before the high utility itemset mining problem was formally defined [5] as above, a variant of the problem had been studied, namely the problem of mining share frequent

**Table 3** Transaction-weighted utility

| Itemset | {a} | {b} | {c} | {d} | {e} | {f} | {g} |
|---|---|---|---|---|---|---|---|
| TWU | 68 | 64 | 49 | 62 | 53 | 23 | 6 |

itemsets [13–15], in which the external utility of each item is invariably defined as 1. The ZP [13], ZSP [13], FSH [14], ShFSH [15], and DCG [16] algorithms for share frequent itemset mining can also be used to mine high utility itemsets. Since the downward closure property cannot be applied, Liu et al. proposed an important property [17] for search space pruning in high utility itemset mining.

**Definition 8** The transaction-weighted utility (abbreviated as TWU) of an itemset X in *DB*, denoted as twu(X), is the sum of the utilities of all transactions containing X in *DB*, that is twu(X) = $\sum_{T \in DB \wedge X \subseteq T} tu(T)$.

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

Table 3 gives the TWUs of all 1-itemsets in the sample database. For example, itemset {f} is contained in T4 and T7, and thus twu({f}) = tu(T4) + tu(T7) = 9 + 14 = 23. Suppose that minutil is equal to 30. Then, all supersets of {f} are not high utility itemsets according to Property 1 and thereby are not required to be checked. The Two-Phase algorithm was the first to apply Property 1 to prune the search space [6, 17]. Afterwards, an isolated items discarding strategy was proposed [7], which can be incorporated in the above algorithms to improve their performances. For example, the FUM and DCG+ algorithms using that strategy outperform ShFSH and DCG, respectively [7].

ZP, ZSP, FSH, ShFSH, DCG, Two-Phase, FUM, and DCG+ mine high utility itemsets as the Apriori algorithm mines frequent itemsets [4]. Let there be a database and a minutil threshold. All 1-itemsets are first considered as candidate high utility itemsets. After overestimating the utilities of the candidates by performing a database scan, the algorithms delete unpromising 1-itemsets and generate candidate 2-itemsets from the remaining 1-itemsets. After overestimating the utilities of the candidates by another database scan, the algorithms delete unpromising 2-itemsets and generate candidate 3-itemsets from the remaining 2-itemsets. The procedure is performed repeatedly until no candidate itemset is generated. Finally, except DCG and DCG+, these algorithms compute the exact utilities of all remaining candidates by an additional database scan to identify high utility itemsets (DCG and DCG+ compute the exact utility in each database scan). Besides the two problems mentioned in Sect. 1, these algorithms have the drawback of repeatedly scanning a database.

Algorithms based on the FP-Growth algorithm [18] show better performance, such as IHUPTWU [8], UP-Growth [9], and UP-Growth+ [10]. Firstly, these algorithms transform a database into a prefix-tree, which maintains the utility information about itemsets. Secondly, for each item of the tree, if it is estimated to be valuable, that is, if

there likely is high utility itemsets containing the item, the algorithms will 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 compute the exact utilities of all candidates by performing database scan to identify high utility itemsets. By speeding up candidate generation and decreasing the number of candidate itemsets, these algorithms outperform Apriori-based algorithms. Even so, the number of candidates generated by these algorithms is still far larger than the number of high utility itemsets in most cases. Hence, generating low utility candidates and computing their exact utilities result in a huge waste of space and time.

Some studies have also considered mining an approximate set of all high utility itemsets [19, 20], a condensed set of all high utility itemsets [21, 22], and a set of top-k high utility itemsets [23]. In this study, we focus on the problem of mining the complete set of all high utility itemsets from a database, and present algorithms that discover high utility itemsets without candidate generation.

## 3 Mining High Utility Itemsets

### *3.1 Utility-List Structure*

To mine high utility itemsets, traditional high utility itemset mining algorithms are directly applied to a database. FP-Growth-based algorithms generate candidate itemsets from prefix-trees, but they have to scan the database to compute the exact utilities of candidates. This subsection proposes a utility-list structure to maintain the utility information about itemsets to avoid repeatedly scanning a database to mine high utility itemsets.

#### 3.1.1 Initial Utility-Lists

In the HUI-Miner algorithm, each itemset is associated with a utility-list. The utility-lists of 1-itemsets are the initial utility-lists, which can be constructed by two database scans.

During the first database scan, the TWUs of all items are accumulated. If the TWU of an item is less than the given minutil threshold, the item is no longer considered in the subsequent mining process according to Property 1. Items having TWU values that are no less than the minutil are sorted in order of ascending TWU. For the sample database, suppose that minutil is set to 38 (40% of the total utility). In that case, the algorithm no longer takes items f and g into consideration after the first database scan. The remaining items are sorted as: $c \prec e \prec d \prec b \prec a$.

**Table 4** Database view

| Tid | Item | Util. | Item | Util. | Item | Util. | Item | Util. | Item | Util. |
|---|---|---|---|---|---|---|---|---|---|---|
| T1 | c | 4 | d | 4 | | | | | | |
| T2 | d | 4 | b | 1 | | | | | | |
| T3 | c | 6 | e | 3 | d | 4 | b | 1 | a | 8 |
| T4 | c | 4 | e | 3 | | | | | | |
| T5 | d | 4 | | | | | | | | |
| T6 | e | 6 | d | 4 | b | 2 | a | 10 | | |
| T7 | b | 4 | a | 6 | | | | | | |
| T8 | c | 2 | a | 8 | | | | | | |

**Definition 9** A transaction is considered as "revised" after (1) all items having TWU values less than the given minutil threshold have been eliminated from the transaction; (2) the remaining items are sorted in order of ascending TWU.

During the second database scan, the algorithm revises each transaction to construct the initial utility-lists. The database view in Table 4 lists all revised transactions from the sample database. In the rest of this paper, a transaction is always considered as revised, and all items in an itemset are in order of ascending TWU.

**Definition 10** For any itemset X and transaction (or itemset) T such that $X \subseteq T$, the set of all items after the last item in X in T is denoted as T/X.

For example, consider the database view of Table 4, T3/{cd} = {ba} and T3/{e} = {dba}.

**Definition 11** The remaining utility of an itemset X in a transaction T, denoted as ru(X, T), is the sum of the utilities of all items in T/X in T in which X is contained, where $ru(X, T) = \sum_{i \in (T/X) \wedge X \subseteq T} u(i, T)$.

**Definition 12** The remaining utility of itemset X, denoted as ru(X), is the sum of the remaining utilities of X in all transactions containing X in *DB*, where $ru(X) = \sum_{T \in DB \wedge X \subseteq T} ru(X, T)$.

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

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

After the second database scan, the initial utility-lists constructed by HUI-Miner are as shown in Fig. 2. For example, consider the utility-list of {e}. In T3, u({e}, T3) = 3, ru({e}, T3) = u(d, T3) + u(b, T3) + u(a, T3) = 4 + 1 + 8 = 13, and thus the <3, 3, 13> element is in the utility-list of {e} ($< x, y, z >$ means <tid, iutil, rutil>

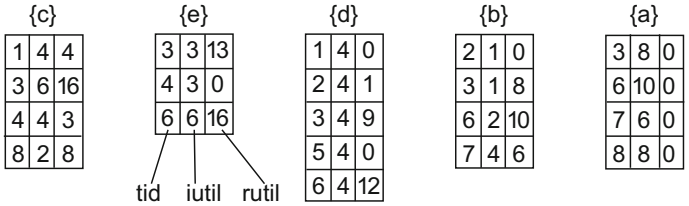



**Fig. 2** Initial utility-lists

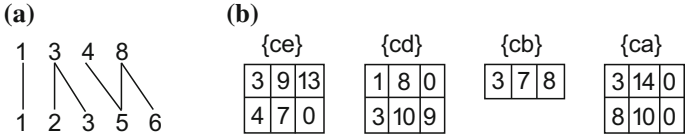



**Fig. 3** **a** Tid comparison. **b** Utility-lists of 2-itemsets

where $x$ represents transaction T$x$). In T4, u({e}, T4) = 3, ru({e}, T4) = 0, and thus element <4, 3, 0> is also in the utility-list. The last element is generated in the same manner.

### 3.1.2 Utility-Lists of 2-Itemsets

Without scanning the database, 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 compares the tids in the two utility-lists to identify common transactions. The identification process is a two-way comparison, because all tids in a utility-list are stored according to the order of ascending natural numbers. For example, the tid comparison between the utility-lists of itemsets {c} and {d} in Fig. 2 is illustrated in Fig. 3a.

For each common transaction $t$, the algorithm generates an element $E$ and subsequently appends 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 of the elements associated with $t$ in the utility-lists of $\{x\}$ and $\{y\}$. The rutil of $E$ is assigned as the rutil of the element associated with $t$ in the utility-list of $\{y\}$ ($x$ precedes $y$).

Figure 3b depicts the utility-lists of all the 2-itemsets having itemset {c} as prefix. For example, to construct the utility-list of itemset {ce}, the algorithm intersects the utility-list of {c}, i.e., {<1, 4, 4>, <3, 6, 16>, <4, 4, 3>, <8, 2, 8>}, and that of {e}, i.e., {<3, 3, 13>, <4, 3, 0>, <6, 6, 16>}, which results in {<3, 9, 13>, <4, 7, 0>}. One can observe from the database view of Table 4 that {ce} only appears in T3 and T4. In T3, u({ce}, T3) = u(c, T3) + u(e, T3) = 6 + 3 = 9, and ru({ce}, T3) = u(d, T3) + u(b, T3) + u(a, T3)= 4 + 1 + 8 = 13. Similarly, in T4, u({ce}, T4) = 4 + 3 = 7, and ru({ce}, T4) = 0.

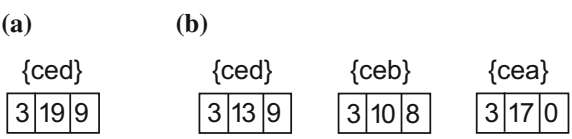



**Fig. 4** **a** An incorrect result. **b** Utility-lists of 3-itemsets

### 3.1.3 Utility-Lists of k-Itemsets (k ≥ 3)

To construct the utility-list of a $k$-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, consider the utility-list of {ced}, the direct intersection of the utility-lists of {ce} and {cd} in Fig. 3b results in the utility-list depicted in Fig. 4a. Itemset {ced} does appear in T3 as shown in Table 4, but the utility of the itemset in T3 is 13 rather than 19.

**Algorithm 1** *Construct(P.UL, Px.UL, Py.UL)*

**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*.
*Pxy.UL* = *NULL*
**foreach** element *Ex* ∈ *Px.UL* **do**
  **if** ∃*Ey*∈*Py.UL* **and** *Ex.tid*==*Ey.tid* **then**
    **if** *P.UL* is not empty **then**
      search such *E*∈*P.UL* that *E.tid*==*Ex.tid* *Exy* = <*Ex.tid*, *Ex.iutil*+*Ey.iutil*-*E.iutil*, *Ey.rutil*>
    **else**
      *Exy* = <*Ex.tid*, *Ex.iutil*+*Ey.iutil*, *Ey.rutil*>
    **end**
    append *Exy* to *Pxy.UL*
  **end**
**end**
**return** *Pxy.UL*

The reason for miscalculating the utility of {ced} in T3 is that the sum of the utilities of both {ce} and {cd} in T3 contains the utility of {c} in T3. Thus, this utility is counted twice. Generally, the utility of $\{i_1 \cdots i_{(k-2)} i_{(k-1)} i_k\}$ in T can be calculated as follows: 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).

In this way, the iutil of the element associated with T3 in the utility-list of {ced} is: u({ced}, T3) = u({ce}, T3) + u({cd}, T3) – u({c}, T3) = 9 + 10 – 6 = 13. The values of u({ce}, T3), u({cd}, T3), and u({c}, T3) can be obtained from the utility-lists of {ce}, {cd}, and {c}, respectively.

Suppose that itemsets *Px* and *Py* are the combinations of itemset *P* with items *x* and *y* (*x* precedes *y*), respectively, and *P.UL*, *Px.UL*, and *Py.UL* are the utility-lists of *P*, *Px*, and *Py*. Procedure 1 shows how to construct the utility-list of itemset *Pxy*. The

utility-list of a 2-itemset is constructed if *P.UL* is empty, that is, if *P* is empty (line 8), and the utility-list of a *k*-itemset (k ≥ 3) is constructed if *P.UL* is not empty (lines 5–6). Note that element *E* in line 5 can always be found out if *P.UL* is not empty, because each tid in either *Px.UL* or *Py.UL* derives from a tid in *P.UL*. According to Procedure 1, the constructed utility-lists of all the 3-itemsets having {ce} as prefix are shown in Fig. 4b.

## *3.2 The Proposed Method: HUI-Miner*

After constructing the initial utility-lists from a database, HUI-Miner can mine all high utility itemsets from the utility-lists in a manner similar to the way the Eclat algorithm mines frequent itemsets [24]. In this subsection, a pruning strategy used by HUI-Miner is introduced, and subsequently the pseudo-code and details of the algorithm are presented.

### 3.2.1 The Pruning Strategy

HUI-Miner searches for high utility itemsets in a set-enumeration tree in depth-first order. The items in the tree are sorted in order of ascending TWU. The following property holds for all itemsets represented by the tree.

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

Starting from the root of the set-enumeration tree, for an itemset, HUI-Miner first constructs the utility-lists of all 1-extensions of the itemset. After identifying and outputting high utility itemsets among these extensions by checking their utility-lists, HUI-Miner recursively processes promising extensions one by one and ignores unpromising extensions. But which extensions are "promising"?

To reduce the search space, HUI-Miner uses the iutils and rutils in utility-lists. The sum of all 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 that sum exceeds the minutil threshold. The sum of all rutils in the utility-list of an itemset is the remaining utility of the itemset according to Definition 12. The following lemma can be used to judge whether an itemset should be extended or not.

**Lemma 1** *If the sum of all iutils and rutils in the utility-list of an itemset X is less than the minutil threshold, any extension X' of X is not high utility.*

*Proof* For $\forall\, transaction\ T \supseteq X'$:

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

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

Let $id(T)$ denotes the tid of transaction $T$, $X.tids$ denotes the set of all tids in the utility-list of $X$, and $X'.tids$ that in the utility-list of $X'$, then:

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

For example, suppose that minutil is 38. Then, {c} should be extended according to Lemma 1, because the sum of all iutils and rutils in its utility-list (see Fig. 2) is 47, which is larger than minutil. However, all 1-extensions of {c} (see Fig. 3b) should not be extended according to the lemma, so there is no need to construct their utility-lists in Fig. 4b.

### 3.2.2 The Mining Procedure of HUI-Miner

The mining procedure of HUI-Miner is shown in Procedure 2, in which $Px$ and $Py$ are 1-extensions of an itemset $P$. $Px.UL$ and $Py.UL$ represents the utility-lists of $Px$ and $Py$, respectively. For each utility-list *Px.UL* in *ULs* (the second parameter), if the sum of all iutils in *Px.UL* exceeds *minutil*, *Px* is high utility and is output. According to Lemma 1, only when the sum of all iutils and rutils in *Px.UL* exceeds *minutil* should it be processed further. When initial utility-lists are constructed from a database, they are sorted and processed in order of ascending TWU. Therefore, all utility-lists in *ULs* follow the same order as the initial utility-lists. To explore the search space, the algorithm intersects *Px.UL* and each utility-list *Py.UL* after *Px.UL* in *ULs*. *Construct*(*P.UL*, *Px.UL*, *Py.UL*) in line 8 is a procedure that constructs the utility-list of itemset *Pxy* as stated in Procedure 1. Finally, the set of the utility-lists of all 1-extensions of *Px* is recursively processed. Given a database and a minutil threshold, after initial utility-lists *IULs* have been constructed, *Mine*(∅, IULs, minutil) outputs all high utility itemsets.

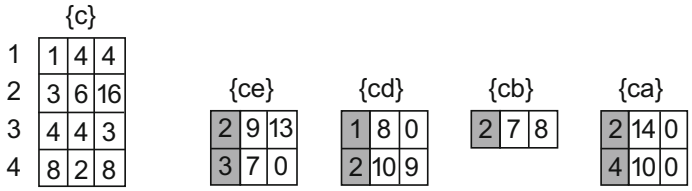



**Fig. 5** Relabeling transactions

We next provide implementation details about HUI-Miner.

In Procedure 1, for the $Ex$ element, if there is an element $Ey$ of tid equal to $Ex.tid$, HUI-Miner will generate a new element with the same tid for the utility-list of itemset $Pxy$. In the implementation of HUI-Miner, the tid of the new element is set to $i$ if $Ex$ is the $i$-th element in $Px.UL$. For example, Fig. 5 shows the utility-list of itemset {c} duplicated from Fig. 2 and the utility-lists of all 1-extensions of {c} derived from Fig. 3b. The transactions of tids 1, 3, 4, and 8, associated with the first, second, third, and fourth elements in the utility-list of {c}, are relabeled as 1, 2, 3, and 4 in the elements in the utility-lists of {c}'s 1-extensions. The purpose of relabeling transactions is to facilitate the search in line 5, because the new tids of transactions in both $Px.UL$ and $Py.UL$ directly indicate the locations of the elements associated with the transactions in $P.UL$. For example, in Fig. 5, when HUI-Miner processes element <2, 9, 13> in the utility-list of {ce} and element <2, 10, 9> in that of {cd}, the algorithm can immediately locate the second element in the utility-list of {c} according to tid 2.

**Algorithm 2** *Mine(P.UL, ULs, minutil)*

**Input**: *P.UL*, the utility-list of itemset *P*, initially empty;
*ULs*, the set of the utility-lists of all *P*'s 1-extensions;
*minutil*, a minimum utility threshold.
**Output**: all high utility itemsets with *P* as prefix.
**foreach** utility-list *Px.UL* in *ULs* **do**
  **if** SUM(*Px.UL.iutils*)≥*minutil* **then**
    output *Px*
  **end**
  **if** SUM(*Px.UL.iutils*)+SUM(*Px.UL.rutils*)≥*minutil* **then**
    *exULs* = *NULL* **foreach** utility-list *Py.UL* after *Px.UL* in *ULs* **do**
      *exULs* = *exULs*+`Construct`(*P.UL, Px.UL, Py.UL*)
    **end**
    `Mine`(*Px.UL, exULs, minutil*)
  **end**
**end**

In Procedure 2, lines 2 and 5 use the sums of the iutils and rutils in a utility-list, and the sums 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 each utility-list.

## 3.3 An Improved Method: HUI-Miner*

HUI-Miner mines high utility itemsets by recursively constructing utility-lists, in which tid comparisons are basic operations. Although tid comparisons are very simple, HUI-Miner has to perform a large number of comparisons during the mining process. In this subsection, we propose an improved method called HUI-Miner*, in which a modified utility-list structure named utility-list* is used. HUI-Miner* can construct utility-list*s without tid comparison, which leads to performance improvement because not all tid comparisons are *effective* in HUI-Miner.

### 3.3.1 Effective Comparison Ratio

In the process of constructing a utility-list, comparisons of tids are considered as effective if they result in new elements in the utility-list. For example, in Fig. 3a, there are six comparisons, two of which are effective.

Suppose that HUI-Miner performs $c$ tid comparisons to intersect two utility-lists of lengths $m$ and $n$, namely two utility-lists containing $m$ and $n$ elements, respectively. If the first tid in the longest utility-list is larger than all tids in the shortest one, $c$ reaches the minimum value $min(m, n)$ denoting the minimum value among $m$ and $n$. Except for the first comparison involving two tids, each comparison at least involves a tid that is different from the tids in the last comparison. Therefore, if all tids in the two utility-lists are used in the intersection, $c$ is the maximum number of comparisons, that is $2 + (m - 1) + (n - 1) = m + n$. The number of effective comparisons $k$ varies from 0 to $min(m, n)$.

In a mining process, suppose that HUI-Miner totally performs $u$ utility-list intersections, in which the numbers of tid comparisons and effective comparisons are $c_1, c_2, \ldots, c_u$ and $k_1, k_2, \ldots, k_u$, respectively. The effective comparison ratio denoted by *ECR* is defined as $(k_1 + k_2 + \cdots + k_u)/(c_1 + c_2 + \cdots + c_u) \times 100\%$. The ratio ranges from 0% to 100%, because $k_i$ is always smaller than or equal to $c_i$ $(1 \leq i \leq u)$. The higher ECR is, the more efficiently HUI-Miner constructs utility-lists. However, we empirically found out that ECRs are very low and do not exceed even 1% when HUI-Miner mines some sparse databases, which indicates that HUI-Miner has a low efficiency in utility-list construction.

### 3.3.2 Utility-List* Structure

To avoid ineffective tid comparisons, HUI-Miner* uses a utility-list* structure. In the utility-list*s of all 1-extensions of an itemset, the elements associated with the same transaction are linked together. In the utility-list* of itemset $X$, an element associated with transaction $T$ contains three fields: *next*, *item*, and *iutil*.

- The next field points to the next element associated with T or stores a number identifier for T.

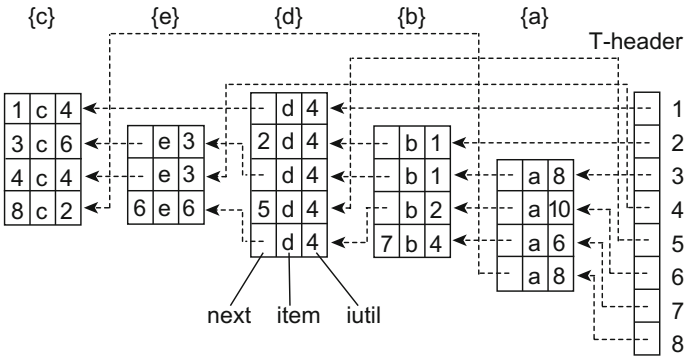



**Fig. 6** Utility-list* structure

- The item field stores the extended (i.e., last) item in X.
- The iutil field is the utility of X in T, i.e., u(X, T).

Figure 6 shows the initial utility-list*s that HUI-Miner* constructs by two scans of the sample database. During the first database scan, HUI-Miner* performs the same operations as HUI-Miner (see Sect. 3.1.1). During the second scan, HUI-Miner* processes the items in each transaction in reverse order. For example, when T4 in Table 4, namely {(c, 4), (e, 3)}, is processed, HUI-Miner* first stores (e, 3) in an element in the utility-list* of {e}; secondly, the algorithm stores (c, 4) in an element in the utility-list* of {c}, and links the next field of the previous element to the element. For a sequence of elements derived from the $i$th transaction, HUI-Miner* links the $i$-th component in a vector called *T-header* to the first element; the next of the last element is assigned as the number identifier for the transaction (or the component), namely $i$. Thus, the fourth component in the T-header in Fig. 6 points to the (e, 3) element, and 4 is assigned to the next of the (c, 4) element.

The utility-list* of an itemset doesn't need to store the information about remaining utility, because HUI-Miner* processes the items in each transaction in reverse order and thereby can accumulate the remaining utility for the itemset in the process of constructing its utility-list*. For example, consider itemset {e} which is contained in T3, T4, and T6. When item e in T3 is processed, HUI-Miner* has traversed the items after e in T3, and the sum of the utilities of these items is 13. Thus, the remaining utility of itemset {e} increases from 0 to 13. Similarly, when item e in T4 and T6 is processed, it increases by 0 and 16, respectively. Finally, the remaining utility of itemset {e} is $13 + 0 + 16 = 29$.

### 3.3.3 Fast Utility-List* Construction

HUI-Miner vertically constructs utility-lists except for initial ones, that is, only when an entire utility-list is constructed will HUI-Miner start to construct another utility-list. In contrast, HUI-Miner* constructs utility-list*s in a horizontal way.

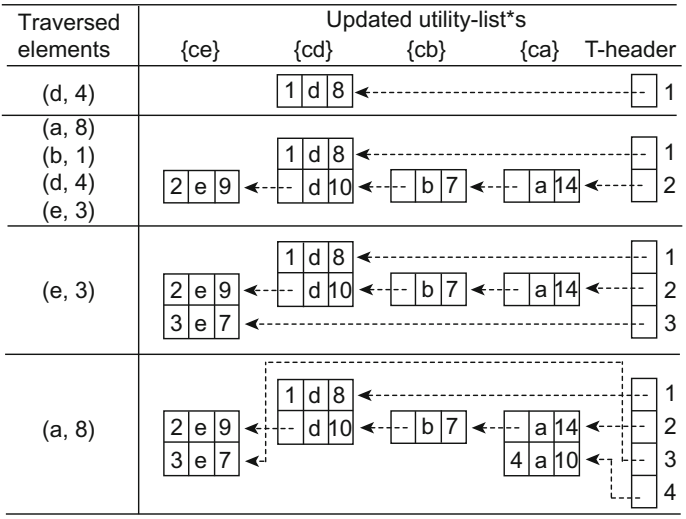



**Fig. 7** Utility-list* construction

To construct the utility-list*s of all 1-extensions of an itemset, HUI-Miner* will process each element $E$ in the utility-list* of the itemset. Firstly, suppose that the next of $E$ is a number identifier $u$. Then, HUI-Miner* can locate element $E_1$ based on the $u$th component in a related T-header. Subsequently, starting from $E_1$, HUI-Miner* traverses a sequence of elements $E_1$, $E_2$, …, $E_n$ until $E$ by following their nexts. For each $E_i$ $(1 \leq i \leq n)$, HUI-Miner* will store $(E_i.item, E.iutil + E_i.iutil)$ in a new element in the utility-list* of itemset $\{E.itemE_i.item\}$. Simultaneously, HUI-Miner* links a component in a new T-header to the first new element, links these new elements in sequence, and assigns $k$ to the next of the last new element if the component is the $k$-th one in the T-header.

Figure 7 demonstrates how HUI-Miner* constructs the utility-list*s of all {c}'s 1-extensions from the initial utility-list*s in Fig. 6. For example, from the second element in {c}'s utility-list*, HUI-Miner* traverses a sequence of elements (a, 8), (b, 1), (d, 4) and (e, 3). Thus, the algorithm stores (a, 6+8), (b, 6+1), (d, 6+4), (e, 6+3) in new elements in the utility-list*s of {ca}, {cb}, {cd}, {ce}, respectively, and simultaneously links these new elements. Because this is the second sequence, the second component in the T-header is linked to the (a, 14) element, and the next of the (e, 9) element is assigned as 2.

### 3.3.4 The Details of HUI-Miner*

The mining framework of HUI-Miner* is similar to that of HUI-Miner except that HUI-Miner* employs the utility-list* structure and performs horizontal construction. The following paragraphs provide additional details about HUI-Miner*.

In the utility-list*s of all 1-extensions of an itemset, the next of an element may be a link rather than a number identifier, if the element is not in the first utility-list*. For example, in the utility-list* of itemset {e} in Fig. 6, the next of the third element is a number identifier, while the nexts of the first two elements are links. For an element whose next is a link, starting from the element, HUI-Miner* will traverse a sequence of elements by following their links until a number identifier is obtained. After that, the number is assigned to the next of the element, which can reduce the number of traversed elements when HUI-Miner* searches for the number again in the process of processing the elements in subsequent utility-list*s.

The value in a component of a T-Header is the entrance to a transaction, and the number identifier for the component functions as a new identifier for the transaction, which facilitates element location when HUI-Miner* constructs the utility-list*s of $k$-itemsets ($k \geq 3$) as explained in Sect. 3.2.2.

Due to horizontal construction, HUI-Miner* must estimate the size of the utility-list* of each 1-extension of an itemset (or the number of elements in the utility-list*), and allocate memory for the utility-list* before constructing it. Suppose the utility-list* of itemset $Px$ contains $m$ elements and that of itemset $Py$ contains $n$ elements, and then the utility-list* of itemset $Pxy$ contains $min(m, n)$ elements at most. For example, before constructing the utility-list*s of {ce}, {cd}, {cb}, and {ca} in the above example, HUI-Miner* estimates that these utility-list*s contain 3, 4, 4, and 4 elements, respectively.

## 4 Experimental Evaluation

We have done extensive experiments on various databases to compare HUI-Miner and HUI-Miner* with state-of-the-art mining algorithms. In this section, experimental results are reported.

### *4.1 Experimental Setup*

Besides HUI-Miner and HUI-Miner*, our experiments include the following algorithms: IHUPTWU (the fastest one among the algorithms proposed in [8]), UP-Growth [9], and UP-Growth+ [10]. The main procedure of IHUPTWU has been introduced in Sect. 2.2. Based on IHUPTWU, UP-Growth incorporates four strategies to lessen the estimated utilities of itemsets and thereby reduces the number of candidate itemsets. UP-Growth+, an improved UP-Growth algorithm, can generate fewer candidates than UP-Growth for a mining task. The smaller the number of candidates is, the less the costs of generating candidates and computing their utilities. The three algorithms were shown to outperform other algorithms such as Two-Phase, ShFSM, DCG, FUM, and DCG+. Furthermore, we optimized the compared algo-

rithms by transforming a database into a view similar to that of Table 4. The view is implemented in memory, which can reduce database size and speed up utility computation.

The five algorithms were implemented in C++, using the same libraries, and were compiled using g++ (version 4.7.0). The experiments were performed on a 2.8 GHz PC machine (Intel Core i5 760) with 4 GB of memory, running a Debian (Linux 2.6.32) operating system.

Twelve databases were used in the experiments. The *BMS-POS* and *BMS-WebView-2* databases were downloaded from the KDD Cup Center [25]. The former contains several years' worth of point-of-sale data from an electronics retailer and the latter contains several months' worth of click stream data from an e-commerce web site [26]. The *chain* database was downloaded from NU-MineBench 2.0 [27], and contains transactions taken from a major grocery store chain in California. *Foodmart* was derived from Microsoft foodmart 2000 database. It contains sale data of a commercial corporation in 1997 and 1998. The other databases were downloaded from the FIMI Repository [28]. The *accidents*, *chess*, *connect*, *kosarak*, *mushroom*, and *retail* databases are real. Synthetic databases *T10I4D100K* and *T40I10D100K* were generated using the IBM Quest Synthetic Data Generation Generator. Except for *chain* and *foodmart*, the other databases do not provide the external and internal utilities of items. As in the performance evaluation of previous algorithms [8–10], 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 10. Table 5 shows statistical information about these databases, including 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).

**Table 5** Statistical information about databases

| Database | #Transactions | #Items | AvgLength | MaxLength |
|---|---|---|---|---|
| Accidents | 340,183 | 468 | 33.8 | 51 |
| BMS-POS | 515,597 | 1,657 | 6.5 | 164 |
| BMS-WebView-2 | 77,512 | 3,340 | 4.6 | 161 |
| Chain | 1,112,949 | 46,086 | 7.3 | 170 |
| Chess | 3,196 | 75 | 37 | 37 |
| Connect | 67,557 | 129 | 43 | 43 |
| Foodmart | 55,624 | 1,559 | 4.5 | 27 |
| Kosarak | 990,002 | 41,270 | 8.1 | 2,498 |
| Mushroom | 8,124 | 119 | 23 | 23 |
| Retail | 88,162 | 16,470 | 10.3 | 76 |
| T10I4D100K | 100,000 | 870 | 10.1 | 29 |
| T40I10D100K | 100,000 | 942 | 39.6 | 77 |

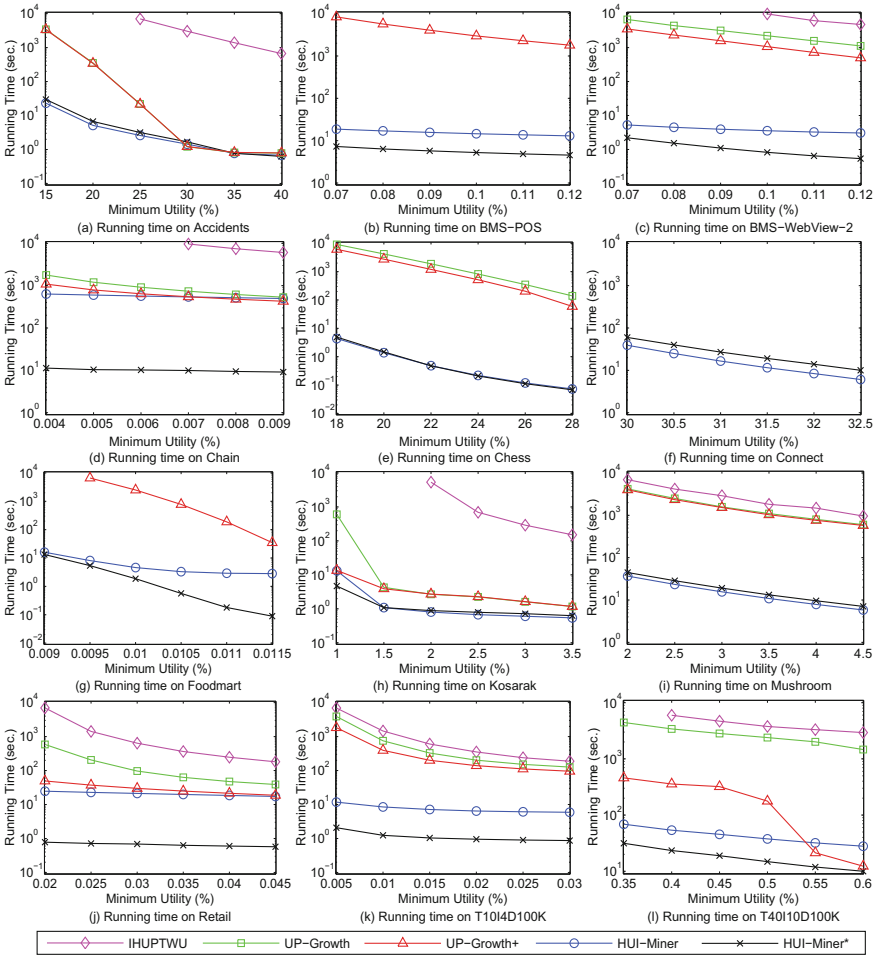



**Fig. 8** Runtime comparison

## *4.2 Running Time*

The running time of the five algorithms on all the databases is depicted in Fig. 8. Running time was recorded by the "time" command and includes input time, CPU time, and output time. For a mining task, all algorithms output the same results, which were written to "/dev/null". We terminated a mining process if its running time exceeded 10000 s.

When measuring running time, we varied the minutil threshold for each database. The lower minutil is, the more high utility itemsets are found, and running times increase. For example, for the *chain* database, when the minutil is set to 0.004% and 0.009%, the numbers of high utility itemsets are 18480 and 4578, respectively, and

the running times of HUI-Miner are 635.9 s and 497.8 s, respectively, as shown in Fig. 8d. It can be observed that HUI-Miner and HUI-Miner* perform the best for almost all mining tasks.

HUI-Miner and HUI-Miner* are two or three orders of magnitude faster than the other algorithms for the dense *accidents*, *chess* and *mushroom* databases, as shown in Fig. 8a, e and i. For example, the running times of HUI-Miner and UP-Growth+ are 36.8 and 4016.2 s for *mushroom*, when minutil is 2%. From Fig. 8f, we can see that except for HUI-Miner and HUI-Miner*, the running times of all the algorithms exceed 10000 s for the *connect* database and any minutil values. For dense databases, HUI-Miner and HUI-Miner* have similar performance.

For the sparse *BMS-POS*, *BMS-WebView-2*, *foodmart*, *T10I4D100K* and *T40I10D100K* databases, which have a relatively small number of distinct items, HUI-Miner is also two to three orders of magnitude faster than the compared algorithms while HUI-Miner* is several times faster than HUI-Miner, as shown in Fig. 8b, c, g, k and l. For sparse databases with a relatively large number of distinct items, such as *chain* and *retail*, HUI-Miner no longer has a big advantage over UP-Growth+, as shown in Fig. 8d and j. In contrast, HUI-Miner* is still several orders of magnitude faster than previous algorithms and also significantly outperforms HUI-Miner.

### *4.3 Memory Consumption*

The peak memory consumption of the five algorithms on all the databases is shown in Fig. 9. Peak memory consumption was recorded by the "massif" tool of the "valgrind" software [29].

It can be observed from the figure that the amount of memory used by HUI-Miner and HUI-Miner* does not significantly change for most tasks, as minutil is decreased, while the amount of memory used by the other algorithms increases. The reason is that these algorithms have to consume much memory to store candidate itemsets while HUI-Miner and HUI-Miner* do not generate candidates. Generally, the memory consumption of previous algorithms is proportional to the number of generated candidates. For example, for the *T10I4D100K* database, IHUPTWU generates 3826341 candidates and consumes 144.6 MB of memory while UP-Growth+ generates 1007230 candidates and consumes 68.5 MB of memory, when minutil is set to 0.005%. However, only 313509 high utility itemsets are found. HUI-Miner and HUI-Miner* neither generate nor store candidate itemsets, and they thereby consume only 28.7 MB and 23.3 MB of memory, respectively. We can also see from Fig. 9 that for most databases, HUI-Miner* consumes less memory than the other algorithms.

Another observation is that UP-Growth+ consumes more memory than UP-Growth in some cases, as shown in Fig. 9d and h, although UP-Growth+ always generates fewer candidates than UP-Growth. This is because each node of the prefix-trees used by UP-Growth+ holds more information than that in the prefix-trees used by UP-Growth [10]. For a very sparse database, the sizes of prefix-trees that UP-Growth and UP-Growth+ construct are relatively large, while the numbers of candi-

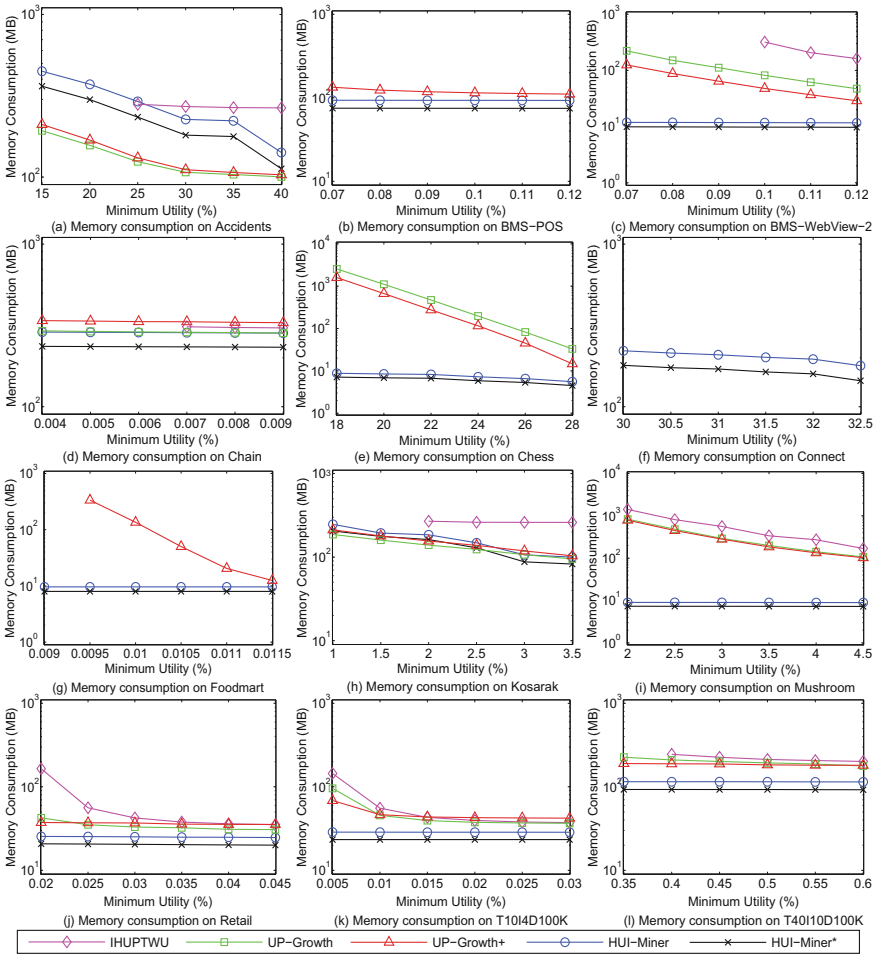



**Fig. 9** Memory consumption comparison

dates they generate are relatively small. For example, the size of the *kosarak* database is 47.55 MB, but UP-Growth and UP-Growth+ only generate 80 and 74 candidates, respectively, when minutil is set to 1.5%.

## *4.4 Orders of Processing Items*

The processing order of items significantly influences the performance of mining algorithms [8]. To evaluate the influence of the processing order on performance, we tested HUI-Miner and HUI-Miner* using the ascending TWU order (asctwu),

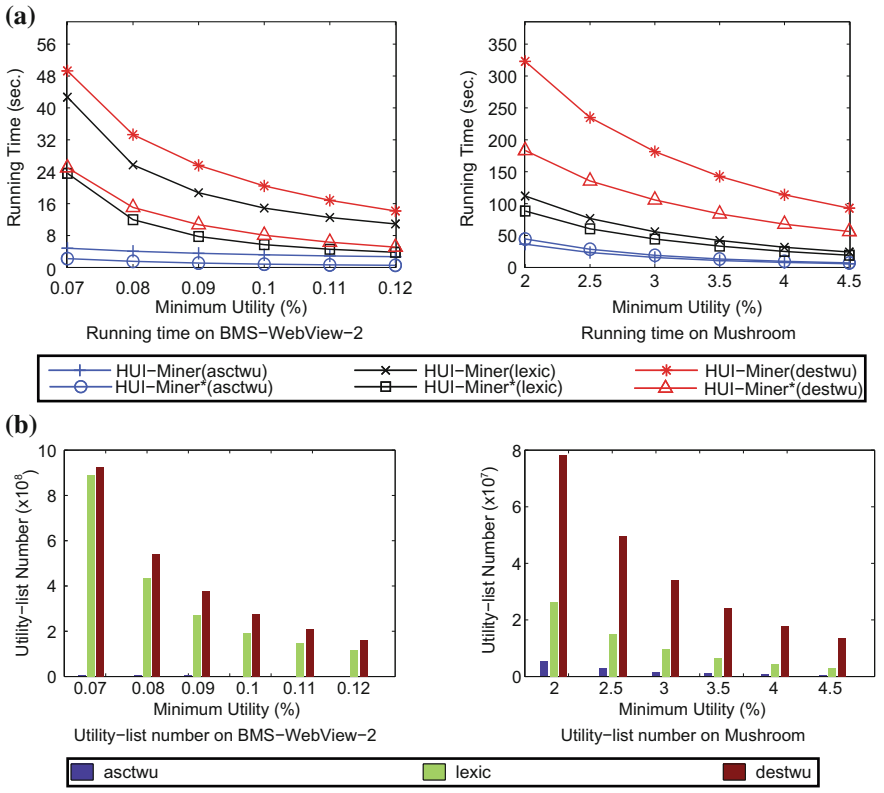
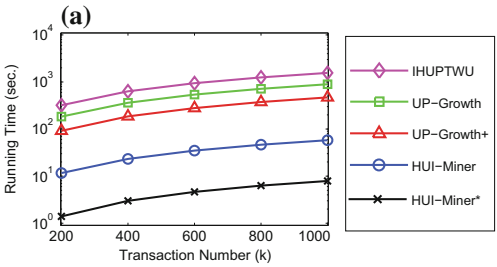



**Fig. 10** Performance comparison for different processing orders of items. **a** Running time. **b** Number of Utility-lists

lexicographical order (lexic), and descending TWU order (destwu). Figure 10a shows the running time for the *BMS-WebView-2* and *mushroom* databases.

The figure shows that the ascending TWU order leads to the best performance for the two algorithms. The reason is that such order results in a decrease in the number of constructed utility-lists. Figure 10b shows the numbers of constructed utility-lists for the above mining tasks. Using the same mining framework, the number of utility-lists constructed by HUI-Miner is the same as that of utility-list*s constructed by HUI-Miner* for a mining task, if the algorithms adopt the same processing order for items. Obviously, the number of utility lists for the ascending TWU order (too small to be clearly visible in some cases in Fig. 10b) is far smaller than that for the descending TWU or lexicographical order.

We also tested the two algorithms using the ascending and descending orders of frequency as processing order. HUI-Miner/HUI-Miner* with the ascending (or descending) frequency order almost shows the same performance as the algorithm with the ascending (or descending) TWU order, and therefore results are not shown.




(b)

| Database | IHUPTWU | UP-Growth | UP-Growth+ | #HUIs |
|---|---|---|---|---|
| 200k | 86057 | 49393 | 23885 | 6168 |
| 400k | 84818 | 48823 | 24072 | 5980 |
| 600k | 84202 | 48428 | 24052 | 6043 |
| 800k | 83716 | 48214 | 24381 | 5939 |
| 1000k | 83607 | 48205 | 24520 | 6037 |

**Fig. 11** Scalability for various database size. **a** Running time. **b** Number of candidates and high utility itemsets

The reason is that, in most cases, the TWU of an item is proportional to its frequency in the database and there is hardly any difference between the item ordering according to the ascending (or descending) frequency order and that of the ascending (or descending) TWU order.

### *4.5 Scalability*

We tested the scalability of all the algorithms by running them on databases generated by the IBM Quest Synthetic Data Generator obtained from Paolo Palmerini's website [30]. In these databases, the numbers of transactions range from 200 thousand to 1 million; the number of distinct items is 1000; the average transaction length is 10. Figure 11a compares the running times of all the algorithms for different database sizes, when minutil is set to 0.05%. All the algorithms show similar scalability.

It can be seen in Fig. 11b that the numbers of candidates generated by IHUPTWU, UP-Growth, and UP-Growth+ do not significantly change for these databases. For these algorithms, the time for computing the exact utilities of candidates increases, as the number of transactions is increased. For HUI-Miner or HUI-Miner*, the more transactions are processed, the larger the size of utility-lists or utility-list*s is, and thereby the more time is required for utility-list or utility-list* construction.

## 5 Discussions

### *5.1 Comparison with Previous Algorithms*

The experimental results show that HUI-Miner and HUI-Miner* outperform the three state-of-the-art algorithms. Table 6 gives the numbers of candidate itemsets generated by the three algorithms and the number of high utility itemsets for each

**Table 6** Number of candidates and high utility itemsets

| Accidents | 15% | 20% | 25% | 30% | 35% | 40% |
|---|---|---|---|---|---|---|
| IHUPTWU | 2,953,170 | 978,215 | 378,987 | 163,371 | 74,149 | 35,116 |
| UP-Growth | 184,255 | 18,763 | 1,215 | 34 | 1 | 0 |
| UP-Growth+ | 178,743 | 18,194 | 1,193 | 34 | 1 | 0 |
| #HUI | 280 | 0 | 0 | 0 | 0 | 0 |
| BMS-POS | 0.07% | 0.08% | 0.09% | 0.1% | 0.11% | 0.12% |
| IHUPTWU | 154,686,457 | 93,068,269 | 60,220,569 | 41,165,914 | 29,378,289 | 21,700,316 |
| UP-Growth | 6,831,360 | 4,311,723 | 2,892,396 | 2,034,779 | 1,484,988 | 1,118,242 |
| UP-Growth+ | 863,605 | 593,183 | 427,411 | 319,498 | 246,582 | 194,622 |
| #HUI | 155,312 | 106,502 | 76,429 | 56,954 | 43,917 | 34,557 |
| BMS-WebView-2 | 0.07% | 0.08% | 0.09% | 0.1% | 0.11% | 0.12% |
| IHUPTWU | — | — | 18,640,622 | 6,793,990 | 4,426,791 | 3,438,437 |
| UP-Growth | 4,546,538 | 3,038,009 | 2,202,442 | 1,581,982 | 1,149,274 | 837,163 |
| UP-Growth+ | 2,424,607 | 1,647,378 | 1,149,378 | 796,095 | 557,606 | 397,644 |
| #HUI | 638,373 | 401,901 | 253,909 | 166,883 | 116,963 | 87,820 |
| Chain | 0.004% | 0.005% | 0.006% | 0.007% | 0.008% | 0.009% |
| IHUPTWU | 43,969,001 | 9,477,024 | 738,861 | 557,703 | 429,246 | 345,320 |
| UP-Growth | 124,380 | 82,316 | 61,153 | 48,152 | 39,609 | 33,630 |
| UP-Growth+ | 72,503 | 51,486 | 40,702 | 33,942 | 29,256 | 25,850 |
| #HUI | 18,480 | 12,244 | 9,040 | 6,920 | 5,585 | 4,578 |
| Chess | 18% | 20% | 22% | 24% | 26% | 28% |
| IHUPTWU | 453,507,091 | 283,147,932 | 181,541,274 | 118,825,976 | 79,065,830 | 53,468,020 |
| UP-Growth | 50,226,810 | 22,578,752 | 9,891,124 | 4,242,056 | 1,786,382 | 702,604 |
| UP-Growth+ | 31,670,469 | 13,725,398 | 5,795,827 | 2,464,758 | 957,931 | 273,424 |
| #HUI | 34,870 | 4,872 | 230 | 0 | 0 | 0 |
| Connect | 30% | 30.5% | 31% | 31.5% | 32% | 32.5% |
| IHUPTWU | 1,356,692,999 | 1,263,731,170 | 1,179,167,256 | 1,099,517,006 | 1,026,138,358 | 958,370,972 |
| UP-Growth | 67,475,214 | 55,813,038 | 45,861,674 | 37,422,297 | 30,329,922 | 24,355,235 |
| UP-Growth+ | 67,414,717 | 55,765,677 | 45,821,301 | 37,386,177 | 30,300,180 | 24,330,378 |
| #HUI | 1,030 | 359 | 119 | 24 | 2 | 0 |
| Foodmart | 0.009% | 0.0095% | 0.01% | 0.0105% | 0.011% | 0.0115% |
| IHUPTWU | 144,690,079 | 135,836,314 | 135,827,988 | 134,248,566 | 134,243,789 | 134,240,071 |
| UP-Growth | 132,787,922 | 130,273,567 | 124,391,075 | 106,042,113 | 62,838,717 | 17,458,319 |
| UP-Growth+ | 9,291,606 | 4,113,786 | 1,540,343 | 476,180 | 117,019 | 22,700 |
| #HUI | 3,919,159 | 1,487,772 | 471,886 | 121,424 | 25,098 | 4,892 |
| Kosarak | 1% | 1.5% | 2% | 2.5% | 3% | 3.5% |
| IHUPTWU | —- | —- | 246,577 | 35,725 | 14,238 | 6,977 |
| UP-Growth | 27,748 | 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 | 29,593,410 | 17,342,264 | 11,985,060 | 7,396,748 | 5,981,220 | 3,741,960 |
| UP-Growth | 17,594,597 | 10,295,645 | 6,383,808 | 4,361,733 | 3,122,163 | 2,349,568 |

(continued)

**Table 6** (continued)

| | | | | | | |
|---|---|---|---|---|---|---|
| UP-Growth+ | 16,681,768 | 9,602,409 | 6,145,028 | 4,037,699 | 2,942,180 | 2,246,587 |
| #HUI | 3,583,596 | 1,879,322 | 1,059,350 | 640,404 | 400,136 | 256,989 |
| Retail | 0.02% | 0.025% | 0.03% | 0.035% | 0.04% | 0.045% |
| IHUPTWU | 3,280,842 | 695,677 | 308,952 | 170,145 | 111,503 | 79,649 |
| UP-Growth | 304,143 | 112,923 | 55,037 | 35,925 | 27,208 | 22,436 |
| UP-Growth+ | 27,919 | 21,047 | 17,006 | 14,279 | 12,329 | 10,781 |
| #HUI | 8,723 | 6,026 | *4,377 | 3,340 | 2,676 | 2,212 |
| T10I4D100K | 0.005% | 0.01% | 0.015% | 0.02% | 0.025% | 0.03% |
| IHUPTWU | 3,826,341 | 802,811 | 335,855 | 197,699 | 133,849 | 105,689 |
| UP-Growth | 2,155,596 | 421,548 | 188,071 | 115,330 | 85,544 | 70,147 |
| UP-Growth+ | 1,007,230 | 226,460 | 114,951 | 78,282 | 61,452 | 51,523 |
| #HUI | 313,509 | 81,582 | 51,457 | 40,898 | 34,092 | 29,176 |
| T40I10D100K | 0.35% | 0.4% | 0.45% | 0.5% | 0.55% | 0.6% |
| IHUPTWU | 4,214,063 | 2,229,140 | 1,752,510 | 1,410,603 | 1,240,640 | 1,105,825 |
| UP-Growth | 1,703,395 | 1,298,210 | 1,079,156 | 912,472 | 759,912 | 541,858 |
| UP-Growth+ | 178,123 | 141,241 | 127,304 | 71,355 | 5,341 | 2,007 |
| #HUI | 20,448 | 4,618 | 328 | 147 | 28 | 19 |

mining task. For the *BMS-WebView-2* and *kosarak* databases and the two smallest minutil values used in the experiments, IHUPTWU spends so much time ($\gg$10000 s) to generate candidates that we had to terminate its execution.

For IHUPTWU, UP-Growth and UP-Growth+, it can be observed in Figs. 8 and 9, and Table 6, that their running times and memory consumption is proportional to the number of candidates they generate. Although the algorithms can significantly reduce the number of candidates, the number is still far larger than the number of high utility itemsets in most cases. For example, IHUPTWU, UP-Growth and UP-Growth+ generate 557703, 48152 and 33942 candidates, when minutil is set to 0.007% for the *chain* database, but there are only 6920 high utility itemsets.

Compared with the previous algorithms, HUI-Miner and HUI-Miner* avoid costly candidate generation and much utility computation. For the above example, IHUPTWU, UP-Growth, and UP-Growth+ have to process 550783 (=557703−6920), 41232 (=48152−6920), and 27022 (=33942−6920) candidates, respectively. These algorithms not only generate these candidates but also compute their exact utilities on 1112949 transactions. Unfortunately, these candidates are discarded because they are not high utility. In addition, because there is no candidate itemset in HUI-Miner and HUI-Miner*, a large amount of memory is saved. For example, the size of the *mushroom* database is only 0.92 MB, but for minutil = 2% UP-Growth and UP-Growth+ consume 834.9 MB and 790.2 MB of memory to store 17594597 and 16681768 candidates, respectively. Although the algorithms can be modified to swap candidates to disk, the disk space requirement is also considerable and, moreover, the algorithms' performance will deteriorate.

**Table 7** Effective comparison ratio

| Accidents | 15% | 20% | 25% | 30% | 35% | 40% |
|---|---|---|---|---|---|---|
| ECR (%) | 72.24 | 64.31 | 55.81 | 56.84 | 58.17 | – |
| BMS-POS | 0.07% | 0.08% | 0.09% | 0.1% | 0.11% | 0.12% |
| ECR (%) | 7.488 | 6.863 | 6.396 | 6.003 | 5.668 | 5.413 |
| BMS-WebView-2 | 0.07% | 0.08% | 0.09% | 0.1% | 0.11% | 0.12% |
| ECR (%) | 5.143 | 4.318 | 3.648 | 3.128 | 2.771 | 2.485 |
| Chain | 0.004% | 0.005% | 0.006% | 0.007% | 0.008% | 0.009% |
| ECR (%) | 0.055 | 0.054 | 0.054 | 0.055 | 0.056 | 0.057 |
| Chess | 18% | 20% | 22% | 24% | 26% | 28% |
| ECR (%) | 93.67 | 90.04 | 83.69 | 76.94 | 72.58 | 67.00 |
| Connect | 30% | 30.5% | 31% | 31.5% | 32% | 32.5% |
| ECR (%) | 95.90 | 94.81 | 93.64 | 92.31 | 91.05 | 89.75 |
| Foodmart | 0.009% | 0.0095% | 0.01% | 0.0105% | 0.011% | 0.0115% |
| ECR (%) | 5.143 | 2.771 | 1.317 | 0.585 | 0.286 | 0.190 |
| Kosarak | 1% | 1.5% | 2% | 2.5% | 3% | 3.5% |
| ECR (%) | 2.589 | 5.903 | 8.837 | 14.73 | 18.24 | 25.16 |
| Mushroom | 2% | 2.5% | 3% | 3.5% | 4% | 4.5% |
| ECR (%) | 92.70 | 92.12 | 91.43 | 90.77 | 90.16 | 89.43 |
| Retail | 0.02% | 0.025% | 0.03% | 0.035% | 0.04% | 0.045% |
| ECR (%) | 0.134 | 0.136 | 0.140 | 0.145 | 0.150 | 0.157 |
| T10I4D100K | 0.005% | 0.01% | 0.015% | 0.02% | 0.025% | 0.03% |
| ECR (%) | 1.655 | 1.540 | 1.557 | 1.562 | 1.550 | 1.533 |
| T40I10D100K | 0.35% | 0.4% | 0.45% | 0.5% | 0.55% | 0.6% |
| ECR (%) | 9.252 | 8.553 | 8.111 | 7.453 | 6.695 | 5.877 |

## 5.2 *HUI-Miner Versus HUI-Miner**

Given a mining task, there is a one-to-one correspondence between the utility-lists constructed by HUI-Miner and the utility-list*s constructed by HUI-Miner*.

We can assess the performance of HUI-Miner using the ECR. Table 7 gives the ECRs of HUI-Miner for the above mining tasks (there is no tid comparison for the *accidents* database when minutil is set to 40%). For the dense *accidents*, *chess*, *connect* and *mushroom* databases, ECRs are high, which indicates that most tid comparisons performed by HUI-Miner are effective, and therefore the running time curves of HUI-Miner and HUI-Miner* are close, as shown in Fig. 8a, e, f and i. In contrast, ECRs are small for sparse databases such as *chain* and *retail*. In this case, HUI-Miner performs so many ineffective tid comparisons that it is far slower than HUI-Miner*, as shown in Fig. 8d and j. Even for the same database, the difference between the running time of HUI-Miner and that of HUI-Miner* changes if the ECR changes when minutil is set to different values. For example, as minutil is increased,

the ECR becomes smaller and smaller for the *foodmart* database, and thereby the two curves for HUI-Miner and HUI-Miner* in Fig. 8g are farther and farther apart. The opposite is observed for the *kosarak* database.

In all algorithm implementations, a transaction/item identifier or a link is represented as a 4-byte integer value, and a basic unit related to various utilities, such as a iutil, is represented as an 8-byte double value, which is necessary for exact results because performing many summations and multiplications of internal and external utilities with several decimals can result in error accumulation. The size of an element in a utility-list is 20 $(= 4 + 8 + 8)$ bytes, while that in a utility-list* is 16 $(= 4 + 4 + 8)$ bytes. Therefore, the memory consumption of HUI-Miner* should decrease by a factor of about 1/5 $(= (20 - 16)/20)$, compared with the memory consumption of HUI-Miner. The experimental results confirm this, as shown in Fig. 9. For example, for the *BMS-POS* database, HUI-Miner* consumes 75 MB of memory while HUI-Miner consumes 93.4 MB of memory when minutil is set to 0.08%.

## 5.3 The Ascending TWU Order

In IHUPTWU, UP-Growth, and UP-Growth+, items are sorted in order of descending TWU, which can increase the chance of sharing more prefix paths and thereby reduce the size of prefix-trees used by these algorithms. However, these algorithms process items in order of ascending TWU [8–10]. HUI-Miner and HUI-miner* employ list structures, the size of which is constant, no matter what order items are sorted in. In HUI-Miner and HUI-Miner*, items are sorted in order of ascending TWU and, moreover, processed in the same order.

As shown in Fig. 10, the ascending TWU order leads to decreases in the number of constructed utility-lists, that is, reduces of search space. This is illustrated by an example: for the sample database and minutil 38, the {edba} itemset is high utility and its utility is 38. Suppose that there is a perfect pruning strategy, which can guarantee that an itemset is not extended if any extension of the itemset is not high utility. Then Fig. 12 depicts the search spaces when the items are processed in ascending

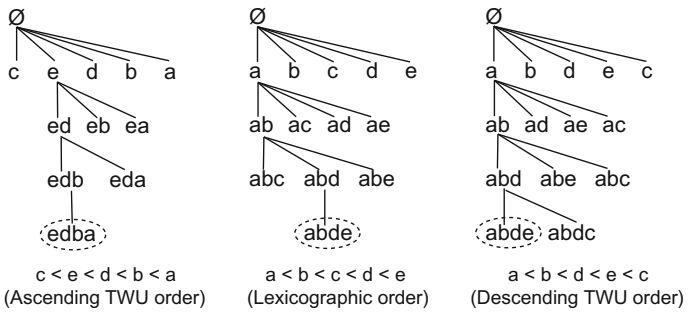



**Fig. 12** Search spaces for three item processing orders

TWU order, lexicographic order, and descending TWU order, respectively. It can be observed that an algorithm checks 11 itemsets for the first order, while the algorithm checks 13 and 14 itemsets for the last two orders, respectively.

The algorithms adopting the ascending TWU order are based on the assumption that most low utility itemsets should appear in the first subtrees of a set-enumeration tree, which can increase the probability of pruning these subtrees. The first subtrees contain more nodes than the remaining subtrees. For example, half of all nodes in a set-enumeration tree are in the first subtree. Therefore, pruning the first subtrees can result in a massive reduction of the search space. One can further consult the related work in [31, 32].

## 6 Conclusion

In this study, we have presented two efficient algorithms, HUI-Miner and HUI-Miner*, for high utility itemset mining. A novel utility-list structure was proposed to provide HUI-Miner with the required information for utility calculation and pruning. Subsequently, based on that structure, a utility-list* structure was developed and an improved algorithm named HUI-Miner* was introduced.

In the process of mining high utility itemsets, previous algorithms have to process a very large number of candidate itemsets in many cases. However, most candidates are not high utility and thereby discarded. Using utility-lists, HUI-Miner can mine high utility itemsets without candidate generation, which avoids the high cost of candidate generation and much of the cost for utility computation.

Further, we found that HUI-Miner performs many ineffective tid comparisons, especially for sparse databases. Using utility-list*s, HUI-Miner* can mine high utility itemsets without tid comparison, which speeds up the construction of its core structures.

Experimental results show that the proposed algorithms significantly outperform the state-of-the-art algorithms on various databases and that HUI-Miner* gains considerable performance improvement over HUI-Miner.

The proposed HUI-Miner algorithm was first published in the proceedings of CIKM 2012. Afterwards, several algorithms based on HUI-Miner were proposed. This includes FHM [33], HUP-Miner [34] and HMiner [35]. HUI-Miner has been also used as the basis for developing dozens of algorithms for variations of the high utility itemset mining problem. This includes algorithms for mining top-k high utility itemsets [23, 36], algorithms for mining high utility itemsets in uncertain data [37, 38], and algorithms for mining high utility itemsets with multiple thresholds [39].

**Acknowledgments** This work was supported by Natural Science Foundation of HuBei Province of China (Grant No. 2017CFB723).

## References


1. Ceglar, A., Roddick, J.F.: Association mining. ACM Comput. Surv. **38**(2), 55–86 (2006)
2. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Min. Knowl. Discov. **15**(1), 55–86 (2007)
3. Fournier-Viger, P., Lin, J.C.-W., Vo, B., Chi, T.T., Zhang, J., Le, H.B.: A survey of itemset mining. WIREs Data Min. Knowl. Discov., e1207 (2017). Wiley. https://doi.org/10.1002/widm.1207
4. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 487–499 (1994)
5. Yao, H., Hamilton, H.J., Butz, C.J.: A foundational approach to mining itemset utilities from databases. In: Proceedings of the SIAM International Conference on Data Mining (SDM), pp. 482–486 (2004)
6. Liu, Y., Liao, W.-K., Choudhary, A.N.: A two-phase algorithm for fast discovery of high utility itemsets. In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp. 689–695 (2005)
7. Li, Y.-C., Yeh, J.-S., Chang, C.-C.: Isolated items discarding strategy for discovering high utility itemsets. Data Knowl. Eng. **64**(1), 198–217 (2008)
8. Ahmed, C.F., Tanbeer, S.K., Jeong, B.-S., Lee, Y.-K.: Efficient tree structures for high utility pattern mining in incremental databases. IEEE Trans. Knowl. Data Eng. **21**(12), 1708–1721 (2009)
9. Tseng, V.S., Wu, C.-W., Shie, B.-E., Yu, P.S.: UP-Growth: an efficient algorithm for high utility itemset mining. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 253–262 (2010)
10. Tseng, V.S., Shie, B.-E., Wu, C.-W., Yu, P.S.: Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans. Knowl. Data Eng. **25**(8), 1772–1786 (2013)
11. Liu, M., Qu, J.: Mining high utility itemsets without candidate generation. In: Proceedings of the ACM 21st International Conference on Information and Knowledge Management (CIKM), pp. 55–64 (2012)
12. Rymon, R.: Search through systematic set enumeration. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, pp. 539–550 (1992)
13. Barber, B., Hamilton, H.J.: Extracting share frequent itemsets with infrequent subsets. Data Min. Knowl. Discov. **7**(2), 153–185 (2003)
14. Li, Y.-C., Yeh, J.-S., Chang, C.-C.: A fast algorithm for mining share-frequent itemsets. In: Proceedings Asia-Pacific Web Conference (APWeb), pp. 417–428 (2005)
15. Li, Y.-C., Yeh, J.-S., Chang, C.-C.: Efficient algorithms for mining share-frequent itemsets. In: Proceedings of the 11th World Congress of International Fuzzy Systems Association, pp. 534–539 (2005)
16. Li, Y.-C., Yeh, J.-S., Chang, C.-C.: Direct candidates generation: a novel algorithm for discovering complete share-frequent itemsets. In: Proceedings of the Fuzzy Systems and Knowledge Discovery, pp. 551–560 (2005)
17. Liu, Y., Liao, W.-K., Choudhary, A.: A fast high utility itemsets mining algorithm. In: Proceedings of the Utility-Based Data Mining Workshop (UBDM), pp. 90–99 (2005)
18. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach*. Data Min. Knowl. Discov. **8**(1), 53–87 (2004)
19. Hu, J., Mojsilovic, A.: High-utility pattern mining: a method for discovery of high-utility item sets. Pattern Recognit. **40**(11), 3317–3324 (2007)
20. Wu, C.W., Fournier-Viger, P., Yu, P., Tseng, V.: Efficient mining of a concise and lossless representation of high utility itemsets. In: Procedings of the IEEE International Conference Data Mining (ICDM), pp. 824–833 (2011)
21. Soulet, A., Crémilleux, B.: Adequate condensed representations of patterns. Data Min. Knowl. Discov. **17**, 94–110 (2008)

22. Soulet, A., Raïssi, C., Plantevit, M., Crémilleux, B.: Mining dominant patterns in the sky. In: Proceedings of the IEEE International Conference on Data Mining (ICDM), pp. 655–664 (2011)
23. Wu, C.W., Shie, B.-E., Tseng, V.S., Yu, P.S.: Mining top-k high utility itemsets. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 78–86 (2012)
24. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
25. KDD Cup Center (2012). http://www.sigkdd.org/kddcup/index.php?section=2000&method=data
26. Zheng, Z., Kohavi, R., Mason, L.: Real world performance of association rule algorithms. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 401–406 (2001)
27. Pisharath, J., Liu, Y., et al.: NU-Minebench: a data mining benchmark suite (2012)
28. Frequent itemset mining dataset repository (2012). http://fimi.ua.ac.be/
29. Armour Brown, C., Armour-Brown, C., et al.: Valgrind: a GPL'd system for debugging and profiling linux programs (2012). http://valgrind.org/
30. Palmerini, P.: Paolo Palmerini's Website (2012). http://miles.cnuce.cnr.it/~palmeri/datam/DCI/datasets.php
31. Liu, G., Lu, H., Lou, W., Xu, Y., Yu, J.X.: Efficient mining of frequent patterns using ascending frequency ordered prefix-tree. Data Min. Knowl. Discov. **9**(3), 249–274 (2004)
32. Liu, G., Lu, H., Yu, J.X., Wang, W., Xiao, X.: AFOPT: an efficient implementation of pattern growth approach. In: Proceedings of the IEEE International Conference on Data Mining Workshop Frequent Itemset Mining Implementations (ICDM FIMI) (2003)
33. Fournier-Viger, P., Wu, C.-W., Zida, S., Tseng, V.S.: FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proceedings of the 21st International Symposium on Methodologies for Intelligent Systems (ISMIS 2014), pp. 83–92. Springer, LNAI (2014)
34. Krishnamoorthy, S.: Pruning strategies for mining high utility itemsets. Expert Syst. Appl. **42**(5), 2371–2381 (2015)
35. Krishnamoorthy, S.: HMiner: efficiently mining high utility itemsets. Expert Syst. Appl. **90**, 168–183 (2017)
36. Duong, Q.-H., Liao, B., Fournier-Viger, P., Dam, T.-L.: An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies. Knowl. Based Syst. **104**, 106–122 (2016)
37. Lin, J.C.-W., Gan, W., Fournier-Viger, P., Hong, T.-P., Tseng, V.S.: Efficiently mining uncertain high-utility itemsets. Soft Comput. **21**, 2801–2820 (2016)
38. Lin., J.C.W., Gan, W., Fournier-Viger, P., Tseng, V.S.: Efficient algorithms for mining high-utility itemsets in uncertain databases. Knowl. Based Syst. (KBS) **96**, 171–187 (2016)
39. Krishnamoorthy, S.: Efficient mining of high utility itemsets with multiple minimum utility thresholds. Eng. Appl. AI **69**, 112–126 (2018)