**Noname manuscript No.**
(will be inserted by the editor)

# Efficient High Utility Itemset Mining using Buffered Utility-Lists

**Quang-Huy Duong**[1] · **Philippe Fournier-Viger**[2(✉)] · **Heri Ramampiaro**[1(✉)] · **Kjetil Nørvåg**[1] · **Thu-Lan Dam**[3]

Received: date / Accepted: date

**Abstract** Discovering high utility itemsets in customer transaction databases is a key task for studying the behavior of customers. It consists of finding groups of items bought together that yield a high profit. Several algorithms have been proposed to mine high utility itemsets using various approaches, with more or less complex data structures. Among existing algorithms, one-phase algorithms employing the utility-list structure have shown to be the most efficient. In recent years, the simplicity of the utility-list structure has led to the development of numerous utility-list based algorithms for various tasks related to utility mining. However, a major limitation of utility-list based algorithms is that creating and maintaining utility-lists are time consuming and can consume a huge amount of memory. The reasons are that numerous utility lists are built and that the utility-list intersection/join operation to construct utility-lists is costly. This paper addresses this issue by proposing an improved utility-list structure called utility-list buffer to reduce the memory consumption and speed up the join operation. This structure is integrated into a novel algorithm named ULB-Miner (Utility-List Buffer for high utility itemset Miner), which introduces several new ideas to more efficiently discover high utility itemsets. ULB-Miner uses the designed utility-list buffer structure to efficiently store and retrieve utility-lists, and reuse memory during the mining process. Moreover, the paper also introduces a linear time method for constructing utility-list segments in a utility-list buffer. An extensive experimental study on various datasets shows that the proposed algorithm relying on the novel utility-list buffer structure is highly efficient in terms of both execution time and memory consumption. The ULB-Miner algorithm is up to 10 times faster than the FHM and HUI-Miner algorithms and consumes up to 6 times less memory. Moreover, it performs well on both dense and sparse datasets.

**Keywords** Pattern mining · Itemset mining · Utility mining · Utility list · Utility list buffer

## 1 Introduction

In the field of data mining, the task of Frequent Itemset Mining (FIM) has been extensively studied [1, 6, 14–16, 37]. The goal of FIM is to find patterns (frequent itemsets) describing groups of products frequently purchased by customers in a transaction database. FIM has been widely applied because of its ability to discover patterns about customer behavior that are easily interpretable by humans and can support decision-making. Even though FIM has attracted a lot of attention from researchers and practitioners, a fundamental limitation of FIM is that it is designed to find frequent patterns. In real-life, however, frequent patterns are not always the most interesting or useful patterns. For example, many

✉Philippe Fournier-Viger
philfv@hitsz.edu.cn

✉Heri Ramampiaro
heri@idi.ntnu.no

[1] Department of Computer and Information Science, Norwegian University of Science and Technology, Norway
[2] School of Natural Sciences and Humanities, Harbin Institute of Technology Shenzhen Graduate School, Shenzhen, Guangdong, 518055, China
[3] Faculty of Information Technology, Hanoi University of Industry, Hanoi, Vietnam

frequent patterns, such as $\{bread, egg\}$, may be found in transaction databases, but they may generate a very low profit even though they are frequently purchased.

To find patterns that are profitable rather than solely frequent, the FIM problem has been generalized as the problem of High Utility Itemset Mining (HUIM) [5,12,22–25,27,29,33]. Key differences between HUIM and FIM are that HUIM allows non-binary purchase quantities for items in transactions, and it considers that all items may not be equally important (e.g., may have different unit profits). To perform HUIM, a user must provide a minimum utility threshold *minutil*, and the output is a set of high utility itemsets (HUIs), i.e. sets of items that yield a high profit when purchased together. HUIM has gained popularity in recent years because finding profitable patterns is more useful for businesses than finding frequent patterns, since many frequent patterns may yield a low profit, and many infrequent patterns may yield a high profit. HUIM has several applications besides market basket analysis such as cross-marketing [3,29], biomedicine [34] and click stream analysis [4,28].

Although HUIM has desirable properties, as it can provide more useful knowledge compared to FIM for several applications, it is a more difficult problem than FIM. In FIM, most algorithms are designed based on the fact that the support is anti-monotonic (the support of an itemset is greater or equal to the support of its subsets). This property of the support measure allows to efficiently reduce the search space, as supersets of infrequent itemsets do not need to be considered. However, in HUIM there is no such property for the utility measure (the utility of an itemset may be less than, equal to or greater than the utility of its subsets). For his reason, designing efficient algorithms for HUIM requires to design new methods for reducing the search space.

Utility-lists were introduced in the HUI-Miner [22] algorithm to discover high utility itemsets. HUI-Miner was shown to be up to 100 times faster than several state-of-the-art algorithms. In this approach, a utility-list is associated to each itemset, and utility-lists of itemsets are built without scanning the database by joining the utility-lists of some of their subsets. The algorithm can directly calculate the utility of itemsets and reduce the search space without having to maintain a set of candidates in memory or to repeatedly scan the database. The simplicity of the utility-list structure and the high performance of utility-list based algorithms have led to the development of numerous utility-list based algorithms for HUIM and variations of the HUIM problem such as closed high utility itemset mining [7,24,32], top-$k$ high utility itemset mining [9,20,30], high utility itemset mining in uncertain databases [21], high utility sequential pattern mining [31], and on-shelf high utility itemset mining [8,13], among others [11,12,17,35]. Although the introduction of the utility-list structure has been a breakthrough in the field of HUIM, the utility-list structure still have to be improved. In particular, it can be observed that the amount of memory required by utility-lists can be quite large.

In this study, we address this need for a more efficient structure for HUIM by proposing an improved utility-list structure called *Utility-List Buffer*, and related operations for exploiting this structure to mine HUIs. The major contributions of this work are fourfold.

- A novel Utility-List Buffer structure is proposed. It is based on the principle of buffering utility-lists to decrease memory consumption. A Utility-List Buffer consists of multiple *segments*, which are reused to store utility-list information.
- An efficient join operation is designed to create utility-lists segments in a Utility-List Buffer in linear time, to decrease the time required for utility-list construction.
- An efficient algorithm named ULB-Miner (Utility-List Buffer Miner) is proposed to mine HUIs efficiently using the designed Utility-List Buffer structure and several implementation optimizations.
- An extensive experimental study is conducted in order to evaluate the efficiency of the proposed utility-list buffer structure and ULB-Miner algorithm on both sparse and dense datasets having various characteristics. In these experiments, the performance of ULB-Miner is compared with state-of-the-art algorithms with or without the novel utility-list buffer structure. Our results show that the proposed ULB-Miner algorithm outperforms the previous state-of-the-art utility-list based HUIM algorithms. Moreover, our experiments show that algorithms employing the novel structure are up to 10 time faster than when using standard utility-lists and consumes up to 6 times less memory. Also, the proposed technique performs quite well on both dense and sparse datasets.

The rest of this paper is organized as follows. In Section 2, we briefly review the related literature. In Section 3, we define the problem of mining high utility itemsets and introduce the preliminaries of this paper. In Section 4, we present the novel utility-list buffer structure, its construction and join operations, and the ULB-Miner algorithm. In Section 5, we report on and discuss the experimental results. Finally, in Section 6, we conclude our paper and outline the future work.

## 2 Related work

A key difference between FIM (Frequent Itemset Mining) and HUIM (High Utility Itemset Mining) is that the interestingness measure used in HUIM to evaluate patterns is neither anti-monotonic nor monotonic, contrarily to the support measure used in FIM. In other words, an itemset may have supersets having a utility that is less than, equal to or greater than its utility. Because of this, the utility measure should not be directly used to reduce the search space. If an algorithm ignores all the supersets of a low utility itemset, some high utility itemsets may be missed, and the algorithm would be incomplete. The solution to this issue, adopted by most HUIM algorithms, has been to rely on upper-bounds on the utility of itemsets that are anti-monotonic to prune the search space without missing any high utility itemsets. The Transaction-Weighted Utilization (TWU) is the first such measure, which was introduced in the Two-Phase algorithm [23]. Because the TWU is anti-monotonic, it can be used to reduce the search space, while ensuring that no High Utility Items (HUIs) are missed. According to the property of the TWU measure, if an itemset has a TWU lower than the *minutil* threshold, all its supersets can be ignored. Although this property is useful for reducing the search space, a problem is that the TWU is a loose upper bound on the utility of itemsets. For this reason, many itemsets still need to be considered by algorithms relying on the TWU measure, to extract the set of HUIs. This can result in long execution times and high memory usage.

Many algorithms have been designed to discover HUIs [3, 12, 22, 23, 25–27, 29, 36]. To reduce the search space and mine HUIs efficiently, these algorithms have included various methods to overestimate the utility of itemsets. Several high utility itemset mining algorithms discover high utility itemsets using the TWU measure and a two phase approach. This includes algorithms such as Two-Phase [23], UP-Growth+ [29], PB [19] and BAHUI [25]. In the first phase, the algorithms overestimate the utility of itemsets to obtain a set of candidate HUIs using the TWU measure and other strategies. Then, in the second phase, they scan the database again to calculate the utility of these candidates and filter those that are not HUIs. Although these algorithms are complete as they can find the whole set of HUIs, the two phase approach can lead to considering and maintaining a very large number of candidate itemsets in memory. The cost of scanning the database for each itemset in the second phase to calculate their utility is also very costly. As a result, these algorithms can be slow and consume a huge amount of memory.

In recent years, to avoid the drawbacks of the two phase approach, algorithms have been proposed to mine high utility itemsets using a single phase. These algorithms can directly calculate the utility of itemsets in memory without having to repeatedly scan the database or maintain candidates in memory. Moreover, they utilize tighter upper-bounds and more efficient strategies to reduce the search space, compared to two phase algorithms. The concept of single phase algorithm was introduced in the HUI-Miner algorithm [22] by using a novel structure called utility-list. This structure stores all the information needed to calculate the utility of itemsets and reduce the search space, without repeatedly scanning the database. To discover HUIs, the HUI-Miner algorithm first constructs a utility-list for each item by scanning the database. Then, HUI-Miner recursively builds utility-lists of larger itemsets by joining the utility-lists of some of their subsets, i.e., without scanning the database again. The HUI-Miner algorithm is a complete algorithm as it can enumerate all high utility itemsets with their utility values using the utility-list structure. In terms of performance, it was shown that HUI-Miner outperforms the state-of-the-art two phase HUIM algorithms [22]. Nonetheless, the performance of HUI-Miner can still be improved. An important observation is that the join operation for obtaining the utility-lists of itemsets is costly in terms of runtime. To reduce the number of join operations performed by HUI-Miner, Fournier et al. designed the Faster High-Utility Itemset Mining (FHM) algorithm [12]. FHM applies a strategy to eliminate low utility itemsets using information about item co-occurrences. For each itemset eliminated using this strategy, the join operation does not need to be applied, thus reducing the execution time. It was shown that this pruning strategy can greatly reduce the number of join operations, and that FHM [12] can be up to six times faster than HUI-Miner.

The utility-list structure was proposed in HUI-Miner [22] to discover HUIs in a single phase, and hence avoid drawbacks of two-phase algorithms, which are to maintain a large amount of candidates in memory and to scan the database repeatedly to calculate the utilities of itemsets. HUI-Miner utilizes utility-lists to store information about the utilities of itemsets in transactions. This information allows to quickly derive the utilities of any itemset and to calculate upper-bounds on the utilities of its supersets for reducing the search space. To discover HUIs, HUI-Miner scans the database to create a utility-list for each item. Thereafter, HUI-Miner performs a depth-first search to explore the search space of all itemsets containing more than one item. During this search, the utility-list of each itemset is constructed by joining the utility-lists of some of its subsets, that is without scanning the database.

Creating the utility-lists of itemsets using the join is costly. It requires a significant amount of memory, since an algorithm has to maintain many utility-lists in memory during the search for HUIs. Moreover, in terms of execution time, the complexity of building a utility-list is also high [12]. In general, it requires to join three utility-lists of smaller itemsets. Recently, improved versions of the HUI-Miner algorithm called HUP-Miner [18] and FHM [12] have been proposed by introducing additional search space pruning strategies and optimizations. It was shown that these algorithms can be up to 6 times faster than HUI-Miner, and are the state-of-the-art algorithms for HUIM. Although some algorithms [9, 12, 21, 30, 32] have introduced strategies to reduce the number of join operations, this operation is repeatedly performed to mine high utility itemsets, and this high cost has a negative impact on the performance, especially when the number of items is huge or a database contains long transactions. Hence, joining utility-lists remains the main performance bottleneck in terms of execution time, and storing utility-lists remains the main issue in terms of memory consumption [12]. Due to the wide applications of the utility-list structure in high utility pattern mining, there is an important need to propose a more effective and efficient utility-list structure that can be constructed in linear time and can reduce memory usage.

## 3 Preliminaries and problem definition

Let there be a set of items $I = \{i_1, i_2, \ldots, i_m\}$ representing products sold in a retail store. For each item $i_j \in I$, the external utility of $i_j$ is a positive number representing its unit profit (or more generally, its relative importance to the user). A transaction database $D$ is a set of transactions denoted as $D = \{T_1, T_2, \ldots, T_n\}$, where for each transaction $T_d \in D$, the relationship $T_d \in I$ holds. For each transaction $T_d \in D$, $d$ is a unique integer that is said to be the TID (transaction identifier) of $T_d$. The internal utility of an item $i_j$ in a transaction $T_d$ is denoted as $q(i, T_d)$. It is a positive number representing the purchase quantity of the item $i_j$ in $T_d$. A set of items $X = \{i_1, i_2, \ldots, i_l\} \subseteq I$ containing $l$ items is said to be an itemset of length $l$, or alternatively, an $l$-itemset. In the rest of this paper, the notation $xy$ will be used to indicate the itemset obtained by concatenating two items $x$ and $y$. Furthermore, the notation $XY$ will be used to refer to the union of two itemsets $X$ and $Y$, i.e., $X \cup Y$.

**Table 1** A transaction database

| TID | Transaction | Transaction Utility |
|---|---|---|
| $T_1$ | (a,1), (c,1), (d,1) | 8 |
| $T_2$ | (a,2), (c,6), (e,2), (g,5) | 27 |
| $T_3$ | (a,1), (b,2), (c,1), (d,6),(e,1),(f,5) | 30 |
| $T_4$ | (b,4), (c,3), (d,3), (e,1) | 20 |
| $T_5$ | (b,2), (c,2), (e,1), (g,2) | 11 |

**Table 2** External utility values of items $\{a, b, c, d, e, f, g\}$

| Item | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| External utility | 5 | 2 | 1 | 2 | 3 | 1 | 1 |

*Example 1* Consider the transaction database depicted in Table 1, which comprises five transactions denoted as $T_1$, $T_2$, $T_3$, $T_4$, and $T_5$. This database will be used as running example. This database contains seven items denoted by the letters $a$ to $g$, that is $I = \{a, b, c, d, e, f, g\}$. Table 2 indicates the external utility of each item (e.g., unit profit). The external utilities of items $a, b, c, d, e, f$, and $g$ are 5, 2, 1, 2, 3, 1, and 1, respectively. The itemset $bc$ is a 2-itemset appearing in transactions $T_3$, $T_4$, and $T_5$. In transaction $T_4$, the items $b$, $c$, $d$, and $e$ have internal utilities (purchase quantities) of 4, 3, 3, and 1, respectively.

In high utility itemset mining, the utility measure is used to assess how important (e.g., how profitable) a pattern is.

**Definition 1 (Utility of an item in a transaction)** Let there be an item $i$ and a transaction $T_d$ such that $i \in T_d$. The utility of $i$ in $T_d$ is the product of the internal utility (purchase quantity) of item $i$ in $T_d$ by the external utility (unit profit) of $i$, that is $u(i, T_d) = q(i, T_d) \times p(i)$.

For example, in the database of Table 1, $u(a, T_1) = 1 \times 5 = 1$, and $u(c, T_1) = 1 \times 1 = 1$.

**Definition 2 (Utility of an itemset in a transaction)** For an itemset $X$ and a transaction $T_d$, the utility of $X$ in $T_d$ is a positive number defined as $u(X, T_d) = \sum_{i \in X} u(i, T_d)$.

For instance, consider the utility of itemset $ac$ in transaction $T_3$ for the database of Table 1. The utility of $ac$ in $T_3$ is calculated as $u(ac, T_3) = 1 \times 5 + 2 \times 2 = 9$. Similarly, the utility of $bc$ in transaction $T_2$ is calculated as $u(bc, T_2) = 4 \times 2 + 1 \times 3 = 11$.

**Definition 3 (Transaction utility and total utility)** The utility of a transaction $T_d$ is the sum of the utilities of items appearing in that transaction, that is $TU(T_d) = u(T_d, T_d)$. The total utility of a database $D$ is the sum of the utilities of all transactions, that is $TUD(D) = \sum_{T_d \in D} TU(T_d, T_d)$.

For example, in Table 1, $TU(T_1) = 8$, $TU(T_2) = 27$, $TU(T_3) = 30$, $TU(T_4) = 20$, $TU(T_5) = 11$. The total utility of database $D$ is $TUD(D) = (TU(T_1) + TU(T_2) + TU(T_3) + TU(T_4) + TU(T_5)) = 8 + 27 + 30 + 20 + 11 = 96$.

**Definition 4 (Utility and relative utility of an itemset)** Let there be a database $D$ and an itemset $X$. The utility of $X$ in $D$ is defined as $u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, T_d)$. The relative utility of $X$ in $D$ is defined as $ru(X) = u(X)/TUD(D)$.

For instance, the utility of the itemset $ac$ in the database of Table 1 is $u(ac) = u(ac, T_1) + u(ac, T_2) + u(ac, T_3) = 6 + 16 + 6 = 28$, while the relative utility of $ac$ in that database is $ru(ac) = 28/96 = 0.29$.

**Definition 5 (Low utility itemset and high utility itemset)** Let the minimum utility threshold (abbreviated as *minutil*) be a positive number specified by the user such that $0 < minutil < TUD(D)$. Consider an itemset $X$. It is said to be a *high utility itemset* (HUI) if its utility is no less than *minutil*. Otherwise, $X$ is said to be a *low utility itemset*.

**Definition 6 (High utility itemset mining)** Given a minimum utility threshold *minutil* and a database $D$, the problem of high utility itemset mining is to enumerate all high utility itemsets appearing in $D$.

Note that the problem of high utility itemset mining can also be defined in terms of the relative utility of itemsets. Given a relative minimum utility threshold $r\_minutil = minutil/TUD(D)$, an itemset $X$ is a high utility itemset if and only if $ru(X) \geq r\_minutil$.

In FIM, the powerful downward closure property is employed for reducing the search space. However, this property does not hold with the utility measure in HUIM. To restore this property, the TWU measure was introduced and used as an upper-bound on the utility. The TWU measure is defined as follows and has the following important properties [23].

**Definition 7 (Transaction-weighted Utilization)** Let there be an itemset $X$ and a database $D$. The Transaction-weighted Utilization(TWU) [23] of $X$ in $D$ is denoted as $TWU(X)$ and defined as $TWU(X) = \sum_{T_d \in D \wedge X \subseteq T_d} TU(T_d)$.

*Property 1 (Overestimation [23])* The utility of an itemset $X$ is less than or equal to its TWU, that is $TWU(X) \geq u(X)$.

For instance, consider the transactions $T_1, T_2$, and $T_3$ in the database of the running example. Their TWU values are 8, 27, and 30, respectively. The TWU of the item $a$ is calculated as $TWU(a) = TU(T_1) + TU(T_2) + TU(T_3) = 8 + 27 + 30 = 65$. The following property has been used by several HUIM algorithms to reduce the search space.

*Property 2 (Search space reduction using the TWU [23])* For an itemset $X$, if $TWU(X) < minutil$, it follows that $X$ and its supersets are low utility itemsets.

For example, the transaction-weighted utilization of item $f$ is $TWU(f) = TU(T_3) = 5 + 4 + 1 + 12 + 3 + 5 = 30$. Table 3 shows the transaction utilities of all transactions in $D$ and the $TWU$ values of each item.

The proposed algorithm relies on the novel utility-list buffer structure inspired by the utility-list structure [22] to mine high utility itemsets in a single phase. The next paragraphs present definition of the utility-list structure and its key properties [22].

**Table 3** The TU and TWU values of transactions for the running example

| Item Name | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| TWU | 65 | 61 | 96 | 58 | 88 | 30 | 38 |
| TID | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | | |
| TU | 8 | 27 | 30 | 20 | 11 | | |

**Definition 8 (Utility-list)** Let $\succ$ be a total order on items from $I$, and $X$ be an itemset appearing in a database $D$. The *utility-list* of $X$ is denoted as $ul(X)$. It contains a tuple $(tid, iutil, rutil)$ for each transaction $T_{tid}$ where $X$ appears ($X \subseteq T_{tid}$). The *iutil* element of a tuple for a transaction $T_{tid}$ stores the utility of $X$ in the transaction $T_{tid}$, i.e., $u(X, T_{tid})$. The *rutil* element of a tuple stores the value $\sum_{i \in T_{tid} \wedge i \succ x \forall x \in X} u(i, T_{tid})$, which is called the remaining utility of $X$ [22].

*Example 2* In the running example, the utility-list of the item $a$ is {($T_1$, 5, 3)($T_2$, 10, 17)($T_3$, 5, 25)}. The utility-list of the item $e$ is {($T_2$, 6, 5)($T_3$, 3, 5)($T_4$, 3, 0)}. The utility-list of the itemset $ae$ is {($T_2$, 16, 5),($T_3$, 8, 5)}.

Two important properties of utility-lists have been proposed to determine the utility of an itemset and to reduce the search space, respectively [22].

*Property 3 (Calculating the utility using the sum of iutil values [22])* The utility of an itemset $X$ (denoted as $u(X)$) can be calculated by performing the sum of the *iutil* values in the utility-list $ul(X)$. If that sum is less than the *minutil* threshold, $X$ is a low utility itemset. Otherwise, it is a high utility itemset [22].

*Property 4 (Pruning using an utility list's iutil and rutil values [22])* Let $X$ and $Y$ be two itemsets. It is said that $Y$ is an *extension* of $X$ if $Y$ can be obtained by adding an item $c$ to $X$, where $c \succ i$, $\forall i \in X$. The sum of the *iutil* and *rutil* values in the utility-list $ul(X)$ is an upper-bound on the utility of $Y$ and any other transitive extension of $X$. As a consequence, if this sum is less than the *minutil* threshold, it follows that any itemset that is a transitive extension of $X$ must be a low utility itemset, and thus be pruned.

## 4 The proposed utility-list buffer method

As proposed in the HUI-Miner algorithm [22], the utility-list of an itemset $Pxy$ can be constructed without accessing the database by joining the utility-lists of some subsets of $Pxy$. For instance, consider some itemsets $Px$, $Py$, and $Pxy$, where $Px$ and $Py$ are extensions of an itemset $P$ obtained by appending an item $x$ and an item $y$, respectively. To build the utility-list of the itemset $Pxy$, Algorithm 1 [22] is applied. The algorithm first considers each element in the utility-list $ul(x)$. For each such element, the algorithm verifies if there exists an element having the same transaction identifier in $ul(y)$. If such an element is found, the algorithm applies a binary search on the utility-list of the itemset $P$ to check if an element in the utility-list of $P$ has the same transaction identifier. The time complexity of this comparison of utility-lists is $\mathcal{O}(m \log nz)$, where $m$, $n$, and $z$ are the number of entries in $ul(x)$, $ul(y)$, and $ul(P)$, respectively. In terms of space complexity, a utility-list has a size proportional to $\mathcal{O}(n)$ in the worst case, where $n$ is the number of transactions. The worst case occurs when a utility-list has an entry for each transaction of the database. The overall worst-case time complexity is thus roughly $\mathcal{O}(n^3)$.

The proposed method is based on the following observations. Joining utility-lists is costly both in terms of runtime and memory consumption. In utility-list-based algorithms, memory has to be allocated to store each utility-list. Since millions of itemsets are often considered by HUI (High Utility Itemset) mining algorithms, the memory used for storing utility-lists can be quite large. Moreover, because utility-lists can contain many entries, the time requires for allocating and reusing memory for utility-lists can be quite important. In addition, a related issue is that a utility-list can be kept in memory during a long period of time by utility-list-based algorithms, even if the corresponding itemset is identified as not being a HUI and/or is not extended by the search procedure to find HUIs. This can lead to high peaks of memory usage. In conclusion, there is an important issue with how memory is managed by the state-of-the-art utility-list-based algorithms. Our experimental evaluation in Section 5 will also show this in more details.

To address this issue, this section proposes a data structure named *utility-list buffer*, designed to both quickly access information stored in utility-lists and more efficiently manage the memory used for storing the information of utility-lists. The proposed utility-list buffer structure is designed for replacing

**Algorithm 1** The traditional utility-list construction procedure

**Input:**
*ul(P)* : the utility-list of itemset $P$;
*ul(Px)*: the utility-list of itemset $Px$;
*ul(Py)*: the utility-list of itemset $Py$;
**Output:**
*ul(Pxy)*: the utility-list of itemset $Pxy$;

```
1: ul(Pxy) = NULL;
2: for each (tuple ex ∈ ul(Px)) do
3:    if (∃ey ∈ ul(Py) and ex.tid==ey.tid) then
4:       if (ul(P) is not empty) then
5:          Search element e ∈ ul(P) such 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 if
10:      ul(Pxy) ⟵ ul(Pxy) ∪ exy;
11:   end if
12: end for
13: return ul(Pxy);
```

traditional utility-lists in any utility-list-based algorithms. As it will be shown in the experimental evaluation, using the utility-list buffer structure leads to considerably lower memory usage and faster execution times for utility-list-based algorithms.

This section first introduces the utility-list buffer structure. Then, the next subsection explains how it is employed to mine high utility itemsets. In particular, an efficient ULB-Miner algorithm is presented based on the designed utility-list buffer structure.

### 4.1 The utility-list buffer structure

The utility-list buffer structure is proposed to tackle the aforementioned limitations of one-phase utility-list-based algorithms for mining high utility itemsets. The utility-list buffer structure is introduced by the following definitions and properties. Then, an example will be given to illustrate the definitions.

**Definition 9 (Utility-list buffer structure)** Let $I$ be the set of items in a database $D$. Let $Tid_D$ be the set of transaction identifiers in the database $D$. The utility-list buffer structure for the database $D$ is denoted as *UTLBuf*. The structure is designed like a memory pipeline to store information about itemsets that would be normally stored in their utility-lists. The utility-list buffer of a database stores a set of tuples of the form $(tid \in Tid_D, iutil \in \mathbb{R}, rutil \in \mathbb{R})$. These tuples called data segments, which store the tuples normally contained in traditional utility-lists. To quickly access the information stored in the utility-list buffer, a set of index segments are created, where an index segment $SUL(X)$ indicates where the information about an itemset $X$ is stored in the utility-buffer. Index segments allow fast accessibility of the data stored in the utility-buffer and are described next.

**Definition 10 (Summary of Utility-list)** The index segment of an itemset $X$ in a database $D$, also called the summary of utility-list of itemset $X$, is denoted as $SUL(X)$. It is defined as a tuple having the form ($X$, *StartPos*, *EndPos*, *SumIutil*, *SumRutil*). The *SumIutil* element stores the sum of the *iutil* values in $ul(X)$, that is $\sum ul(X).iutil$. The *SumRutil* element stores the sum of *rutil* values in the utility-list of $X$, that is $\sum ul(X).rutil$. The *StartPos* and *EndPos* elements respectively indicate the start index and end index of the data segments in the utility-list buffer structure *UTLBuf*, where the information that would be normally contained in the utility-list of $X$ is stored.

**Definition 11 (Summary List)** Let $I$ be the set of items in a database $D$. A structure called *Summary List* is further defined. It is a memory pipeline denoted as $SULs_D$, and defined as $SULs_D = \{SUL(X), X \subseteq I\}$.

The proposed utility-list buffer structure is used as follows by the proposed algorithm. When the algorithm considers an itemset $X$ from the search space as a potential HUI and as an itemset that could be extended to find other HUIs, the algorithm stores the utility-list of $X$ in the *UTLBuf* structure by temporally inserting its information in the data segments of *UTLBuf* from the *StartPos* to *EndPos* positions. Then, when needed, the algorithm accesses this information by reading the values in the *UTLBuf* from the *StartPos* to *EndPos* positions. Thanks to the utility-buffer structure, data can be

quickly accessed. For efficient memory management, the temporary memory that is allocated for an itemset $X$ in the *UTLBuf* structure is reused for storing the utility-lists of other itemsets when it is found that the utility-list of the itemset $X$ is not needed anymore by the search process. In this case, the memory is recalled and reused for other candidate itemsets (this idea will be described in more details in subsection 4.4).

In terms of implementation, we implement the proposed structures as follows. Four array lists are created, named *TIDs*, *Iutils*, *Rutils* and *SULs*. The three first lists store the information of the *UTLBuf* structure, and the fourth list is the *SULs* structure. These lists are initialized as empty and their size is increased when they are full and more space is needed. Lists are used for storing the utility-lists of itemsets, and when the utility-list of an itemset is not needed, the memory is reused to store other utility-lists. This reduces the time for allocating memory and the overall memory usage for mining HUIs.

The proposed algorithm first creates the utility-list of all single items according to the total order $\succ$ by performing a database scan. For example, consider the utility-list of the item $f$. In transaction $T_3$, we have that $u(f, T_3) = 5$ and $ru(f, T_3) = 25$. The item $f$ only appears in the transaction $T_3$. Hence, the summary of $f$ is stored in the *SULs* list and contains the following information: the item is $f$, its start position index in the lists is 0, its information ends at position index 1, the sum of its utilities is 5, and the sum of its remaining utilities is 25. The state of the utility-list buffer after inserting the item $f$ is depicted in Fig. 1. Thereafter, the other items are inserted in the same manner. The resulting state of the utility-list buffer is depicted in Fig. 2. In this figure, it can be seen that a utility-list segment is used for each item. Accessing a utility-list stored in the utility-buffer is efficient thanks to the *SULs* structure. For example, assume that the algorithm is currently processing the itemset $X = \{a\}$. To access its utility-list, the summary information of $\{a\}$ is obtained from the *SULs*. After the summary information of $\{a\}$ is obtained, its utility-list $UL(\{a\})$ is read in *UTLBuf* from the $SULs(\{a\}).StartPos$ to $SULs(\{a\}).EndPos$ positions (in red color in Fig. 2).

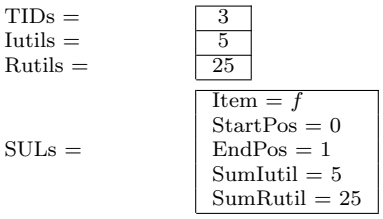



**Fig. 1** The utility-list buffer after inserting the item $f$

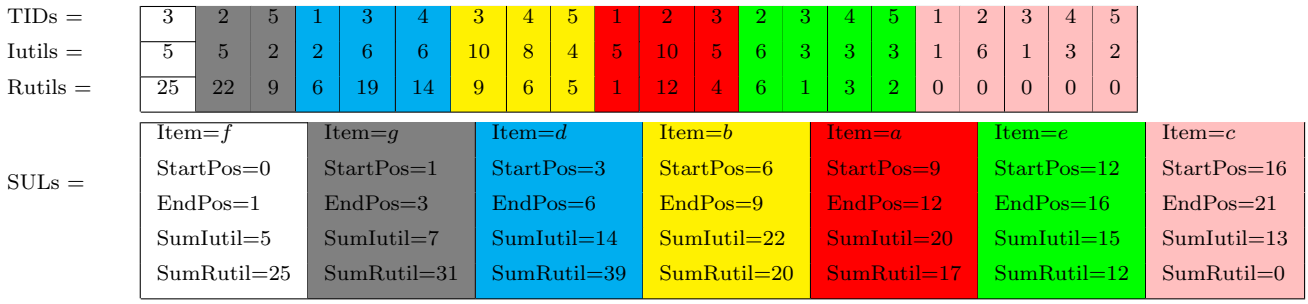



**Fig. 2** The utility-list buffer after inserting all single items

### 4.2 An efficient utility-list segment construction method

The previous subsection has explained how the proposed data structures are used to store the utility-lists of itemsets containing a single item. This subsection explains the more general case where itemsets can have two or more items.

As it has been pointed out in traditional utility-list-based algorithms [12, 22], the utility-list of a 2-itemset $xy$ can be constructed without scanning the database by joining (intersecting) the utility-lists of its items $x$ and $y$. Moreover, the utility-list of any $k$-itemset $\{i_1 \dots i_{k-1} i_k\}$ $(k \geq 3)$ can be obtained by

intersecting the utility-lists of three itemsets: $\{i_1 \dots i_{k-2} i_{k-1}\}$, $\{i_1 \dots i_{k-2}\}$ and $\{i_1 \dots i_{k-2} i_k\}$. The basic procedure for intersecting utility-lists was proposed in the HUI-Miner algorithm [22]. This procedure is given in Algorithm 1, where the utility-list of an itemset $Pxy$ is built by intersecting the utility-lists of the itemsets $Px$, $Py$, and $P$. $P$ is the prefix itemset, $x$ and $y$ are items. For each element in the utility list $ul(x)$, the procedure checks if an element has the same transaction identifier in the utility-list $ul(y)$. If yes, then a binary search is performed on the utility-list of $P$ to find an element with the same transaction identifier. Hence, the time complexity of this procedure is $\mathcal{O}(sxlog(sy))$, where $sx$ and $sy$ are respectively the number of entries in $ul(x)$ and $ul(y)$.

Although this algorithm is useful for constructing utility-lists, it cannot be directly applied to utility-lists stored in the proposed utility-buffer structure. Thus, an adapted utility-list segment construction procedure is proposed and depicted in Algorithm 2. This procedure constructs a utility-list in the next free data segments of the utility-list buffer and updates the Summary List $SULs$ structure to allow the quick retrieval of the utility-list from the buffer when needed.

**Algorithm 2** The basic utility-list segment construction procedure

**Input:**
*PPosition* : the *StartPos* of itemset $P$;
*PxPosition*: the *StartPos* of itemset $Px$;
*PyPosition*: the *StartPos* of itemset $Py$;

**Output:**
*PxyPosition*: the *StartPos* of itemset $Pxy$;

1: $PxyPosition = NULL$;
2: $countX = SULs(Px).EndPos$ - $SULs(Px).StartPos$;
3: **for** ($i = 0$; $i < countX$; $i$++) **do**
4: **if** ($\exists j \in [SULs(Py).StartPos, SULs(Py).EndPos]$ and $TIDs[SULs(Px).StartPos+i]=TIDs[j]$) **then**
5: **if** ($PPosition \geq 0$) **then**
6: Search index $p \in [SULs(P).StartPos, SULs(P).EndPos]$ such that $TIDs[p] = TIDs[SULs(Px).StartPos+i]$;
7: $TIDs[PxyPosition + p] = TIDs[PxPosition + i]$;
8: $Iutils[PxyPosition + p] = Iutils[PxPostion + i] + Iutils[PyPostion + j]$ - $Iutils[PPosition + p]$;
9: $Rutils[PxyPosition + p] = Rutils[PyPostion + j]$;
10: **else**
11: $TIDs[PxyPosition + p] = TIDs[PxPosition + i]$;
12: $Iutils[PxyPosition + p] = Iutils[PxPostion + i] + Iutils[PyPostion + j]$;
13: $Rutils[PxyPosition + p] = Rutils[PyPostion + j]$;
14: **end if**
15: **end if**
16: **end for**
17: Update $SULs$ of $Pxy$;
18: **return** $PxyPosition$;

Since transaction identifiers (Tids) in utility-lists are ordered in ascending order, an efficient way of identifying transactions that are common to two utility-lists $ul(x)$ and $ul(y)$ are to read the two utility-lists at the same time by reading the Tids sequentially in each utility-list. The complexity of this search method is $\mathcal{O}(m + n)$, which is less than $\mathcal{O}(mlogn)$ for the basic utility-list construction method. Based on this observation, we introduce an improved construction procedure named ULB-Construct, and it is presented in Algorithm 3.

### 4.3 High utility itemset miner employing the Utility-list buffer

Having presented the proposed utility-buffer structure and how utility-lists are constructed and stored in that structure, this subsection proposes a novel algorithm named ULB-Miner for discovering all high utility itemsets using that structure.

After constructing the initial utility-list buffer from an input database, the algorithm can efficiently mine all high utility itemsets by employing the utility-list buffer. The proposed approach for mining HUIs is inspired by the HUI-Miner [22] and FHM [12] algorithms, but adapted to work with the novel utility-buffer structure. In particular, it integrates the novel ULB-construct procedure, described in the previous subsection, that constructs utility-list segments in linear time. The main procedure of ULB-Miner is shown in Algorithm 4. The input is a transaction database $D$ and the *minutil* threshold, and the output is the high utility-itemsets. The main procedure performs the following steps. The algorithm

**Algorithm 3** ULB-Construct: the efficient utility-list segment construction procedure

**Input:**
*UTLBuf, SULs*;
Itemsets $P$, $Px$; $Py$;
**Output:**
Updated *UTLBuf, SULs* with itemset $Pxy$

1: Let *PPnt*, *PxPnt*, *PyPnt* are three pointers that initially point to *UTLBuf* at positions $SULs(P).StartPos$, $SULs(Px).StartPos$, and $SULs(Py).StartPos$, respectively.
2: **while** (*PxPnt* not reach $SULs(Px).EndPos$ **and** *PyPnt* not reach $SULs(Py).EndPos$) **do**
3: **if** ($TIDs[PxPnt] < TIDs[PyPnt]$) **then**
4: shift *PxPnt* to the right by 1;
5: **else if** ($TIDs[posX] > TIDs[posY]$) **then**
6: shift *PyPnt* to the right by 1;
7: **else**
8: **if** ($SULs[P]$ is not *NULL*) **then**
9: **while** (*PPnt* not reach $SULs(P).EndPos$ **and** $TIDs[PPnt] \neq TIDs[PxPnt]$) **do**
10: shift *PPnt* to the right by 1;
11: **end while**
12: **end if**
13: $UTLBuf.TIDs[TIDs.count{+}{+}] = TIDs[PxPnt]$;
14: $UTLBuf.Iutils[Iutils.count{+}{+}] = Iutils[PxPnt] + Iutils[PyPnt] - Iutils[PPnt]$;
15: $UTLBuf.Rutils[Rutils.count{+}{+}] = Rutils[PyPnt]$;
16: shift both *PxPnt* and *PyPnt* to the right by 1;
17: **end if**
18: **end while**
19: Update $SULs[Pxy]$;

first scans the database to calculate the $TWU$ of all items (line 1). Then, based on these $TWU$ values, the set $I^*$ is created, which contains all items having a $TWU$ greater than or equal to the *minutil* threshold (line 2). The $TWU$ values of items are used to build a total order $\succ$ on items, which is the ascending order of $TWU$ values (line 3). The algorithm then scans the database again (line 4) to reorder items in transactions according to that total order. At the same time, the utility-list buffer of all single items $i \in I$ and the Estimated Utility Co-occurrence Structure (EUCS) [12] are built. The EUCS stores the $TWU$ values of all pairs of items. It will be discussed in more details in the next subsection. After that the algorithm starts a recursive depth-first search by invoking the Search procedure (line 5).

**Algorithm 4** The ULB-Miner algorithm

**Input:**
$D$: a transaction database;
*minutil*: a user-defined threshold;
**Output:**
The set of high utility itemsets;

1: Scan $D$ to calculate the $TWU$ of single items;
2: Let $I^*$ be the list of single items $i$ such that $TWU(i) \geq minutil$;
3: Let $\succ$ be the total order of $TWU$ ascending values on $I^*$;
4: Scan $D$ again to build the initial *utility-list buffer* *UTLBuf*, and *SULs* of item $i \in I^*$ and build the *EUCS*;
5: **Search** ($\emptyset$, $I^*$, *minutil*, *EUCS*, *UTLBuf*, *SULs*);

The Search procedure is presented in Algorithm 5. It performs the following operations. For each extension $Px$ of $P$, if the sum of the *iutil* values of $Px$ is no less than *minutil*, then $Px$ is a high utility itemset based on Property 3. Hence, the itemset $Px$ is output (lines 2-4). Then, if the sum of the *SumIutil* and *SumRutil* values of $Px$ is greater than or equal to *minutil*, the extensions of $Px$ are considered for further exploration (line 5), based on Property 4. This process is done by combining $Px$ with each extension $Py$ of $P$ such that $y \succ x$ to produce a larger itemset itemset $Pxy$ (line 9). The utility-list segment of $Pxy$ is then constructed by calling an improved version of the ULB-Construct procedure, which will be presented in the next subsection (Algorithm 6). This procedure joins the utility-list segments of $P$, $Px$ and $Py$ (line 10). Then, a recursive call to the Search procedure with $Pxy$ is done to calculate the utility of that itemset and recursively explore its extensions (line 15).

As other utility-list based algorithms for mining high utility itemsets [12, 22], the Search procedure starts from single items and then recursively explores the search space of itemsets by appending single items, while reducing the search space using Properties 3 and 4. It thus can be easily seen that this process is correct and complete to discover all high utility itemsets.

**Algorithm 5** The Search Procedure

**Input**:
*P*: an itemset;
*ExtensionsOfP*: a set of extensions of *P*;
*minutil*: the user-specified utility threshold;
*EUCS*: the *EUCS* structure;
*utility-list buffer UTLBuf*: the utility-list buffer structure;
*SULs: The summary list*;

**Output**:
The set of high utility itemsets;

1: **for each** *itemset* $Px \in ExtensionsOfP$ **do**
2: **if** $(SULs(Px).SumIutil \geq minutil)$ **then**
3: output $Px$;
4: **end if**
5: **if** $(SULs(Px).SumIutil + SULs(Px).SumRutil \geq minutil)$ **then**
6: $ExtensionsOfPx \leftarrow \emptyset$;
7: **for each** *itemset* $Py \in ExtensionsOfP$ *such that* $y \succ x$ **do**
8: **if** $(\exists(x, y, c) \in EUCS$ such that $c \geq minutil)$ **then**
9: $Pxy \leftarrow Px \cup Py$;
10: **if** (**ULBReusingMemory-Construct**($UTLBuf$, $SULs$, $P$, $Px$, $Py$) **then**
11: $ExtensionsOfPx \leftarrow ExtensionsOfPx \cup Pxy$;
12: **end if**
13: **end if**
14: **end for**
15: **Search** ($Px$, $ExtensionsOfPx$, $minutil$, $EUCS$, $UTLBuf$, $SULs$);
16: **end if**
17: **end for**

### 4.4 Implementation optimizations

The Estimated Utility Co-Occurrence Structure (EUCS) [12] is a very useful structure for pruning the search space. The EUCS has been designed to avoid performing join operations to construct utility-lists of itemsets when some specific conditions are met. It was demonstrated that this structure and the corresponding Estimated Utility Co-occurrence Pruning (EUCP) strategy can considerably reduce the number of join operations for HUI mining using utility-lists. Hence in the proposed framework, this structure and its search space pruning strategy are also used to reduce the search space and increase the performance of the proposed algorithm. This structure is used in line 8 of Algorithm 5.

Moreover, to obtain better performance for utility-list construction, an approach is proposed in [9] for abandoning utility-list construction early named EA (Early Abandoning) strategy. This strategy and its stopping criterion are designed and employed during the construction of utility-lists of all candidate itemsets to avoid completely constructing utility-lists. The utility-list construction process is immediately stopped if a specific condition is met. This strategy can reduce the runtime and memory consumption of the algorithms considerably. Therefore, EA is also implemented in the *UTLBuf* framework. Detail of how the EA strategy is implemented in the *UTLBuf* framework is shown in Algorithm 6 using the variable *EACriterion*.

Finally, a novel optimization is proposed to reuse memory in the utility-buffer. It is based on the following observation. In utility-list based algorithms, the utility-list of an itemset containing more than one item is constructed by intersecting the utility-lists of some of its subsets. For instance, the utility-list of an itemset $Pxy$, $ul(Pxy)$, can be obtained by intersecting the utility-lists of itemsets $P$, $Px$ and $Py$. However, after constructing the utility-list of $Pxy$, it is possible that $Pxy$ is considered to not be a HUI according to Property 3, and also to not be useful for generating larger HUIs according to Property 4. Hence, the memory allocated for storing the utility-list of $Pxy$ is wasted and could be reused for storing other utility-list(s). This is a serious problem because the construction of utility-lists is a process that is repeatedly performed by the search procedure. To save memory, this paper proposes the following strategy for memory reutilization. If an itemset $Pxy$ is not a candidate for exploring the search space, then the memory allocated for storing its utility-list will be recalled and reused for the next potential candidates that will be considered by the search procedure. All the memory used for $Pxy$ will be reused and new memory is only allocated when the utility-buffer is full. The pseudo-code of the improved ULB-Construct procedure integrating this strategy is shown in Algorithm 6.

**Algorithm 6** ULBReusingMemory-Construct: Construction procedure for reusing memory

**Input:**
*UTLBuf*, *SULs*;
Itemsets *P*, *Px*; *Py*;
**Output:**
Updated *UTLBuf*, *SULs* with itemset *Pxy*

Let *PPnt*, *PxPnt*, *PyPnt* are three pointers that initially point to *UTLBuf* at positions $SULs(P).StartPos$, $SULs(Px).StartPos$, and $SULs(Py).StartPos$, respectively.
Let $EACriterion = SULs[Px].SumIutil + SULs[Py].SumIutil + SULs[Px].SumRutil + SULs[Py].SumRutil$
Let $insertionPosition = SULs.Last.endPos$;
**while** (*PxPnt* not reach $SULs(Px).EndPos$ **and** *PyPnt* not reach $SULs(Py).EndPos$) **do**
  **if** ($TIDs[PxPnt] < TIDs[PyPnt]$) **then**
    shift *PxPnt* to the right by 1;
    Substract *EACriterion* by ($Iutils[PxPnt] + Rutils[PxPnt]$)
  **else if** ($TIDs[posX] > TIDs[posY]$) **then**
    shift *PyPnt* to the right by 1;
    Substract *EACriterion* by ($Iutils[PyPnt] + Rutils[PyPnt]$)
  **else**
    **if** (*SULs*[*P*] is not *NULL*) **then**
      **while** (*PPnt* not reach $SULs(P).EndPos$ **and** $TIDs[PPnt] \neq TIDs[PxPnt]$) **do**
        shift *PPnt* to the right by 1;
      **end while**
    **end if**
    **if** ($insertionPosition \geq UTLBuf.TIDs.size()$) **then**
      $UTLBuf.TIDs[TIDs.count{+}{+}] = TIDs[PxPnt]$;
      $UTLBuf.Iutils[Iutils.count{+}{+}] = Iutils[PxPnt] + Iutils[PyPnt]$ - $Iutils[PPnt]$;
      $UTLBuf.Rutils[Rutils.count{+}{+}] = Rutils[PyPnt]$;
    **else**
      $insertionPosition{+}{+}$ //Reused Memory ;
      $UTLBuf.TIDs[insertionPosition] = TIDs[PxPnt]$;
      $UTLBuf.Iutils[insertionPosition] = Iutils[PxPnt] + Iutils[PyPnt]$ - $Iutils[PPnt]$;
      $UTLBuf.Rutils[insertionPosition] = Rutils[PyPnt]$;
    **end if**
    shift both *PxPnt* and *PyPnt* to the right by 1;
  **end if**
**end while**
**if** ($EACriterion < minutil$ ) **then**
  **return** *false*;
**end if**
Update *SULs*[*Pxy*];
**return** *true*;

### 4.5 An illustrative example

To give a better understanding of how the proposed ULB-Miner algorithm works, and at the same time show the benefits of the designed utility-list buffer structure, this subsection provides a detailed example. In this example, ULB-Miner is applied on the database $D$ shown in Table 1 with $minutil = 35$ and the external utilities of items are shown in Table 2.

Step 1. The database $D$ is scanned to calculate the $TWU$ of single items. The resulting $TWU$ values of items are shown in Table 3. The set of single items $I^*$ sorted by ascending $TWU$ values and having $TWU \geq 35$ is $\{g, d, b, a, e, c\}$. Item $f$ is dismissed because $TWU(f) = 30 < 35 = minutil$.

Step 2. The initial *UTLBuf* and *SULs* structures for items in $I^*$ are constructed. The result is shown in Fig. 3.

Step 3. The Search procedure is invoked to perform the recursive search.

(a) The procedure explores the search space starting from item $g$. Because $SULs(g).SumIutil = 7 < minutil = 35$, $g$ is not a high utility itemset. But $SULs(g).SumIutil + SULs(g).SumRutil = 7 + 31 = 38 > minutil$. Thus, extensions of $g$ should be considered as potential high utility itemsets.

(b) The algorithm appends each item $y$ to $g$ such that $y \succ g$ and $y \in I^*$ to form larger itemsets. The algorithm first considers appending $d$ to $g$ to form the larger itemset $gd$. Because $g$ and $d$ never appear together (an empty utility-list is constructed), the itemset $gd$ is not further considered.

(c) Then, the algorithm considers appending $b$ to $g$ to create the itemset $gb$. The utility-list of $bd$ is inserted into the utility-buffer *UTLBuf* as shown in Fig. 4 (cells filled with white color). The sum of the *Iutils* and *RUtils* values of $gb$ is $6 + 5 = 11 < minutil$. Hence, the itemset $gb$

is not considered as a candidate by the search procedure. Note that at this point, previous utility-list-based algorithms would allocate new memory for storing the utility-lists of the following candidates. The proposed method will instead reuse the memory allocated for the utility-list of $gb$ for storing the utility-lists of the following candidates.

(d) The algorithm next considers the itemset $ga$. The state of the utility-list buffer $UTLBuf$ after inserting the utility-list of $ga$ is shown in Fig. 5. The sum of the $Iutils$ and $RUtils$ values of $ga$ is $15 + 12 = 27 < minutil$. Thus, $ga$ will not be considered by the search procedure to generate further extensions. This memory will be reused for storing the utility-lists of the following candidates.

(e) The following item $e$ is appended to itemset $g$ to form the itemset $ge$. The algorithm inserts the utility-list of $ge$ into the utility-buffer. The resulting state of the buffer is shown in Fig. 6. The itemset $ge$ is not extended by the search procedure because the sum of the $Iutils$ and $RUtils$ values of $ge$ is $11 + 5 + 6 + 2 = 24 < minutil$. This memory will thus be reused to store the utility-lists of the following candidates.

(f) Then, the item $c$ is appended to $g$ to create the itemset $gc$. The state of the utility-list buffer after inserting the utility-list of $gc$ is shown in Fig. 7. The itemset $gc$ is also not a high utility itemset due to its low utility.

Step 4. The search for high utility itemsets is then continued with other items until no more itemsets can be generated. The result is the set of all high utility itemsets found in the dataset $D$. This set is $\{dbec : 40, dbe : 36\}$, where the number besides each itemset indicates its utility.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TIDs = | 2 | 5 | 1 | 3 | 4 | 3 | 4 | 5 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |
| Iutils = | 5 | 2 | 2 | 6 | 6 | 10 | 8 | 4 | 5 | 10 | 5 | 6 | 3 | 3 | 3 | 1 | 6 | 1 | 3 | 2 |
| Rutils = | 22 | 9 | 6 | 19 | 14 | 9 | 6 | 5 | 1 | 12 | 4 | 6 | 1 | 3 | 2 | 0 | 0 | 0 | 0 | 0 |

| SULs = | | | | | | |
|---|---|---|---|---|---|---|
| | Item=$g$ | Item=$d$ | Item=$b$ | Item=$a$ | Item=$e$ | Item=$c$ |
| | StartPos=0 | StartPos=2 | StartPos=5 | StartPos=8 | StartPos=11 | StartPos=15 |
| | EndPos=2 | EndPos=5 | EndPos=8 | EndPos=11 | EndPos=15 | EndPos=20 |
| | SumIutil=7 | SumIutil=14 | SumIutil=22 | SumIutil=20 | SumIutil=15 | SumIutil=13 |
| | SumRutil=31 | SumRutil=39 | SumRutil=20 | SumRutil=17 | SumRutil=12 | SumRutil=0 |

**Fig. 3** The initial utility-list buffer

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TIDs = | 2 | 5 | 1 | 3 | 4 | 3 | 4 | 5 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 5 |
| Iutils = | 5 | 2 | 2 | 6 | 6 | 10 | 8 | 4 | 5 | 10 | 5 | 6 | 3 | 3 | 3 | 1 | 6 | 1 | 3 | 2 | 6 |
| Rutils = | 22 | 9 | 6 | 19 | 14 | 9 | 6 | 5 | 1 | 12 | 4 | 6 | 1 | 3 | 2 | 0 | 0 | 0 | 0 | 0 | 5 |

**Fig. 4** The utility-list buffer after inserting the utility-list of $gb$

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TIDs = | 2 | 5 | 1 | 3 | 4 | 3 | 4 | 5 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 2 |
| Iutils = | 5 | 2 | 2 | 6 | 6 | 10 | 8 | 4 | 5 | 10 | 5 | 6 | 3 | 3 | 3 | 1 | 6 | 1 | 3 | 2 | 15 |
| Rutils = | 22 | 9 | 6 | 19 | 14 | 9 | 6 | 5 | 1 | 12 | 4 | 6 | 1 | 3 | 2 | 0 | 0 | 0 | 0 | 0 | 12 |

**Fig. 5** The utility-list buffer after inserting the utility-list of $ga$

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TIDs = | 2 | 5 | 1 | 3 | 4 | 3 | 4 | 5 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 2 | 5 |
| Iutils = | 5 | 2 | 2 | 6 | 6 | 10 | 8 | 4 | 5 | 10 | 5 | 6 | 3 | 3 | 3 | 1 | 6 | 1 | 3 | 2 | 11 | 5 |
| Rutils = | 22 | 9 | 6 | 19 | 14 | 9 | 6 | 5 | 1 | 12 | 4 | 6 | 1 | 3 | 2 | 0 | 0 | 0 | 0 | 0 | 6 | 2 |

**Fig. 6** The utility-list buffer after inserting the utility-list of $ge$

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TIDs = | 2 | 5 | 1 | 3 | 4 | 3 | 4 | 5 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 2 | 5 |
| Iutils = | 5 | 2 | 2 | 6 | 6 | 10 | 8 | 4 | 5 | 10 | 5 | 6 | 3 | 3 | 3 | 1 | 6 | 1 | 3 | 2 | 11 | 0 |
| Rutils = | 22 | 9 | 6 | 19 | 14 | 9 | 6 | 5 | 1 | 12 | 4 | 6 | 1 | 3 | 2 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |

**Fig. 7** The utility-list buffer after inserting the utility-list of *gc*

In the above example, the proposed algorithm relying on the novel utility-list buffer allocates only 2 entries in the utility-buffer for storing the utility-lists of extensions of the item $g$. Previous utility-list-based algorithms such as HUI-Miner and FHM would utilize 6 entries to store the utility-lists, due to the lack of a mechanism for reusing memory. If we consider the full search space for the previous example, the proposed algorithm only needs 33 entries in the utility-buffer and reuses 39 times some existing entries to store utility-lists. This simple example shows that the proposed utility-list buffer structure is useful for mining high utility itemsets while reusing memory.

## 5 Performance study

We performed a series of large scale experiments to evaluate the performance of the proposed ULB-Miner algorithm employing the designed utility-list buffer structure. The algorithms were implemented by extending the SPMF open-source Java data mining library [10]. The source code was compiled using the J2SDK 1.7.0, and the memory measurements were done using the standard Java API. The experiments were run on a computer equipped with an Intel core i3 processor 2.4 GHz and 4 GB of RAM, running the Windows 7 operating system.

**Table 4** Characteristics of the datasets

| Dataset | #Transactions | #Distinct items | Avg. trans. length |
|---|---|---|---|
| Connect | 67,557 | 129 | 43 |
| Chainstore | 1,112,949 | 46,086 | 7.2 |
| Chess | 3196 | 75 | 37 |
| Foodmart | 4141 | 1559 | 4.4 |
| Kosarak | 990,000 | 41,270 | 8.1 |
| Retail | 88,162 | 16,470 | 10.3 |

### 5.1 Experimental setup

Both real and synthetic datasets having varied characteristics were used in the experiments. These datasets are standard benchmark datasets used to evaluate HUIM algorithms. The characteristics of these datasets are described in Table 4, where #Transactions, #Distinct items and Avg. trans. length indicate the number of transactions, the number of distinct items and the average transaction length, respectively. These datasets were selected because they are standard benchmark datasets and they have varied characteristics.

We used two real-world customer transaction datasets named Chainstore[1] and Foodmart[2]. Chainstore is a very large dataset consisting of transactions from a Californian retail store, while Foodmart is a small dataset of customer transactions obtained from the Microsoft Food-Mart 2000 database. Retail[3] is a sparse dataset containing customer transactions from a Belgian retail store. Kosarak[4] is a very sparse dataset with moderately short transactions. Lastly, two dense datasets named Chess[5] and Connect[5] were used. Although these two datasets are not retail data, they are often used in the pattern mining literature as benchmark datasets to evaluate the performance on dense data. Chess is especially a quite challenging dataset for most mining algorithms because it contains many long itemsets. The Chainstore and Foodmart datasets already contain real unit profits and purchase quantities. For other datasets,

[1] http://cucis.ece.northwestern.edu/projects/DMS/MineBenchDownload.html
[2] https://www.microsoft.com/en-us/download/details.aspx?id=51958
[3] http://fimi.cs.helsinki.fi/data/
[4] http://fimi.cs.helsinki.fi/data/
[5] http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php

external utilities of items are generated between 1 and 1000 by using a log-normal distribution and quantities of items are generated randomly between 1 and 5, as the settings of previous studies [12,22].

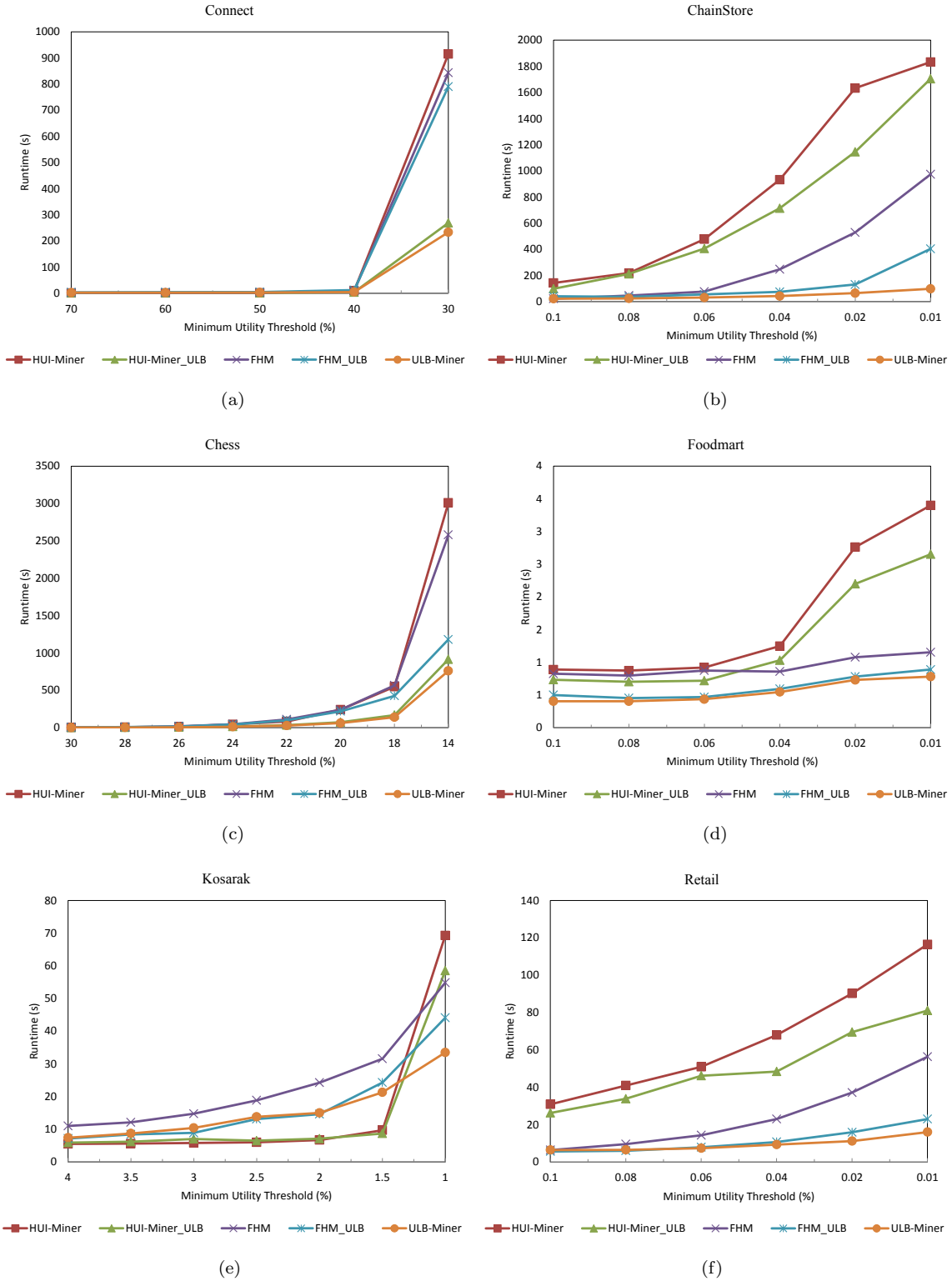



**Fig. 8** Runtime comparison on different datasets

### 5.2 Running time

The performance of ULB-Miner is compared with two state-of-the-art HUI mining algorithms, namely HUI-Miner and FHM. These algorithms were chosen since they are state-of-the art HUIM algorithms. These algorithms are also based on the traditional utility-list structure. Moreover, we also prepared two improved versions of HUI-Miner and FHM, named HUI-Miner_ULB and FHM_ULB, respectively. These versions employ the proposed utility-list buffer structure and the basic utility-list segment construction procedure (Algorithm 2).

We rant the compared algorithms on each dataset while decreasing the *minutil* threshold until the algorithms became too long to execute, ran out of memory or a clear winner was observed. For each dataset, we recorded the execution time and memory consumption. The comparison of execution times is shown in Fig 8. As presented in these figures, the HUI-Miner_ULB and FHM_ULB versions are faster than the original implementations of these algorithms on all datasets. Especially, when *minutil* is decreased, there is a big gap between the runtimes of the original and improved versions. The proposed ULB-Miner algorithm is faster than the compared algorithms when *minutil* is small on the Kosarak dataset. For the remaining datasets, the proposed algorithm is the fastest for all *minutil* values. The compared algorithms are one-phase algorithms employing the traditional utility-list structure, which perform the costly utility-list intersection operation. To reduce this cost, ULB-Miner employs the designed efficient utility-list segment construction method to quickly search for transactions that are common to two utility-list segments. This considerably reduces its execution time.

### 5.3 Memory consumption

Table 5 compares the peak memory usage of the algorithms on the six datasets when the *minutil* threshold is set to the smallest values used in the previous experiment. All memory measurements were done using the standard Java API. By observing these results, it is found that the proposed utility-list buffer structure reduces the memory consumption of the HUI-Miner and FHM algorithms on all datasets. The FHM_ULB and ULB-Miner algorithms consume almost the same amount of memory on the experimental datasets because they employ similar strategies. Both FHM_ULB and ULB-Miner consume less memory than FHM. The best results are obtained on the Chess and Connect datasets. There, the memory consumption is reduced by up to 4 and 6 times, respectively. This can be explained as follows. These datasets are dense with long transaction and many items. As a result, the algorithms generate a huge amount of candidates. But the proposed ULB-Miner algorithm reuses most of the memory for storing utility-lists thanks to its utility-buffer structure, and it thus have a low memory consumption. Similar results are also obtained when comparing the HUI-Miner and HUI-Miner_ULB algorithms. HUI-Miner_ULB consumes less memory than HUI-Miner on all datasets. The best result is obtained on the Chess dataset. Here, the gap in terms of memory usage is clear and large. The gap shrinks a bit due to the EUCS structure. But it is an acceptable trade-off when considering the runtime performance. On overall, the results depicted in Table 5 show that the proposed utility-list buffer structure is efficient in terms of memory consumption. In some cases, the proposed method can reduce memory consumption by up to 6 times.

**Table 5** Comparison of peak memory usage (MB)

| Dataset | HUI-Miner | HUI-Miner_ULB | FHM | FHM_ULB | ULB-Miner |
|---|---|---|---|---|---|
| Connect | 452.6 | 399.9 | 1516.9 | 398.1 | **368.6** |
| Chainstore | 1367.3 | **1021.1** | 2792.7 | 2396.9 | 2402.7 |
| Chess | 752.4 | **140.1** | 1319.7 | 209.7 | 208.3 |
| Foodmart | 423.4 | 257.2 | 68.3 | **40.1** | 41.3 |
| Kosarak | 1060.2 | **910.1** | 1270 | 1030 | 1015.7 |
| Retail | 803.53 | **442.1** | 670.22 | 544.7 | 544.6 |

### 5.4 Comparison on the number of utility-lists

To analyze in more details the memory consumption of the proposed algorithm, we performed an experiment to compare the number of utility-lists created by allocating new memory when using the

designed utility-list buffer structure and when not using that structure. For this experiment, a version of the proposed ULB-Miner that employ the traditional utility-list structure [22] was prepared (i.e., that does not use the novel utility-list buffer structure). Then, the number of utility-lists generated by allocating new memory was measured for both versions of the algorithm on each dataset. Fig 9 shows the comparison.

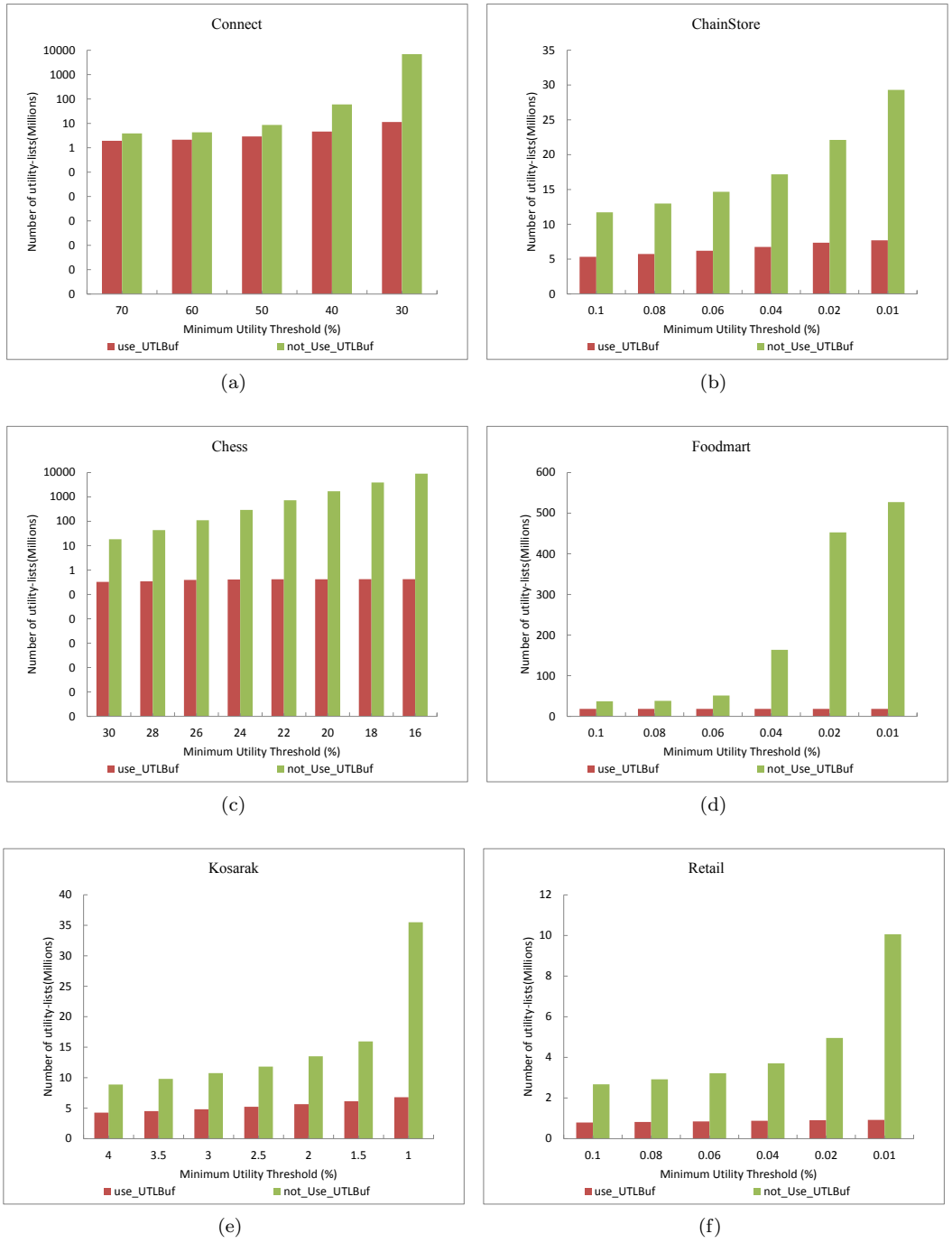


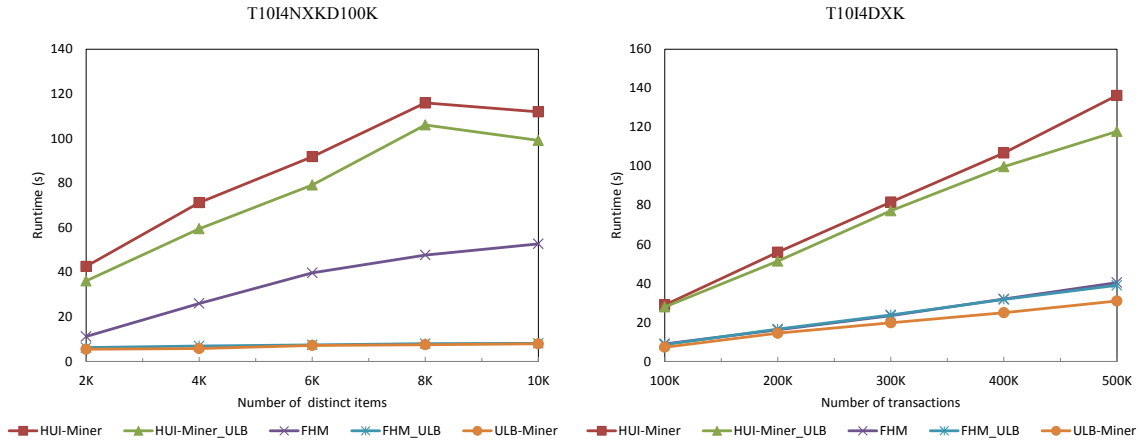

**Fig. 9** Comparison of the number of utility-lists created by allocating new memory when using or not using the utility-list buffer structure

As presented in this figure, employing the designed utility-list buffer structure can greatly reduce the number of utility-lists created by allocating new memory during the mining process, especially for dense and long transaction datasets such as Chess. When the *minsup* threshold is set to small values,

the difference in terms of number of generated utility-lists becomes clear and large. The reason is that for these datasets, there are many extensions for each considered itemsets. Hence, the number of utility-lists generated during the process of itemset extension is huge if the traditional utility-list structure is used. Fortunately, using the proposed utility-list buffer reduces the need to allocate new memory for utility-lists during the search by reusing the memory used for storing previously generated utility-lists.

### 5.5 Scalability evaluation

Lastly, we performed experiments to evaluate the scalability of the proposed algorithm on a synthetic dataset named T10I4NXKDYK, where the number of transactions Y and the number of items X were varied. The dataset was generated using the IBM Quest synthetic data generator [2], where the numbers after T, I, N, and D represent the average transaction size, average size of maximal potentially frequent patterns, number of items, and the number of transactions, respectively. For this experiment, the *minutil* threshold was set to 0.05%, the number of items was varied from 2K to 10K, and the number of transactions was varied from 100K to 500K. Results are shown in Fig. 10(a) and Fig. 10(b), respectively. As can be observed from these figures, the proposed algorithm has almost constant scalability when the number of items increases, and it has linear scalability when the number of transactions increases.




(a) Varied number of items (b) Varied number of transactions

**Fig. 10** Scalability of the compared algorithms for different parameter values

## 6 Conclusion

In recent years, utility-list-based algorithms for discovering high utility itemsets have become widely used because of their efficiency and simplicity to implement. However, it can be observed that the amount of memory required by utility-lists can be quite large. To address this issue, this paper has presented a novel structure named utility-list buffer for reusing the memory for storing utility-lists. We have proposed an algorithm for high utility itemset mining named ULB-Miner. This algorithm integrates the utility-list buffer structure with an efficient method for constructing utility-list segments to reduce the time and the memory usage required for mining high utility itemsets.

We have performed an extensive experimental study on six real-life datasets to compare the performance of ULB-Miner with the state-of-the-art algorithms HUI-Miner and FHM, which both employ traditional utility-lists. Our results show that the proposed utility-list buffer structure and its construction method increase the effectiveness of HUI mining both in terms of execution time and memory consumption. The peak memory usage was reduced by up to six times, and execution times was reduced by up to 10 times. In addition to this, the important contribution of this work is that the proposed utility-list buffer structure can be adapted to other utility-list-based algorithms for variations of the HUI mining problem, including closed high utility itemset mining, top-$k$ high utility itemset mining,

high utility itemset mining in uncertain databases, high utility sequential pattern mining, and on-shelf high utility itemset mining.

Since data stream has become widespread in many fields such as sensor network monitoring, trade management, and medical data analysis, methods for mining patterns in data stream have attracted a lot of attention in recent years. In future work, we plan to adapt the proposed utility-list buffer structure for streams, and investigate other optimization approaches involving itemset mining for mining patterns in data streams.

## Acknowledgements

This research was partly supported by the Youth 1000 funding of Prof. Philippe Fournier-Viger and partly funded by the NTNU through the MUSED project.

## References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. VLDB pp. 487–499 (1994)
2. Agrawal, R., Srikant, R.: Quest Synthetic Data Generator. Available at. (<http://www.almaden.ibm.com/cs/quest/syndata.html>) (1994)
3. Ahmed, C., Tanbeer, S., Jeong, B.S., Lee, Y.K.: Efficient tree structures for high utility pattern mining in incremental databases. IEEE Transactions on Knowledge and Data Engineering **21**(12), 1708–1721 (2009)
4. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S., Lee, Y.K.: Efficient mining of utility-based web path traversal patterns. In: Proceedings of the 11th International Conference on Advanced Communication Technology - Volume 3, ICACT'09, pp. 2215–2218 (2009)
5. Chan, R., Yang, Q., Shen, Y.D.: Mining high utility itemsets. In: Third IEEE International Conference on Data Mining (ICDM 2003), pp. 19–26 (2003)
6. Dam, T.L., Li, K., Fournier-Viger, P., Duong, Q.H.: An efficient algorithm for mining top-rank-k frequent patterns. Applied Intelligence **45**(1), 96–111 (2016)
7. Dam, T.L., Li, K., Fournier-Viger, P., Duong, Q.H.: CLS-Miner: efficient and effective closed high utility itemset mining. Frontiers of Computer Science pp. 1–27 (2017)
8. Dam, T.L., Li, K., Fournier-Viger, P., Duong, Q.H.: An efficient algorithm for mining top-k on-shelf high utility itemsets. Knowledge and Information Systems pp. 1–35 (2017)
9. 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. Knowledge-Based Systems **104**, 106–122 (2016)
10. Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.W., Tseng, V.S.: SPMF: A java open-source pattern mining library. Journal of Machine Learning Research **15**, 3569–3573 (2014)
11. Fournier-Viger, P., Lin, J.C.W., Duong, Q.H., Dam, T.L.: PHM: Mining Periodic High-Utility Itemsets. In: Lecture Notes in Computer Science, ICDM 2016, pp. 64–79. Springer (2016)
12. Fournier-Viger, P., Wu, C.W., Zida, S., Tseng, V.: FHM: Faster High-Utility Itemset Mining Using Estimated Utility Co-occurrence Pruning. In: Foundations of Intelligent Systems, *Lecture Notes in Computer Science*, vol. 8502, pp. 83–92. Springer International Publishing (2014)
13. Fournier-Viger, P., Zida, S.: FOSHU: Faster On-shelf High Utility Itemset Mining – with or Without Negative Unit Profit. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, pp. 857–864 (2015)
14. Grahne, G., Zhu, J.: Fast algorithms for frequent itemset mining using fp-trees. IEEE Transactions on Knowledge and Data Engineering **17**(10), 1347–1362 (2005)
15. Han, J., Wang, J., Lu, Y., Tzvetkov, P.: Mining top-k frequent closed patterns without minimum support. In: Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, pp. 211–218 (2002)
16. Han, J.W., Pei, J., Yin, Y.W.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery **8**(1), 53–87 (2004)
17. Joshi, M., Bhalodia, D.: Mining High Utility Itemset Using Graphics Processor, pp. 665–674. Springer International Publishing, Intelligent Systems Technologies and Applications ISTA, 2016 (2016)
18. Krishnamoorthy, S.: Pruning strategies for mining high utility itemsets. Expert Systems with Applications **42**(5), 2371 – 2381 (2015)
19. Lan, G.C., Hong, T.P., Tseng, V.S.: An efficient projection-based indexing approach for mining high utility itemsets. Knowledge and Information Systems **38**(1), 85–107 (2014)
20. Lee, S., Park, J.S.: Top-k high utility itemset mining based on utility-list structures. In: 2016 International Conference on Big Data and Smart Computing (BigComp), pp. 101–108 (2016)
21. Lin, J.C.W., Gan, W., Fournier-Viger, P., Hong, T.P., Tseng, V.S.: Efficient algorithms for mining high-utility itemsets in uncertain databases. Knowledge-Based Systems **96**, 171 – 187 (2016)
22. Liu, M., Qu, J.: Mining high utility itemsets without candidate generation. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12, pp. 55–64 (2012)
23. Liu, Y., Liao, W.k., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Advances in Knowledge Discovery and Data Mining, pp. 689–695. Springer Berlin Heidelberg (2005)
24. Sahoo, J., Das, A.K., Goswami, A.: An efficient fast algorithm for discovering closed+ high utility itemsets. Applied Intelligence pp. 1–31 (2016)
25. Song, W., Liu, Y., Li, J.: BAHUI: Fast and Memory Efficient Mining of High Utility Itemsets Based on Bitmap. International Journal of Data Warehousing and Mining **10**(1), 1–15 (2014)
26. Song, W., Liu, Y., Li, J.: Mining high utility itemsets by dynamically pruning the tree structure. Applied Intelligence **40**(1), 29–43 (2014)

27. Song, W., Zhang, Z., Li, J.: A high utility itemset mining algorithm based on subsume index. Knowledge and Information Systems **49**(1), 315–340 (2016)
28. Thilagu, M., Nadarajan, R.: Efficiently mining of effective web traversal patterns with average utility. Procedia Technology **6**, 444 – 451 (2012)
29. Tseng, V., Shie, B.E., Wu, C.W., Yu, P.: Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Transactions on Knowledge and Data Engineering **25**(8), 1772–1786 (2013)
30. Tseng, V., Wu, C.W., Fournier-Viger, P., Yu, P.: Efficient algorithms for mining top-k high utility itemsets. IEEE Transactions on Knowledge and Data Engineering **28**(1), 54–67 (2016)
31. Wang, J.Z., Huang, J.L., Chen, Y.C.: On efficiently mining high utility sequential patterns. Knowledge and Information Systems **49**(2), 597–627 (2016)
32. Wu, C.W., Fournier-Viger, P., Gu, J.Y., Tseng, V.S.: Mining closed+ high utility itemsets without candidate generation. In: 2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 187–194 (2015)
33. Wu, C.W., Shie, B.E., Tseng, V.S., Yu, P.S.: Mining top-k high utility itemsets. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, pp. 78–86 (2012)
34. Y-C, L., C-P, C., VS, T.: Mining differential top-k co-expression patterns from time course comparative gene expression datasets. BMC Bioinformatics **14**(230) (2013)
35. Yun, U., Ryang, H., Lee, G., Fujita, H.: An efficient algorithm for mining high utility patterns from incremental databases with one database scan. Knowledge-Based Systems pp. 1–19 (2017)
36. Yun, U., Ryang, H., Ryu, K.H.: High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates. Expert Systems with Applications **41**(8), 3861 – 3878 (2014)
37. Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 326–335 (2003)