Efficient high utility itemset mining without the join operation

Yihe Yan[a], Xinzheng Niu[a,*], Zhiheng Zhang[b], Philippe Fournier-Viger[c], Libin Ye[d,*], Fan Min[e]

[a]*School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China*
[b]*School of Information and Engineering, Sichuan Tourism University, Chengdu 610100, China*
[c]*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China*
[d]*College of Computer Science, Sichuan University, Chengdu 610065, China*
[e]*School of Computer Science and Software Engineering, Southwest Petroleum University, Chengdu 610500, China*

Abstract

The task of mining high-utility itemsets in a database given a minimum threshold is attracting more and more interest due to its many applications. Existing algorithms such as the vertical ones have the advantages of high scalability, efficiency and extensibility. However, they depend on a costly join operation to generate new itemsets. To overcome this limitation, this paper proposes a novel vertical algorithm, FOTH (Fast sOrted iTemset searcH), which employs a novel effective data structure, the IndexSet. The IndexSet self-propagates to produce sub-IndexSets, eliminating the need to perform join operations, which considerably reduces the memory and computation requirements. Experiments were conducted on eight benchmark databases to compare the performance of FOTH with four state-of-the-art list-based algorithms. The results show that FOTH outperforms the other algorithms on dense databases.

Keywords:
High utility itemset, Utility-list, Pattern discovering, Propagation.

1. Introduction

Mining interesting patterns is an essential task in the data mining research field. The FIM (Frequent Itemset Mining) problem was first introduced by Agrawal et al [1] in 1993 to discover the frequent itemsets (sets of values) that occur in a database. A traditional application of FIM, among many others, is to identify the sets of items that are often purchased (e.g., {egg, milk, bread}) by customers. Nonetheless, FIM [2] does not capture the utility of itemsets, that is their relative importance in terms of aspects such as profit. Hence, Yao et al. [3] formulated the problem of High Utility Itemset Mining (HUIM), where the aim is to identify all the itemsets that are greater or equal to a given minimum utility threshold. As HUIM is a generalization of FIM, most HUIM algorithms extend FIM algorithms [4–10] but also incorporate several tailored optimizations and strategies.

HUIM algorithms can be classified into two main types: two-phase algorithms [11–14] and one-phase algorithms [15–22; 22–26]. Two-phase algorithms require two scans of the database. The initial scan generates the set of candidate high utility itemsets; the subsequent scan identifies the true high utility itemsets among these candidates. Generally, one-phase algorithms outperform two-phase algorithms as they conduct a single scan of the database and eliminate the need to generate candidates. Among one-phase algorithms, list-based algorithms [15; 18; 21; 22; 24; 27] have garnered significant attention in recent years due to their excellent performance. However, a key limitation of list-based algorithms that hinders their performance is that they rely on a list join operation for exploring the search space of itemsets. Three main types of join operations have been suggested, with time complexities [15; 21; 22] of $O(n \log m)$, $O(n + m)$ and $O(n)$, where m and n denote the numbers of entries in two utility-lists being joined. Despite this, joins remain the main bottleneck of list-based algorithms. Thus, an important question arises: is it possible to entirely eliminate the join operation?

In this paper, we answer this question positively by proposing a novel list-based algorithm called FOTH (Fast sOrted iTemset searcH), which leverages a new data structure called IndexSet. First, the item space is converted into a binary one. Each item x_i is encoded as a binary number. The parameter i is the position of item x_i, when items are sorted in Transaction-Weighted Utilization [11] descending order. Using these codes, every itemset can be encoded with its items' codes, which allows sorting itemsets for efficient search.

Second, a new method for generating itemsets without the join operation, called propagation, is designed. Unlike previous algorithms, there is no need to compare the transaction identifiers of lists. During the propagation process, a transaction can generate itemsets with utility information called vectors, by itself. These vectors can also continue to generate sub-vectors without repetition.

Third, an IndexSet structure is designed to mine high utility itemsets. An IndexSet for an itemset X contains all the utility information of X, that is its vectors. IndexSets are all sorted in a red-black tree [28]. When vectors are about to be merged into

*Corresponding authors. E-mail addresses: yiheyan@std.uestc.edu.cn (Y. Yan), xinzhengniu@uestc.edu.cn (X. Niu), zhihengzhang406@163.com (Z. Zhang), philfv@qq.com (P. Fournier-Viger), 22653231@qq.com (L. Ye), minfan@swpu.edu.cn (F. Min)

the IndexSets, the ordered IndexSet structure can be obtained in $O(\log n)$ complexity.

The proposed FOTH algorithm performs a new type of search that is different from a depth-first or breath-first search. A special set of IndexSets called *mining area* is introduced to control the mining process. Although there is no join operation in FOTH, there is still a performance overhead for propagation. The complexity analysis proves that FOTH is faster in mining large itemsets in theory.

Experiments are conducted in various databases to test the performance of FOTH. The results show that FOTH outperforms several state-of-the-art list-based algorithms, except for sparse databases.

The rest of this paper is organized as follows: Section 2 introduces the related work on HUIM. Section 3 describes the problem definition. Section 4 presents the design of FOTH. Then, Section 5 describes the experimental results. Finally, Section 6 draws a conclusion.

2. Related work

Finding all the high utility itemsets in a transaction database with N items and M transactions is a very challenging task. There are 2^N possible high utility itemsets even for a small N, M, and a low threshold. To simplify the HUI mining (HUIM) problem, several approximate HUIM algorithms were designed such as HUIM-SA [29], HUIM-BPSO [30] and HUIM-AF [31]. Approximate algorithms are generally very fast since they use heuristic strategies to explore the search space, but they cannot guarantee to find all the HUIs. Based on the Apriori algorithm [4], Liu et al. proposed the first complete algorithm for HUIM, named Two-Phase [11] in 2005. Thereafter, more and more researchers began developing exact algorithms to improve the efficiency of HUIM in terms of time and memory.

The Two-Phase algorithm is applied in two steps (phases): generating candidate high utility itemsets and then filtering those that are low utility itemsets. To reduce the search space, Two-Phase [11] introduced an important monotone upper-bound on the utility, named the TWU (Transaction-Weighted Utilization) measure [11]. If the TWU value of an itemset is less than the user-defined threshold, all its supersets can be ignored, as they cannot be high utility itemsets [11]. The TWU measure has been used by many subsequent HUIM algorithms due to its low computation cost and its relatively powerful effect on search space reduction. However, Two-Phase [11] may generate many itemsets that do not exist in the database, and perform multiple costly database scans. To address those issues, some pattern-growth algorithms were developed based on FP-Growth [5] such as UP-Growth [12], HUP-Growth [32], and MU-Growth [14]. These pattern-growth algorithms use a database projection technique and a tree-based database representation to avoid generating unexisting itemsets. A projected database usually contains fewer items and transactions than the original one, which reduces the cost of database scanning.

Nevertheless, the aforementioned algorithms still perform two phases, and thus have to keep a large number of candidates in memory during phase 1 before performing phase 2, which degrades performance [15]. For this reason, one-phase algorithms were designed. The first two algorithms of this type, HUI-Miner [15] and d^{2}HUP [16], were introduced in 2012. Different from two-phase algorithms, one-phase algorithms transform a database into vertical or horizontal data structures using a single database scan. Then, these structures are used to directly search for high utility itemsets (without keeping candidates in memory).

Inspired by the Eclat algorithm [6] for FIM, the HUI-Miner [15] algorithm relies on a data structure called utility-list. A utility-list stores information about an itemset's utility and the transactions where it appears. A new itemset and its utility-list can be constructed by joining the utility-lists of some of its subsets. The itemset's utility can then be directly derived from its utility-list without reading the database. Besides, an upper-bound tighter than the TWU was also proposed in HUI-Miner [15], based on the concept of remaining utility.

Thereafter, various list-based algorithms and optimization strategies were proposed. The FHM [18] algorithm introduced a search space pruning strategy based on the TWU of item pairs. Thanks to this optimization, FHM performs fewer join operations than HUI-Miner [15], increasing its performance. HUP-Miner [20] utilizes a partitioned utility-list data structure to divide a database into many parts. Moreover, two pruning strategies, *LA-Prune* and *PU-Prune*, are applied to reduce the search space. IMHUP [23] links elements from the same transactions to reduce the cost of the join operation, while mHUIMiner [24] combines a utility-list with a prefix-tree structure to reduce the number of generated itemsets. To reduce the cost of the join operation, ULB-Miner [27] uses a buffer to efficiently store and retrieve utility-lists, and reuse memory during the mining process. HMiner [19] uses a compact utility-list structure with virtual hyper-links for the same purpose. HUI-Miner* [21] takes the same idea but uses a different structure called utility-list*. The join operation complexity is then reduced to $O(m+n)$. The UBP-Miner [22] algorithm introduces the utility bit partition list structure where transaction identifiers are encoded into a binary form. Then, the join operation is done using the bitwise AND operation, and the complexity is reduced from $O(m + n)$ to $O(n)$.

As outlined above, many researchers have worked on designing efficient join operations. This is because joins are the main performance bottleneck of list-based HUIM algorithms [18]. However, previous studies have not found a method to avoid the join operation completely. Designing a list-based algorithm without join operations has thus great potential.

Many extensions of High Utility Itemset Mining (HUIM) have also been extensively studied. For instance, it was observed that for a low minimum utility threshold, too many itemsets may be output. And it can be hard for users to find interesting patterns among them. To address this issue, algorithms were developed to extract concise representations of high utility itemsets, that is a smaller set of representative high utility itemsets. Several types of representations have been proposed, such as Closed High Utility Itemsets [33–36], Maximal Itemsets [37; 38], and Generators of High-Utility Itemsets [39; 40]. Another challenge is how to choose an appropriate threshold.

In traditional HUIM algorithms, this value is specified by users, who may not have prior knowledge of the utility distribution in the database. The task of Top-k high utility itemset mining was proposed to solve this problem, which allows users to discover the k itemsets with the highest utilities in a database. Several efficient algorithms have been proposed for this problem based on Frequent Itemset Mining (FIM) and HUIM techniques [41–43]. This paper focuses on the traditional HUIM problem.

3. Preliminaries

This section introduces preliminaries and the problem definition. First, a database D is given by users.

Definition 1. (*Database* [3]) A database is a triple

$$D = (T, I, q) \tag{1}$$

where $T = \{t_1, t_2, \ldots, t_M\}$ is the set of $Tids$ (transaction IDs), $I = \{x_1, x_2, \ldots, x_N\}$ is the ordered set of all items from the database, and $q : I \times T \rightarrow Z^+$ is an injective function indicating the quantity of each item in each transaction. Moreover, Z^+ is the set of non-negative integers. Without losing generality, we assume that the order is $x_1 < x_2 < \cdots < x_N$.

Definition 2. (*Item utility* [3]) Given a database D, the unit utility of an item $x \in I$ is $eu(x) \in R^+$. Moreover, R^+ is the set of positive real numbers.

Therefore, given an item $x \in I$, its utility in a transaction $t \in T$ is

$$f(x, t) = eu(x) \times q(x, t). \tag{2}$$

Definition 3. (*Itemset* [3]) Given a database D and an ordered index list $i_1 < i_2 < \cdots < i_k$, $X = \{x_{i_1}, x_{i_2}, \ldots x_{i_k}\}$ is called an itemset if $\forall j \in [1, k]$, $x_{i_j} \in I$ and $i_j \in [1, N]$.

Given a database D and an itemset X, X matches a transaction t iff $\forall x \in X, q(x, t) \neq 0$. Formally, the matching function of X on $t \in T$ is

$$m(X, t) = \begin{cases} 1, & \text{if } X \text{matches } t; \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Given a database D and an itemset X, the utility of X in a transaction $t \in T$ is

$$u(X, t) = m(X, t) \times \sum_{x \in X} f(x, t). \tag{4}$$

Therefore, the utility of X in the whole database D is

$$U(X, T) = \sum_{t \in T} u(X, t). \tag{5}$$

Example 1. An example database is shown in Table 1 and the unit utility of items is shown in Table 2. In this database, $T = \{t_1, t_2, \ldots, t_6\}$ and $I = \{a, b, \ldots, g\}$ is sorted in alphabetical order. The utility of item b in transaction t_1 is $eu(b) \times q(b, t_1) = 2$. Itemset $\{b, c\}$ appears in t_4 and t_6, then $U(\{b, c\}, T) = u(\{b, c\}, t_4) + u(\{b, c\}, t_6) = 8 + 4 = 12$. The utility value represents the importance of an itemset (e.g., profit).

Table 1: The example database

Tid	Items	Quantity
t_1	$\{a, b, d, e\}$	1, 1, 1, 3
t_2	$\{d\}$	3
t_3	$\{e, f, g\}$	1, 2, 3
t_4	$\{b, c, d, e\}$	3, 2, 2, 4
t_5	$\{a, d, g\}$	3, 3, 3
t_6	$\{b, c, d, g\}$	1, 2, 2, 3

Table 2: Unit utility

Item	a	b	c	d	e	f	g
Unit utility	4	2	1	1	2	6	3

Let σ be the minimum utility threshold. An itemset X is a high utility itemset iff

$$U(X, T) \geq \sigma. \tag{6}$$

The goal of the High Utility Itemset Mining (HUIM) problem is to discover all the high utility itemsets in D.

Example 2. In the example database, let $\sigma = 15$. We have $U(\{b, c\}, T) = 12$ and $U(\{b, c, d, g\}, T) = 15$. Thus, $\{b, c, d, g\}$ is a high utility itemset and $\{b, c\}$ is not.

Because HUIM is an NP-hard problem, pruning strategies must be used to reduce the search space. Two popular pruning strategies called TWU-Prune and RU-Prune are presented next. The TU (Transaction Utility) [11] of a transaction $t \in T$ is denoted as

$$TU(t) = \sum_{y \in I} f(y, t). \tag{7}$$

The TWU [11] of an item x in D is calculated by

$$TWU(x, T) = \sum_{t \in T} m(\{x\}, t) \times TU(t). \tag{8}$$

Property 1. (*TWU-prune* [11]) Let Y be any itemset that contains an item x. Y is not a high utility itemset if $TWU(x, T) < \sigma$.

Example 3. In the example database, all the TWU of items in I are shown in Table 3. Let $\sigma = 35$, then any superset of $\{c\}$ or $\{f\}$ is not a high utility itemset.

Property 1 provides a method to reduce the size of transactions in the input database. Revised transactions are easier to mine because they contain fewer items. We say that a transaction $\forall t \in T$ is revised if $q(x, t)$ has been changed to $q(x, t) = 0$ if $x \in \{x \in I \wedge TWU(x, T) < \sigma\}$ in the database D.

Let X be an itemset and $y \in I$ be an item in D. We say that $X < y$ iff $\forall x \in X, x < y$. The remaining utility [15] of X is

$$RU(X, T) = \sum_{t \in T} (m(X, t) \times \sum_{X < y} f(y, t)). \tag{9}$$

Property 2. (*RU-prune* [15]) Let X be an itemset. Any superset of X is not high utility if $U(X, T) + RU(X, T) < \sigma$.

Table 3: TWU of items

Item	*a*	*b*	*c*	*d*	*e*	*f*	*g*
TWU	37	46	33	73	54	23	62

Example 4. Let items be sorted in alphabetical order in the example database, then $RU(\{b, d\}, T) = 6 + 8 + 9 = 23$ and $U(\{b, d\}, T) = 3+8+4 = 15$. If σ is set to 40, then any superset of $\{b, d\}$ is not a high utility itemset because $RU(\{b, d\}, T) + U(\{b, d\}, T) = 38$.

4. FOTH Algorithm

This section presents the novel FOTH algorithm to mine high utility itemsets. First, the proposed IndexSet structure and propagation mechanism are explained. Then, the algorithm is described, and finally, a complexity analysis is presented.

4.1. IndexSet structure and propagation

Traditional list-based HUIM algorithms apply a join operation to generate new itemsets while calculating their utility. In the FOTH algorithm, joins are replaced by a novel merge operation. We first describe how to efficiently implement this operation using an encoding method for all items of I. This method allows to efficiently compare, sort itemsets, and access itemsets to update their utility information. The encoding method assigns a unique number to each item from I as follows.

Definition 4. (*Encoding*) Let $x_i \in I$ be the i-th item. The code of x_i is a $|I|$ bits binary number, denoted as $code(x)$ and is equal to

$$B(2^{i-1}), \tag{10}$$

where B : decimal $\rightarrow$ binary. For an itemset X, it be can be encoded by

$$code(X) = \sum_{x \in X} code(x). \tag{11}$$

Thus, the code is a binary number where the i-th bit is set to '1' iff $x_i \in X$

Example 5. Itemset $\{a, d\}$ is encoded by the binary number $(0000001)_2 + (0001000)_2 = (0001001)_2$. The item a and d are respectively represented as $(0000001)_2$ and $(0001000)_2$ because they are the first and fourth items according to the alphabetical order used in this example. For convenience, we will omit the notation $(\cdot)_2$ when referring to itemset codes in this paper.

From an implementation perspective, an item's code may be too large to fit into a single variable. Therefore, a code array of integers is utilized to represent each code.

Definition 5. (*Code array*) Let $\mathbf{a}$ be an integer array and $\mathbf{a}[i]$ be the i-th integer that has k bits. Given an itemset X, the code array $\mathbf{a}$ of X is calculated by

$$\mathbf{a}[l] = \sum_{x \in X \wedge code(x) \in [2^{kl}, 2^{k(l+1)})} code(x) \gg kl \tag{12}$$

where '$\gg$' is the right shift operator.

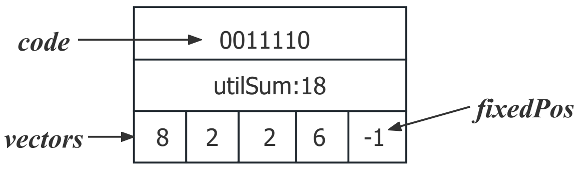


Figure 1: IndexSet for $\{b, c, d, e\}$.

Example 6. If $code(X) = 2^{100}$ and $k = 32$, the code array of X is $\langle 000\ldots010000, 0, 0, 0\rangle$ in binary form.

Based on this encoding method, the proposed IndexSet structure of an itemset X has four fields:

1. *code*: a number that represents the itemset, that is $code(X)$
2. *utilSum*: the utility of the itemset, that is $U(X, T)$.
3. *vectors*: a list of vectors. A vector is the sub-structure of IndexSet that contains the utility information of X in a transaction t. And one vector is encoded in the same way. And every vector of *vectors* has the same code as X.
4. *fixedPos*: a variable used to facilitate the itemset propagation process.

Each transaction is an itemset itself. Thus, after the first database scan, IndexSets are constructed for every transaction. Figure 1 shows an IndexSet for t_4. All the IndexSets are stored in a red-black tree [28] because it has an excellent average complexity for insertion and search. Note that it could be beneficial to use other structures supporting the same operations in some specific situations such as a hash map if there are few IndexSets.

The second important component of FOTH is a new method that allows each itemset to propagate information to their subsets. To achieve this, a key challenge is that different vectors coming from a given transaction may propagate to the same vector during the propagation process, resulting in redundant propagation and an incorrect result. This is explained with an example. Consider the IndexSet of $\{b, c, d, e\}$ shown in Figure 1. The IndexSet $\{b, c, d, e\}$ can generate many smaller subsets as vectors but only $\{b, c, d\}$ and $\{b, c, e\}$ will make utility contributions to $\{b, c\}$. In particular, vectors of $\{b, c, d\}$ and $\{b, c, e\}$ can both propagate to vector $\{b, c\}$, and thus vectors from T_4 may be collected twice by the IndexSet $\{b, c\}$. This will lead to a miscalculation because $u(\{b, c\}, t_4)$ must be counted only once. Although this problem could be avoided if all the vectors of T_4 could be generated at the same time, it is a very tough task to do because if a transaction has l items, there will be 2^l vectors. Hence, the propagation process should be executed step by step and a mechanism should be devised to avoid redundant propagation. For this purpose, a *fixedPos* value is introduced in the IndexSet structure.

A vector has a pointer *fixedPos* that points to an item from the itemset represented by the vector. Items that are greater than or equal to *fixedPos* are fixed and cannot be removed during propagation to generate other vectors. Each IndexSet propagates its vectors according to *fixedPos*. If *fixedPos* is equal to -1, it means that no item is fixed and any item can be removed. The pseudo code of the propagation process is shown in Algorithm

Algorithm 1 Propagate.

```
Input: $IS$, an IndexSet; $R$, a red-black tree.
 1: for all vector $v$ in $IS.vectors$ do
 2:   for all item $x$ in $\{x|x \in v \wedge x < v.fixedPos\}$ do
 3:     $c \leftarrow IS.code$;
 4:     for all item $y$ in $\{y|y \in v \wedge y \leq x\}$ do
 5:       $sub\_v \leftarrow$ a copy of $v$;
 6:       remove item $y$ from $sub\_v$;
 7:       $sub\_v.utilSum = sub\_v.utilSum - u(y, v)$;
 8:       $c = c - y.code$; // calculate code of $sub\_v$.
 9:       $sub\_v.code \leftarrow c$;
10:       $S = R$.find($sub\_v.code$);
11:       $sub\_v.fixedPos \leftarrow$ the position of the next largest
          item that is smaller than $y$;
12:       if $S == None$ then
13:         $S \leftarrow$ an empty IndexSet;
14:         $S.utilSum \leftarrow 0$;
15:         $S.code \leftarrow sub\_v.code$;
16:         $R$.put($S$);
17:       end if
18:       $S.utilSum = S.utilSum + sub\_v.utilSum$;
19:       $S.vectors$.append($v$) ;
20:     end for
21:   end for
22: end for
```

1. This procedure takes an IndexSet and a red-black tree as input, and consists of three loops to iterate over all vectors of the input IndexSet. When a vector v is being propagated, a copy of IS'code is stored in a variable c (line 3). All the codes of v's sub-vectors are derived from c. Sub-vectors of v are constructed by operations in lines 5-9. Lines 10-19 show how to merge vectors with an IndexSet.

To describe the propagation process more clearly, take the propagation of IndexSet $\{b, c, d, e\}$ as an example, which is illustrated in Figure 2. At first, the algorithm selects a vector as the parent. Here, it is $\{b, c, d, e\}$, which is represented by the code 0011110. The notation [0011110] represents a vector structure with the code 0011110. As this vector's *fixedPos* is equal to -1, all items can be removed. The position of the next largest item after *fixedPos* is item 5. Hence, the new *fixedPos* is set to 5 while the parent is turned into a set of vectors, called base-1. This is done by removing each '1' one by one from *fixedPos* to generate the sub-vectors: [0001110], [0000110], [0000010] and [0000000]. The *fixedPos* value of each sub-vector is set to the position of the next largest item after the removed one. In the figure, a value '1' in red indicates that *fixedPos* has been changed by the algorithm. The real *fixedPos* value in a vector is the smallest red '1'. After base-1 has been generated, the *fixedPos* of base-1 is moved again, turning base-1 into base-2. The same process is performed until *fixedPos* reaches the last position. All the sub-vectors are merged into the IndexSets that have the same code as them in the red-black tree.

Merging vectors is relatively simple. For instance, assume that there already exists an IndexSet $\{b, d\}$ from transactions t_1 and t_6. The vector $\{b, d\}$ from the transaction t_4 will be merged with this IndexSet. As the algorithm does not pay attention to *fixedPos* and the order during the merging process, *fixedPos* is replaced by unknown parameters x and y for convenience. The vector $\{b, d\}$ can be simply appended to the end of the *vectors* field. And *utilSum* will be updated to $7 + 8 = 15$. The merge operation for a vector shown in Figure 3 is completed.

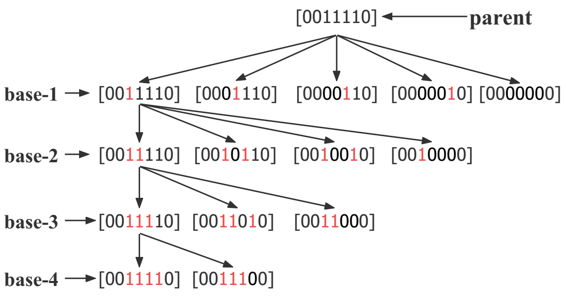


Figure 2: Propagation of vector $\{b, c, d, e\}$ for one step.

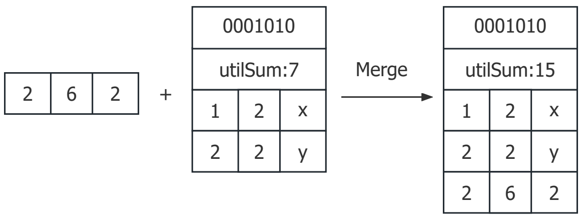


Figure 3: Merge operation for vector $\{b, d\}$ of t_4.

4.2. The FOTH procedure

In this section, the FOTH algorithm is described. In general, FOTH is composed of two processes: reconstruction and mining.

The reconstruction process takes a database D and a positive real number σ as input. The pseudocode is shown in Algorithm 2. The goal of this process is to convert all transactions in D into IndexSets. Then, these IndexSets are sorted in R. First, each item is encoded (lines 1-5). If the number is too large, an integer array is used. Second, transactions are transformed into vectors (lines 8-10) and merged into the IndexSets (lines 18-19). When looking up for an IndexSet in the red-black tree by its code, if there is no IndexSet, an empty one is created and initialized (lines 12-17). From Algorithm 1 and Algorithm2, the most two important operations of R are 'put' and 'find'. If there are not too many IndexSets in R, a hash map will be better for these two operations. Using a hash map, a particular IndexSet can be found in $O(1)$ time complexity if there is no collision.

The mining process is the main part of FOTH. Instead of using a depth-first or breadth-first search, an approach similar to a binary-search is applied on all 2^n combinations, as shown in Algorithm 3. After the reconstruction process is completed, a loop is created to search for high utility itemsets. First, a cut point denoted as *low* is initialized to 2^{n-1}. All the IndexSets which are bigger than or equal to *low* are collected into a set called *mining area*. The sum of the *utilSum* of all IndexSets in the *mining area* is accumulated, denoted as *vs*. It is actually the sum of $U(X, T)$ and $RU(X, T)$ where X is the itemset encoded by *low*. Therefore, *vs* can be used to estimate whether there are high utility itemsets in this *mining area*. This feature can be maintained automatically if IndexSets are sorted in ascending order. For example, given two codes 01100 and 11001, *low* is set to 01000. A code c which is larger than *low* satisfies the equation $low \,\&\, c = low$ where & is the bitwise AND operator. For example, $01000 \,\&\, 01100 = 01000$. This means that c has all of *low*'s items. But if c is too large, this equation will be broken like $01000 \,\&\, 11001 \neq 01000$. Therefore, a binary-search-like search is performed to control the highest bit.

Back to the mining process, since *vs* is equal to $U(X, T) + RU(X, T)$, *RU-Prune* can be applied to reduce the search space. If $vs < \sigma$, there will be no high utility itemset in the current *mining area*. If not, *low* is set to $low + low/2$, $low + low/4$, $low + low/8$, ..., $low + 1$ and the *mining area* also changes with *low*. These steps are performed recursively until there is no IndexSet left in R. Whether *vs* is smaller than σ or not, IndexSets must be propagated after the *mining area* is searched. The pseudocode of the mining process is shown in Algorithm 3.

So far, all the necessary processes of FOTH have been introduced. But an additional, simple process shown in Algorithm 4 is needed to start FOTH.

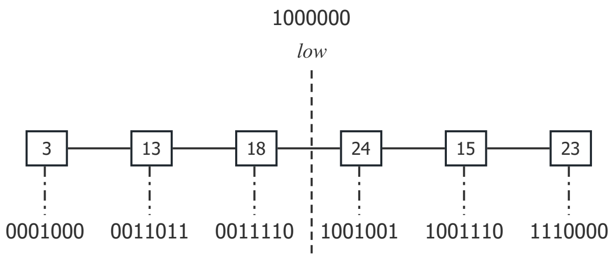


Figure 4: IndexSets of the example database.

4.3. Mining high utility itemsets using FOTH

To more clearly describe how the FOTH works, consider the example database previously presented and σ is set to 20. First, Algorithm 4 is called to start FOTH. Second, the reconstruction process (Algorithm 2) is invoked for initialization. In this example, items are first sorted in alphabetical order, for convenience (line 4). But in practice, FOTH actually sorts items in the descending order of TWU. In this way, there will be fewer IndexSets because FOTH always attends to split low bits. The utility of items that have large TWU will be recombined with smaller ones. It was also shown in prior work that this order is effective for HUIM [21]. After reconstruction, the whole search space is 2^7 because there are 7 items in the database. IndexSets are sorted as shown in Figure 4, and *low* is initialized to 2^6 first.

Third, the mining process (Algorithm 3) starts. The *mining area* is set to {[1001001], [1001110], [1110000]} which are all larger than *low* (line 1). Because the sum of the *utilSum* is $24 + 15 + 23 = 62 > \sigma$ in the current *mining area* (lines 2-5), there may exist high utility itemsets (line 7). Therefore, *low* is updated to $1000000 + 0100000 = 1100000$, and Algorithm 3

Algorithm 2 Reconstruction.

```
Input: $D$, a transaction database.
Output: $R$, a red-black tree initialized with $D$'s transactions;
        $n$, the number of items in $D$.
 1: scan $D$ and calculate the TWU of all the items in $D$;
 2: revise all the transactions in $D$ according to the TWU;
 3: $n \leftarrow$ the remaining number of items;
 4: sort items in TWU descending order;
 5: encode all the remaining items;
 6: create an empty red-black tree $R$;
 7: for all transaction $T$ in $D$ do
 8:     $v \leftarrow$ vector form of $T$;
 9:     $v.utilSum \leftarrow$ the sum of items' utility in $T$;
10:     $v.code \leftarrow$ the sum of all items' codes in $T$;
11:     $IS = R$.find($v.code$);
12:     if $IS == None$ then
13:         $IS \leftarrow$ an empty IndexSet;
14:         $IS.utilSum \leftarrow 0$;
15:         $IS.code \leftarrow v.code$;
16:         $R$.put($IS$);
17:     end if
18:     $IS.utilSum = IS.utilSum + v.utilSum$;
19:     $IS.vectors$.append($v$);
20: end for
21: return $R, n$;
```

is invoked recursively (line 10). Then the *mining area* is set to {[1110000]} (line 1). Since $23 > \sigma$, *low* is changed to 1110000 and Algorithm 3 is called again. The next *mining area* is set to {[1110000]}. The sum of the current *mining area*'s *utilSum* is $23 > \sigma$. The condition of line 7 is still satisfied. Algorithm 3 is executed and *low* becomes 1111000. Since there is no IndexSet larger than the *low*, the current recursive invocation concludes and control returns to the last call level (line 10) where *low* is 1110000. And the *mining area* is {[1110000]}. With no IndexSet large enough, the for-loop (lines 9-11) terminates early. Execution proceeds to lines 13-16. The *utilSum* of IndexSet [1110000] is $23 > \sigma$ (line 14). Hence, $\{e, f, g\}$ qualifies as a high utility itemset after decoding (line 15).

Fourth, the IndexSet [1110000] undergoes the propagation process commencing at line 17 (Algorithm 1), propagating five vectors to other IndexSets as depicted in Figure 5. The propagation process has been explained in Section 4.1. Thus, further details are omitted here. After propagation, the mining process returns to the last *mining area* and resumes on line 13 where *low* is 1100000. The IndexSet {[1100000]} is then output as a high utility itemset (lines 14-15) and propagated (line 17). Next, the algorithm reverts to the preceding call level (line 10) with $low = 1000000$. The for-loop (lines 9-10) continues and sets $i = 2$. The method *Mine* is called with $low = 1010000$. These steps are repeated until all the high utility itemsets have been identified. The results are shown in Table 4.

4.4. Complexity

Let there be an itemset P, and two itemsets Px and Py, where the notation Px means $P \cup \{x\}$. The lists of Px and Py are

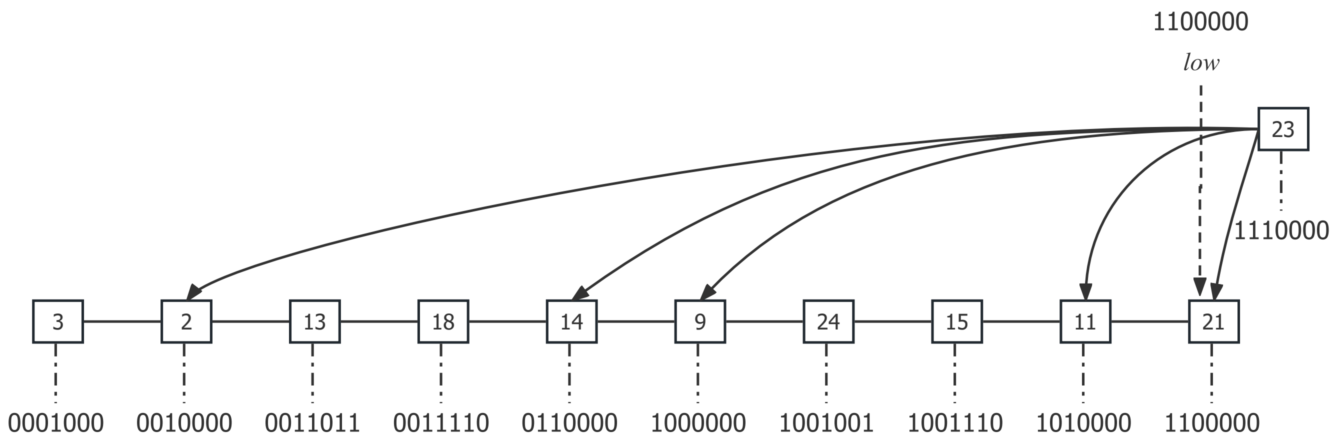


Figure 5: Propagation of [1110000].

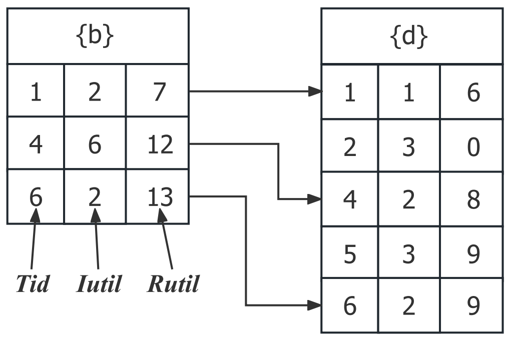


Figure 6: The join operation between two utility-lists.

Table 4: The itemsets mined from the example database

Itemsets	Utility	Itemsets	Utility
$\{e, f, g\}$	23	$\{b, e\}$	22
$\{f, g\}$	21	$\{b, d, e\}$	25
$\{a, g\}$	21	$\{g\}$	27
$\{a, d, g\}$	24	$\{d, g\}$	23
$\{a, d\}$	20		

denoted as $list(Px)$ and $list(Px)$. Let D be a database which has N items and M transactions. Without losing generality, assume that $list(Px)$ contains n transactions and $list(Py)$ contains m transactions where $n \leq m \leq M$. There are two necessary processes to generate an itemset Pxy: constructing and discovering.

The join operation is used by algorithms such as HUI-Miner [15] is illustrated in Figure 6. It consists of searching for lines that have the same *Tids* in both lists. Although a binary search can be performed on the *Tids* of $list(Py)$, the complexity is up to $O(n \log m)$. Thus, algorithms such as ULB-Miner [27] instead perform the join as a two-way scan, which has a complexity of $O(n + m)$. The fastest join operation in UBP miner, called BEO, is $O(n)$ [22]. But Wu et al. ignored the complexity of encoding each *Tidset* and handling hash collisions in UBP-Miner [22]. Because a *Tidset* may be too large to be stored in an integer variable, an array must be used. The encoding complexity and that of handling hash collisions must be considered. If an integer has L bits, then the actual complexity of BEO increases to $O(n2^N + M/L)$ in the worst case.

In the proposed FOTH algorithm, the complexity of encoding is $O(N/L)$ and the complexity of searching an IndexSet in R is $O(N)$ or $O(1)$. It depends on the red-black tree and the number of hash collisions. Different structures (hash map or red-black tree) will be investigated for specific databases. A red-black tree is more general but may not be the fastest in spe-

Algorithm 3 The mining process.

Input: R, a red-black tree returned by Algorithm 2; σ, a minimum utility threshold; *low*, a cut point which is initialized to 2^{n-1}.
Output: all high utility itemsets.
Method: Mine.

```
1:  $mining\_area \leftarrow$ a set of IndexSets which are larger than $low$ in $R$;
2:  $vs \leftarrow 0$; // the sum of $utilSum$.
3:  for all IndexSet $IS$ in $mining\_area$ do
4:      $vs = vs + IS.utilSum$;
5:  end for
6:  item $x \leftarrow$ the smallest item satisfying $low$;
7:  if $vs \geq \sigma$ then
8:      // U-prune.
9:      for $i = 1, 2, 3, \ldots, \log_2 low$ do
10:         Mine($R$, $low + low/2^i$);
11:     end for
12: end if
13: for all IndexSet $IS$ in $mining\_area$ do
14:     if $IS.utilSum \geq \sigma$ then
15:         decode $IS.code$ and output;
16:     end if
17:     Propagate($IS$, $R$);
18:     $R$.remove($IS$);
19: end for
```

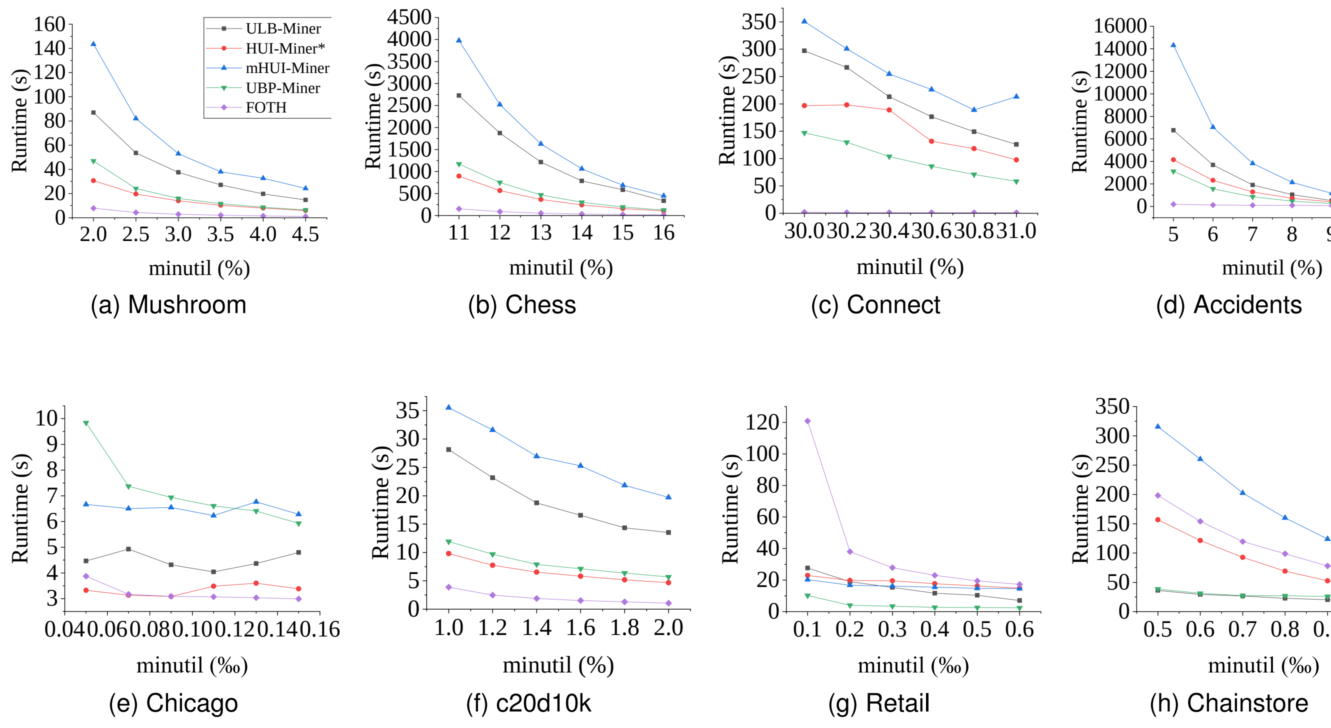


Figure 7: Runtime comparison.

Algorithm 4 FOTH booting function.

Input: D, a transaction database; σ, a minimum utility threshold;

Output: all high utility itemsets;

```
1: $R, n \leftarrow$ Reconstruction($D, \sigma$);
2: for $i = n-1, n-2, n-3, \dots, 1$ do
3:     Mine($R, 2^i$);
4: end for
```

cific situations. The complexity of generating a sub-vector is $O(1)$ because FOTH only needs to remove a '1' and set *fixedPos*. Therefore, the complexity of constructing a complete IndexSet is $O(MN^2/L)$ in the worst case. Compared to the join operation, FOTH does not construct the complete *Pxy* at once from two smaller existing utility-lists. Instead, FOTH merges vectors propagated by other IndexSets into the IndexSet of *Pxy* until this IndexSet is complete. This merging operation does not perform any kind of *Tids* comparison. It just needs to append a new vector to the end of the filled *vectors* and update *utilSum* in $O(1)$ complexity.

In the worst case, FOTH needs to generate an itemset X that appears in all transactions, that is $n = m = M$ and σ is set to 0. The complexities of the join operation of previous algorithms are $O(M \log M)$, $O(2M)$ and $O(M2^N + M/L)$. In practice, L usually is equal to 32. It seems that $O(2M)$ is the fastest join operation when $N \geq 8$. If L is equal to 64, the critical condition will be changed to $8\sqrt[2]{2}$. However, the worst case will not happen frequently in practice. And high utility itemsets rarely cause numerous hash collisions. For example, if D is a supermarket transaction database, it is unlikely that all customers will buy exactly the same set of items. Therefore, the complexity of BEO and FOTH is still $O(n)$ in most cases.

The above analysis is about constructing an itemset X. Let us consider the complexity of discovering X. If an itemset $|X| = k$, we say that X is a k-itemset. Without considering any pruning strategy, previous algorithms have to generate 1-itemsets, 2-itemsets, 3-itemsets, ... , $(k-1)$-itemsets first. The total number of them is $\sum_{i=1}^{k-1} \binom{N}{i}$. In FOTH, only the itemsets whose codes are larger than X's may contribute to X. As $|X| = k$, $code(X)$ is greater than or equal to 2^k. Therefore, there are at most 2^{N-k} itemsets that not only contain X but are also larger than X's code. Comparing these two numbers, there exists a k' that makes

$$\sum_{i=1}^{k'-1} \binom{N}{i} = 2^{N-k'}.$$

And for $k \geq k'$, $2^{N-k} \leq \sum_{i=1}^{k-1} \binom{N}{i}$. Therefore, FOTH is faster in generating large itemsets.

In summary, FOTH has the same minimum complexity for construction as BEO. But FOTH uses less steps to discover large itemsets. Therefore, FOTH is expected to perform better than other list-based algorithms for mining large high utility itemsets.

5. Experiment

In this section, experiments are conducted to evaluate the performance of FOTH. Four recent list-based algorithms are compared with FOTH, namely UBP-Miner [22], HUI-miner* [21],

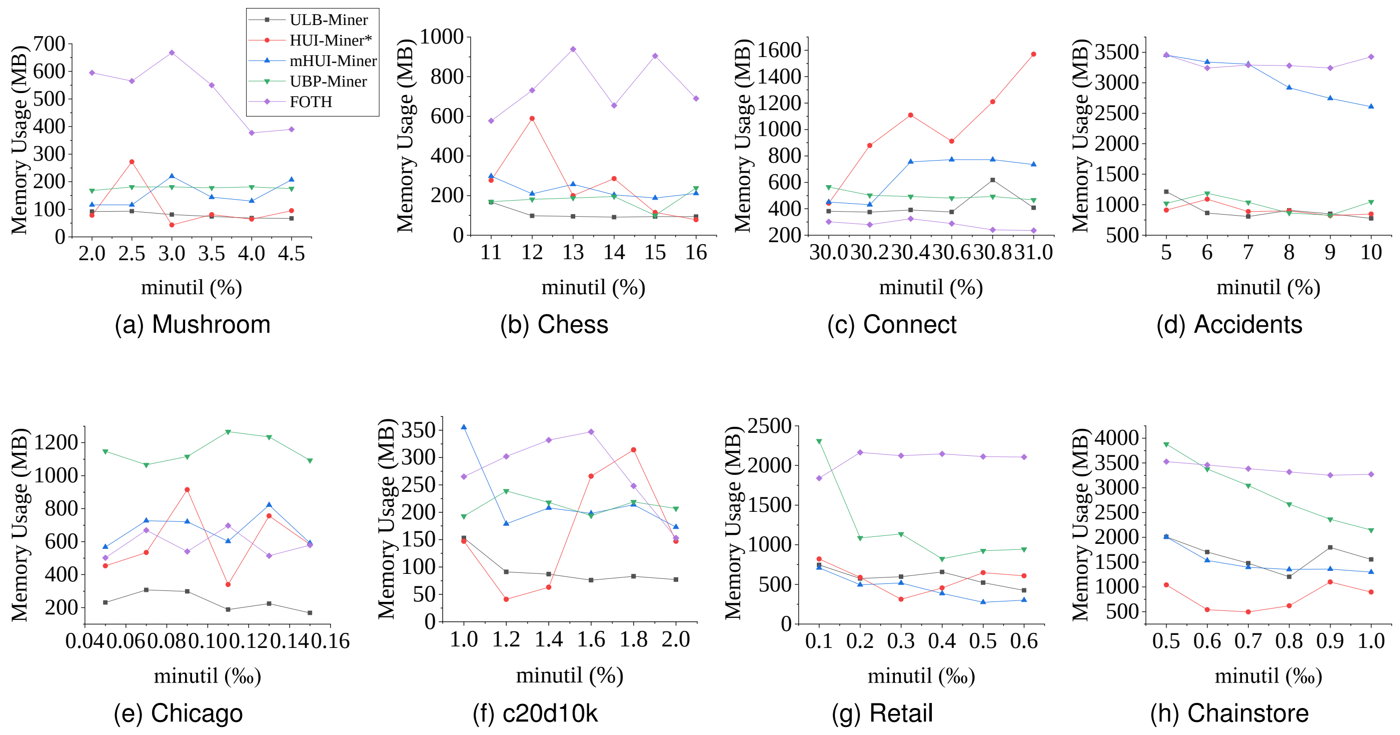


Figure 8: Memory consumption comparison.

Table 5: The information of databases.

Database	Trans	Items	AvgLen	Density	Size (MB)
Mushroom	8,416	119	23.0	19.33%	1.0
Chess	3,196	75	37.0	49.33%	0.6
Connect	67,557	129	43.0	33.33%	16.2
Accidents	340,183	468	3.8	7.22%	63.4
Chicago	2,662,309	35	1.8	5.13%	27.7
c20d10k	10,000	192	20.0	10.42%	1.3
Retail	88,162	16,470	1,030.0	0.06%	6.5
Chainstore	1,112,949	46,086	7.2	0.02%	79.2

mHUI-Miner [24], and ULB-Miner [27]. All of them have different implementations of the join operation. These algorithms' implementations can be obtained from the SPMF website [44].

FOTH and other algorithms are implemented in Java. All these algorithms were run on a machine equipped with an Intel Core i7 processor and 16 GB of memory.

5.1. Experimental setup

5.1.1. Databases information

Eight standard benchmark databases were used to conduct the experiments. Detailed information about them is given in Table 5. All the databases can be obtained from the SPMF website [44], which is an open-source data mining library. In the table, *AvgLen* refers to the average item count per transaction. The density of a database is defined as

$$density(D) = \frac{AvgLen}{Items}, \tag{13}$$

where D is a database from Table 5. The *density* value is generally an indicator of the probable size of itemsets in a database. Depending on the density, we can roughly divide databases into dense databases and sparse databases. There is no strict boundary between these two types of databases. In [22], if a database's density > 5%, it will be considered dense. In this paper, we adopt the same rule. For a particular σ, dense databases are more likely to generate larger itemsets than sparse ones. Database *c20d10k* was generated by the IBM Quest Synthetic Data Generator. Except *c20d10k*, other databases in Table 5 are real. *Mushroom*, *Chess*, *Connect* and *Retail* were originally used for FIM. Database *Chicago* was used to mine fuzzy association rules by Zhang et al. [45]. Then, it has been converted to the SPMF format and shared for HUIM by Zhang et al. Further information about *Chicago* can be obtained on the SPMF website [44]. *Chainstore* is transactions from a major grocery store chain in California, USA. In the experiments, a value called *minutil* is used to represent the threshold σ as a percentage. From *minutil*, σ can be obtained as:

$$\sigma = minutil \times \sum_{t \in T} \sum_{x \in I} f(x, t). \tag{14}$$

5.1.2. Evaluation metrics

Four evaluation metrics: runtime, memory consumption, speedup, and scalability are utilized in the experimental evaluation. To obtain suitable experimental results, we have run the algorithm with many different *minutil* values. On the one hand, when *minutil* values are too large, few HUIs are found, the task of HUI mining becomes easier, runtimes are short, and it is hard to compare the algorithms. On the other hand, when *minutil*

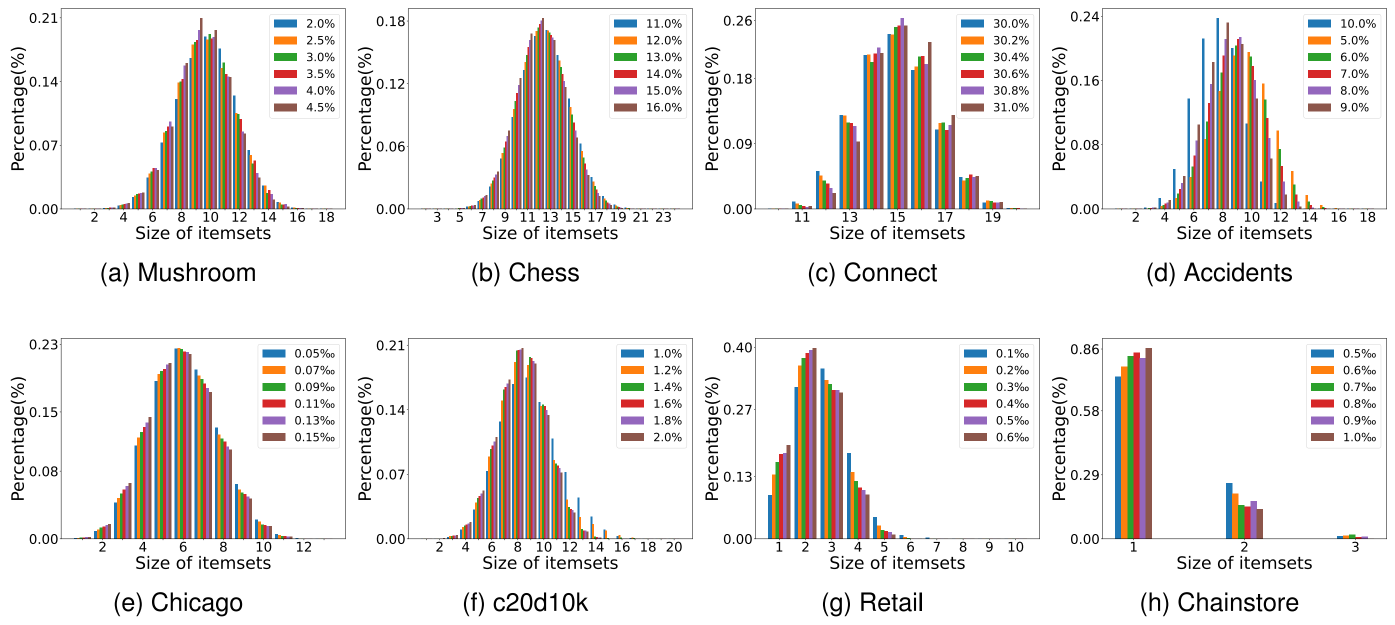


Figure 9: High utility itemsets' size under different thresholds.

values are too small, a huge number of HUIs may be found, and algorithms may have very long runtimes, which makes it time-consuming to run experiments, especially because some algorithms are considerably slower than others. Thus, to provide fair experiments, we took into account the choice of *minutil* values from other papers such as [22], and made some slight changes due to using different hardware. These *minutil* values are also used in memory consumption and speedup metrics.

For memory consumption, all the algorithms were implemented in Java and run in the Java Virtual Machine (JVM). The JVM may slightly affect the memory consumption of the algorithms. Therefore we recorded the peak memory consumption during algorithms running without losing generality. To verify the effect of FOTH's joinless approach, a speedup metric [25] is used, which is calculated by

$$\text{speedup}(x) = \frac{RJO(x)}{RSPP(FOTH)}, \tag{15}$$

where *RJO* is short for *the runtime of join operation* and *RSPP* stands for *the runtime of search and propagation processes*. It is very hard to record the *RSPP* precisely because search and propagation are deeply integrated with the other parts of FOTH. But we can still compute a lower bound by

$$\text{speedup}^*(x) = \frac{RJO(x)}{runtime(FOTH)}. \tag{16}$$

We replace *RSPP(FOTH)* by the runtime of FOTH. In this way, a lower bound on speedup(x) can be easily computed.

5.2. Runtime analysis

5.2.1. Overall runtime comparison

The runtime of the five algorithms on the databases shown in Table 5 is depicted in Figure 7. Except for databases *Retail* and *Chainstore*, FOTH is obviously faster than the others. To assess by exactly how much FOTH is faster than other algorithms, we calculated the ratio value $runtime(x)/runtime(FOTH)$ where x is another algorithm. Table 6 shows the average ratio for a particular algorithm under different *minutil* values. Besides, a significance test was conducted to determine if the experimental results have a significant difference for each pair of algorithms. Inspired by Krishna and Ravi [46], we performed the pairwise t-test between FOTH and the other algorithms, where

Table 6: The ratio value $runtime(x)/runtime(FOTH)$

Database	UBP-Miner	mHUI-Miner	HUI-Miner*	ULB-Miner
Mushroom	5.6	**19.3**	4.7	12.5
Chess	7.9	**27.5**	6.2	20.7
Connect	71.2	**186.3**	112.9	147.2
Accidents	8.0	**35.9**	11.5	17.7
Chicago	**2.2**	2.0	1.0	1.4
c20d10k	4.4	**14.7**	3.6	10.2
Retail	0.1	**0.6**	**0.6**	0.5
Chainstore	0.3	**1.6**	0.7	0.2

Table 7: The standard deviation of runtime

Database	UBP-Miner	mHUI-Miner	HUI-Miner*	ULB-Miner	FOTH
Mushroom	13.7	40.8	8.4	24.5	**2.3**
Chess	362.6	1214.6	272.3	823.6	**44.4**
Connect	31.3	55.1	40.7	61.2	**0.3**
Accidents	1027.1	4724.0	1365.5	2257.1	**42.3**
Chicago	1.3	**0.2**	**0.2**	0.3	0.3
c20d10k	2.1	5.4	1.7	5.0	**0.9**
Retail	2.7	**1.9**	2.6	6.7	36.2
Chainstore	**5.3**	76.9	40.6	6.2	46.0

Table 8: Average speedup*

Database	UBP-Miner	mHUI-Miner	HUI-Miner*	ULB-Miner
Mushroom	5.1	**19.1**	3.1	11.7
Chess	7.5	**24.0**	4.1	19.5
Connect	58.6	**150.3**	44.5	147.2
Accidents	7.5	**35.9**	5.0	17.7
Chicago	1.1	**1.5**	0.2	0.8
c20d10k	4.1	**14.1**	2.3	10.0

the database and the *minutil* thresholds are fixed. Each algorithm was executed 25 times for each test. The significance level threshold α was set to 0.05. Due to the involvement of multiple comparisons, the significance level needs to be adjusted using the Bonferroni correction. The new threshold is $\alpha/k = 0.0125$, where $k = 4$ is the number of comparisons. The results are shown in Table 9. If the p-value is below 0.0125, then the null hypothesis is rejected, which means that the experimental results are significantly different. There is only one p-value > 0.0125, where the *minutil* is 0.05 ‰ on *Chicago*. Some further observations are made.

First, the results show that for relatively dense databases, FOTH outperforms the other algorithms. However, FOTH performs badly on *Retail* and *Chainstore*, which are relatively sparse. Especially on *Connect*, FOTH is 70.2 times faster than UBP-Miner [22], 185.3 times faster than mHUI-Miner [24], 111.9 times faster than HUI-Miner* [21] and 146.2 times faster than ULB-Miner [27]. For the database *Chicago*, FOTH is not so excellent. It is just 0.4 times faster than ULB-Miner [27] and has nearly the same runtime as HUI-Miner* [21]. Except for *Connect* and *Chicago*, FOTH is 5.5 times faster than UBP-Miner [22], 23.3 times faster than mHUI-Miner [24], 5.5 times faster than HUI-Miner* [21] and 14.3 times faster than ULB-Miner [27] on average in dense databases.

Second, as the *minutil* values increase, the standard deviation of the runtime of FOTH is the smallest in most databases. Table 7 shows the standard deviation of algorithms' runtime. For the database *Chicago*, the standard deviation of FOTH is almost the same as others. In *Connect*, FOTH's standard deviation is at least 100 times smaller than for the other dense databases. For sparse databases, especially *Retail*, FOTH had a less stable performance than for dense databases.

Therefore, we can draw three conclusions from the above experimental results:

1) FOTH is faster in dense databases.
2) FOTH is more stable in dense databases.
3) FOTH does not perform well in sparse databases.

We next analyze the reasons behind these three conclusions. Conclusions 1 and 3 can be directly explained by the complexity analysis in Section 4.4. FOTH has a low complexity in mining large utility itemsets in theory. However, the relationship between density and high utility itemset's length is not absolute. A dense database can also generate numerous small high utility itemsets and vice versa. The result of runtime comparison cannot completely prove the above conclusions. Therefore, we collect all the high utility itemsets mined in different

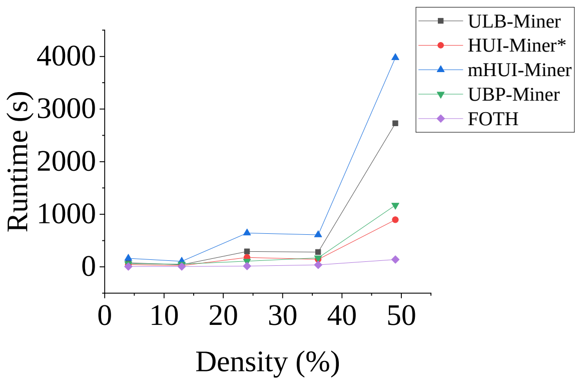
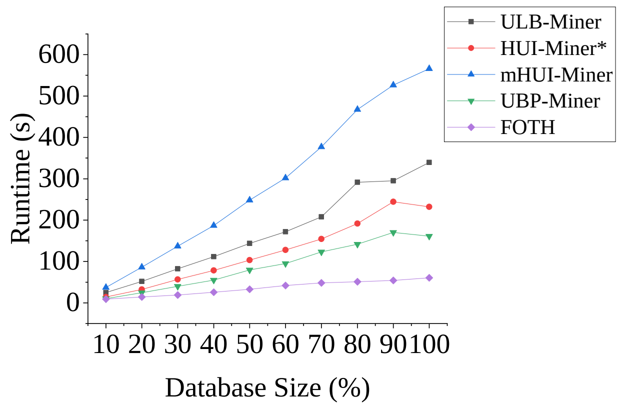


Figure 10: Runtime in different densities

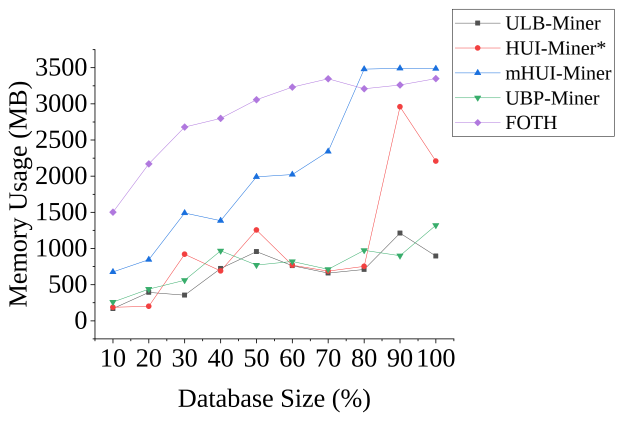
databases and thresholds. Figure 9 illustrates the percentage distribution of high utility itemsets of various sizes under a specific threshold. The thresholds are represented with different colors in that figure. It should be noted that there is only one high utility itemset that contains 65 items in the database *Retail*. For the convenience of plotting, it is ignored. Figure 9 shows that dense databases can generate numerous large itemsets. For instance, consider the results for *Connect* in Figure 9c. More than 90% of high utility itemsets of *Connect* contain at least 13 items whatever the *minutil* value is. These high utility itemsets are very large compared to the other databases. This is why FOTH achieved a remarkable runtime result in *Connect*. The sizes of high utility itemsets in both *Retail* and *Chainstore* are obviously smaller. And the database *Chainstore* mostly contains high utility 1-itemsets and high utility 2-itemsets. Therefore, according to the complexity analysis, FOTH consumed more time to mine these small itemsets.

To better understand the influence of density on performance, we have then added synthetic transactions to the densest database, *Chess*, to obtain databases with different densities. The *Chess* database is selected for this experiment because it has the largest density. The runtime results are shown in Figure 10. It can be observed that the runtime of FOTH grows slowly as the density increases. The larger the density, the better FOTH performed.

The last conclusion we want to explain is why FOTH is more stable in dense databases. As the *minutil* value increases, it is clear that the runtime of algorithms will decrease for all algorithms. Returning to Figure 9, except for *Connect*, the percentage of small size itemsets becomes larger and larger. This is the reason why the bars look like stairs for a particular small size in some databases. For example, when the size is 9, bars of different *minutil* values look like stairs. According to the complexity analysis, algorithms like UBP-Miner [22] are very good at mining small high utility itemsets. Therefore, the runtime of these algorithms decreases rapidly as the *minutil* increases. As for FOTH, the runtime decreases slowly because some large itemsets are no longer high utility itemsets. FOTH spends little time to mine these itemsets. Therefore, the runtime of FOTH is more stable.



(a) Runtime



(b) Memory consumption

Figure 11: Scalability comparison.

5.2.2. *Speedup of the join operation*

From Section 5.2.1, FOTH is proven to be the fastest for dense databases. To verify the effect of the joinless approach of FOTH, another experiment was conducted where the join operation of the compared algorithms was considered separately.

Table 8 shows the average speedup* for the six dense databases, where the threshold values are the same as in Figure 7. Compared with Table 6, we can easily observe that the non-joining approach improves performance. For example, in the database *Mushroom*, the ratio of UBP-Miner [22] is 5.6 in Table 6 and the average speedup* is 5.1 in Table 8. We can calculate the percentage 5.1/5.6 = 91.1%, which indicates how much the join operation costs to the whole runtime. In this way, the average percentages of the four algorithms from left to right in Table 8 are 84.2%, 89.6%, 49.8%, and 90.5%, respectively. Although the other operations of HUI-Miner* cost a lot of time, the non-joining optimization still reduces the runtime by nearly 50%. Notice that the speedup* is smaller than 1 except for the *Chicago* database. From Table 5, we can see that the AvgLen of *Chicago* is only 1.8. The sizes of itemsets in *Chicago* are relatively small compared to the other dense databases, as shown in Figure 9. Therefore, the join operation is performed less. That is also why FOTH did not provide a very large improvement on *Chicago*. Overall, the non-joining optimization of FOTH works well.

5.2.3. *Algorithm scalability in terms of runtime*

To compare the scalability of these algorithms, we transformed the largest database *Accidents* into ten smaller databases containing 10% to 100% of the transactions. The *minutil* value was fixed to 10%. Each database has at least 30,000 transactions.

Figure 11a shows the runtime of different algorithms as the database size changes. The standard deviations of ULB-Miner [27], HUI-Miner* [21], mHUI-Miner [24], UBP-Miner [22] and FOTH are 103.6, 76.9, 176.7 54.5 and 17.2 respectively. The Pearson correlation coefficient of FOTH is 0.99 and the p-value is $0.00 < 0.05$. Therefore, FOTH has a good linear scalability in runtime.

5.3. *Memory consumption analysis*

5.3.1. *Overall memeory consumption*

Figure 8 shows the peak memory consumption of the algorithms in different databases. FOTH has a higher memory consumption than the other algorithms, except for *Connect* and *Chicago*. Especially in *Mushroom*, FOTH consumes 5.6 times more memory than ULB-Miner [27]. FOTH consumes the least memory only in *Connect*. Therefore, we can conclude that FOTH consumes more memory in most databases.

The conclusion can be explained by the propagation process of FOTH. There are two nested for loops in Algorithm 1. Hence, the space complexity of propagating a vector of *IS* is $O(n^2)$ where n is the vector size. The example shown in Figure 2 also confirms this. However, algorithms using the join operation like Figure 6 construct only one new list structure each time. Therefore, FOTH attempts to generate numerous unnecessary vectors, leading to excessive memory consumption. But why does FOTH consume the least memory in *Connect*? We already know that high utility itemsets are very large in *Connect*. According to the analysis in Section 4.4, FOTH has a very low complexity for discovering a large itemset. Therefore, FOTH did not perform propagation too many times. On the contrary, the join operation is performed frequently. Then many unnecessary utility-lists are constructed. Although there are large itemsets in other databases, they are not large enough. Overall, the memory consumption of FOTH is deemed acceptable.

5.3.2. *Algorithm scalability in terms of memory consumption*

The same settings as the Section 5.2.3 were used to measure the memory scalability. The result is shown in Figure 11b. When the database size is smaller than 70%, FOTH consumes more memory than the other algorithms. The standard deviation of FOTH is 573.4 which is bigger than ULB-Miner [27] and UBP-Miner [22]. The Pearson coefficient of FOTH is 0.87 and the p-value is 0.00. It indicates that the memory overhead of FOTH grows linearly with the database's size. Overall, FOTH shows good scalability in terms of memory consumption.

5.4. Discussion

The experiments have been conducted to evaluate runtime and memory consumption. Based on the previous analysis and experimental results, we have two conclusions:

1) Because of the non-joining optimization, FOTH demonstrates superior performance compared to other list-based algorithms in dense databases in terms of runtime.
2) FOTH consumes more memory to mine HUIs unless most itemsets are very large.

FOTH can mine itemsets very quickly in dense databases and slowly in sparse databases. The large memory overhead is another weakness of FOTH. Although some analyses have been done before, there are still some questions that could be investigated further such as: why does the density affect the performance of FOTH so much? The complexity analysis presented in Section 4.4 is the key to answering this question. In sparse databases, transactions generally have few common items, unlike dense databases. However, HUIs usually contain items that are common to many transactions. Therefore, sparse databases generally contain smaller HUIs than dense databases. In Section 4.4, it was proved that FOTH will take more time to discover small HUIs (no matter from dense databases or sparse databases). Thus, because there are few small HUIs in dense databases, FOTH is generally faster than the compared algorithms, Both Figure 7 and Figure 9 corroborate this analysis. In the case of sparse databases, FOTH executes the propagation process more frequently to generate smaller itemsets, that is, IndexSets. Therefore, FOTH no longer performs better than the other algorithms. Numerous IndexSets impede the search process of FOTH and consume a considerable amount of memory.

6. Conclusion

This paper presented FOTH, a novel high utility itemset mining algorithm that introduces a new structure called the IndexSet. By propagating IndexSets, FOTH avoids the join operation. After analyzing the complexity of FOTH, it was concluded that FOTH can perform better than other list-based algorithms for mining large itemsets.

The performance of FOTH was evaluated on eight databases and compared with four state-of-the-art utility-list-based algorithms. For a specific *minutil* threshold, the experimental results show that FOTH outperforms the other algorithms on dense databases. As the *minutil* threshold is increased, FOTH progressively demands less time and memory to identify high utility itemsets. This efficiency gain is attributed to the reduction in the number of high utility itemsets as the *minutil* value increases.

In future work, we aim to enhance FOTH by introducing an additional pruning strategy to reduce the number of IndexSets generated during the propagation process. In this way, FOTH could be more efficient. Besides, the current IndexSet structure and the propagation procedure of FOTH are difficult to apply to mine condensed representations such as the Closed High Utility Itemsets[2]. The IndexSet structure would have to be adapted to store more information about itemsets such as the support. We will design new search procedures and data structures to mine condense representations in future work.

Acknowledgments

This work is supported by the National Natural Science Foundation of China [grant numbers 62272087]; Science and Technology Planning Project of Sichuan Province [grant number 2023YFGO161].

References

[1] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, SIGMOD Rec. 22 (2) (1993) 207–216.

[2] P. Fournier-Viger, J. Chun-Wei Lin, T. Truong-Chi, R. Nkambou, A survey of high utility itemset mining, High-utility pattern mining: Theory, algorithms and applications (2019) 1–45.

[3] H. Yao, H. J. Hamilton, C. J. Butz, A foundational approach to mining itemset utilities from databases, in: Proceedings of the 2004 SIAM International Conference on Data Mining, SIAM, 2004, pp. 482–486.

[4] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, Proceedings of the 1994 international conference on very large data bases (VLDB'94) 1215 (1994) 487–499.

[5] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, SIGMOD Rec. 29 (2) (2000) 1–12.

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

[7] J. S. Park, M.-S. Chen, P. S. Yu, An effective hash-based algorithm for mining association rules, SIGMOD Rec. 24 (2) (1995) 175–186.

[8] A. Savasere, E. Omiecinski, S. B. Navathe, An efficient algorithm for mining association rules in large databases, in: Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, p. 432–444.

[9] H. Toivonen, et al., Sampling large databases for association rules, in: Vldb, Vol. 96, 1996, pp. 134–145.

[10] R. C. Agarwal, C. C. Aggarwal, V. Prasad, A tree projection algorithm for generation of frequent item sets, J. Parallel Distrib. Comput. 61 (3) (2001) 350–371.

[11] Y. Liu, W.-k. Liao, A. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, in: T. B. Ho, D. Cheung, H. Liu (Eds.), Advances in Knowledge Discovery and Data Mining, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 689–695.

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

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

[14] U. Yun, H. Ryang, K. H. Ryu, High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates, Expert Systems with Applications 41 (8) (2014) 3861–3878.

[15] M. Liu, J. Qu, Mining high utility itemsets without candidate generation, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 55–64.

[16] J. Liu, K. Wang, B. C. M. Fung, Direct discovery of high utility itemsets without candidate generation, in: Proceedings of the 2012 IEEE 12th International Conference on Data Mining, ICDM '12, IEEE Computer Society, USA, 2012, p. 984–989.

[17] S. Zida, P. Fournier-Viger, J. C.-W. Lin, C.-W. Wu, V. S. Tseng, Efim: A highly efficient algorithm for high-utility itemset mining, in: G. Sidorov, S. N. Galicia-Haro (Eds.), Advances in Artificial Intelligence and Soft Computing, Springer International Publishing, Cham, 2015, pp. 530–546.

[18] P. Fournier-Viger, C.-W. Wu, S. Zida, V. S. Tseng, Fhm: Faster high-utility itemset mining using estimated utility co-occurrence pruning, in: T. Andreasen, H. Christiansen, J.-C. Cubero, Z. W. Raś (Eds.), Foundations of Intelligent Systems, Springer International Publishing, Cham, 2014, pp. 83–92.

[19] S. Krishnamoorthy, Hminer: Efficiently mining high utility itemsets, Expert Systems with Applications 90 (2017) 168–183.

[20] S. Krishnamoorthy, Pruning strategies for mining high utility itemsets, Expert Systems with Applications 42 (5) (2015) 2371–2381.

[21] J.-F. Qu, M. Liu, P. Fournier-Viger, Efficient algorithms for high utility itemset mining without candidate generation, High-Utility Pattern Mining: Theory, Algorithms and Applications (2019) 131–160.

[22] P. Wu, X. Niu, P. Fournier-Viger, C. Huang, B. Wang, Ubp-miner: An efficient bit based high utility itemset mining algorithm, Knowledge-Based Systems 248 (2022) 108865.

[23] H. Ryang, U. Yun, Indexed list-based high utility pattern mining with utility upper-bound reduction and pattern combination techniques, Knowl. Inf. Syst. 51 (2) (2017) 627–659.

[24] A. Y. Peng, Y. S. Koh, P. Riddle, mhuiminer: A fast high utility itemset mining algorithm for sparse datasets, in: J. Kim, K. Shim, L. Cao, J.-G. Lee, X. Lin, Y.-S. Moon (Eds.), Advances in Knowledge Discovery and Data Mining, Springer International Publishing, Cham, 2017, pp. 196–207.

[25] J.-F. Qu, P. Fournier-Viger, M. Liu, B. Hang, F. Wang, Mining high utility itemsets using extended chain structure and utility machine, Knowledge-Based Systems 208 (2020) 106457.

[26] J.-F. Qu, P. Fournier-Viger, M. Liu, B. Hang, C. Hu, Mining high utility itemsets using prefix trees and utility vectors, IEEE Transactions on Knowledge and Data Engineering (2023) 1–14.

[27] Q.-H. Duong, P. Fournier-Viger, H. Ramampiaro, K. NØrvåg, T.-L. Dam, Efficient high utility itemset mining using buffered utility-lists, Applied Intelligence 48 (7) (2018) 1859–1877.

[28] L. J. Guibas, R. Sedgewick, A dichromatic framework for balanced trees, in: 19th Annual Symposium on Foundations of Computer Science (sfcs 1978), 1978, pp. 8–21.

[29] M. S. Nawaz, P. Fournier-Viger, U. Yun, Y. Wu, W. Song, Mining high utility itemsets with hill climbing and simulated annealing, ACM Trans. Manage. Inf. Syst. 13 (1) (oct 2021).

[30] J. C.-W. Lin, L. Yang, P. Fournier-Viger, T.-P. Hong, M. Voznak, A binary pso approach to mine high-utility itemsets, Soft Comput. 21 (17) (2017) 5103–5121.

[31] W. Song, J. Li, C. Huang, Artificial fish swarm algorithm for mining high utility itemsets, in: Advances in Swarm Intelligence: 12th International Conference, ICSI 2021, Qingdao, China, July 17–21, 2021, Proceedings, Part II, Springer-Verlag, Berlin, Heidelberg, 2021, p. 407–419.

[32] C.-W. Lin, T.-P. Hong, W.-H. Lu, An effective tree structure for mining high utility itemsets, Expert Systems with Applications 38 (6) (2011) 7419–7424.

[33] T.-L. Dam, K. Li, P. Fournier-Viger, Q.-H. Duong, Cls-miner: Efficient and effective closed high-utility itemset mining, Front. Comput. Sci. 13 (2) (2019) 357–381.

[34] S. Pramanik, A. Goswami, Discovery of closed high utility itemsets using a fast nature-inspired ant colony algorithm, Applied Intelligence 52 (8) (2022) 8839–8855.

[35] A. Hidouri, S. Jabbour, B. Raddaoui, B. Ben Yaghlane, Mining closed high utility itemsets based on propositional satisfiability, Data & Knowledge Engineering 136 (2021) 101927.

[36] C.-W. Wu, P. Fournier-Viger, J.-Y. Gu, V. S. Tseng, Mining closed+ high utility itemsets without candidate generation, in: 2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI), 2015, pp. 187–194.

[37] A. Hidouri, S. Jabbour, I. O. Dlala, B. Raddaoui, On minimal and maximal high utility itemsets mining using propositional satisfiability, in: 2021 IEEE International Conference on Big Data (Big Data), 2021, pp. 622–628.

[38] H. Duong, T. Hoang, T. Tran, T. Truong, B. Le, P. Fournier-Viger, Efficient algorithms for mining closed and maximal high utility itemsets, Knowledge-Based Systems 257 (2022) 109921.

[39] P. Fournier-Viger, C.-W. Wu, V. S. Tseng, Novel concise representations of high utility itemsets using generator patterns, in: X. Luo, J. X. Yu, Z. Li (Eds.), Advanced Data Mining and Applications, Springer International Publishing, Cham, 2014, pp. 30–43.

[40] J. Sahoo, A. K. Das, A. Goswami, An efficient approach for mining association rules from high utility itemsets, Expert Systems with Applications 42 (13) (2015) 5754–5778.

[41] X. Han, X. Liu, J. Li, H. Gao, Efficient top-k high utility itemset mining on massive data, Information Sciences 557 (2021) 382–406.

[42] K. Singh, S. S. Singh, A. Kumar, B. Biswas, Tkeh: An efficient algorithm for mining top-k high utility itemsets, Applied Intelligence 49 (3) (2019) 1078–1097.

[43] S. Krishnamoorthy, Mining top-k high utility itemsets with effective threshold raising strategies, Expert Systems with Applications 117 (2019) 148–165.

[44] P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, H. T. Lam, The spmf open-source data mining library version 2, in: Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part III 16, Springer, 2016, pp. 36–40.

[45] Z. Zhang, J. Huang, J. Hao, J. Gong, H. Chen, Extracting relations of crime rates through fuzzy association rules mining, Applied Intelligence 50 (2) (2020) 448–467.

[46] G. J. Krishna, V. Ravi, High utility itemset mining using binary differential evolution: An application to customer segmentation, Expert Systems with Applications 181 (2021) 115122.

Table 9: The p-values of the pairwise t-test.

Database	*minutil*	UBP-Miner	mHUI-Miner	HUI-Miner*	ULB-Miner
Mushroom	2.0%	0.0000	0.0000	0.0000	0.0000
	2.5%	0.0000	0.0000	0.0000	0.0000
	3.0%	0.0000	0.0000	0.0000	0.0000
	3.5%	0.0000	0.0000	0.0000	0.0000
	4.0%	0.0000	0.0000	0.0000	0.0000
	4.5%	0.0000	0.0000	0.0000	0.0000
Chess	11%	0.0000	0.0000	0.0000	0.0000
	12%	0.0000	0.0000	0.0000	0.0000
	13%	0.0000	0.0000	0.0000	0.0000
	14%	0.0000	0.0000	0.0000	0.0000
	15%	0.0000	0.0000	0.0000	0.0000
	16%	0.0000	0.0000	0.0000	0.0000
Connect	30.0%	0.0000	0.0000	0.0000	0.0000
	30.2%	0.0000	0.0000	0.0000	0.0000
	30.4%	0.0000	0.0000	0.0000	0.0000
	30.6%	0.0000	0.0000	0.0000	0.0000
	30.8%	0.0000	0.0000	0.0000	0.0000
	31.0%	0.0000	0.0000	0.0000	0.0000
Accidents	5%	0.0000	0.0000	0.0000	0.0000
	6%	0.0000	0.0000	0.0000	0.0000
	7%	0.0000	0.0000	0.0000	0.0000
	8%	0.0000	0.0000	0.0000	0.0000
	9%	0.0000	0.0000	0.0000	0.0000
	10%	0.0000	0.0000	0.0000	0.0000
Chicago	0.05‰	0.0000	0.0002	**0.4517**	0.0000
	0.07‰	0.0000	0.0000	0.0000	0.0000
	0.09‰	0.0000	0.0000	0.0001	0.0000
	0.11‰	0.0000	0.000	0.0000	0.0000
	0.13‰	0.0000	0.0000	0.0000	0.0000
	0.15‰	0.0000	0.0000	0.0000	0.0000
c20d10k	1.0%	0.0000	0.0000	0.0000	0.0000
	1.2%	0.0000	0.0000	0.0000	0.0000
	1.4%	0.0000	0.0000	0.0000	0.0000
	1.6%	0.0000	0.0000	0.0000	0.0000
	1.8%	0.0000	0.0000	0.0000	0.0000
	2.0%	0.0000	0.0000	0.0000	0.0000
Retail	0.1‰	0.0000	0.0000	0.0000	0.0000
	0.2‰	0.0000	0.0000	0.0000	0.0000
	0.3‰	0.0000	0.0000	0.0000	0.0000
	0.4‰	0.0000	0.0000	0.0000	0.0000
	0.5‰	0.0000	0.0000	0.0000	0.0000
	0.6‰	0.0000	0.0000	0.0000	0.0000
Chainstore	0.5‰	0.0000	0.0000	0.0000	0.0000
	0.6‰	0.0000	0.0000	0.0000	0.0000
	0.7‰	0.0000	0.0000	0.0000	0.0000
	0.8‰	0.0000	0.0000	0.0000	0.0000
	0.9‰	0.0000	0.0000	0.0000	0.0000
	1.0‰	0.0000	0.0000	0.0000	0.0000