Fast Identification of High Utility Itemsets from Candidates

Jun-Feng Qu [1,*] **, Mengchi Liu** [2] **, Chunsheng Xin** [3] **and Zhongbo Wu** [1]

[1] School of Computer Engineering, Hubei University of Arts and Science, Xiangyang 441053, China; wzb_80@163.com

[2] School of Computer Science, Carleton University, Ottawa, ON K1S 5B6, Canada; mengchi@scs.carleton.ca

[3] Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA 23529, USA; cxin48@126.com

[*] Correspondence: qmxwt@163.com; Tel.: +86-071-0359-3152

Abstract: High utility itemsets (HUIs) are sets of items with high utility, like profit, in a database. Efficient mining of high utility itemsets is an important problem in the data mining area. Many mining algorithms adopt a two-phase framework. They first generate a set of candidate itemsets by roughly overestimating the utilities of all itemsets in a database, and subsequently compute the exact utility of each candidate to identify HUIs. Therefore, the major costs in these algorithms come from candidate generation and utility computation. Previous works mainly focus on how to reduce the number of candidates, without dedicating much attention to utility computation, to the best of our knowledge. However, we find that, for a mining task, the time of utility computation in two-phase algorithms dominates the whole running time of these algorithms. Therefore, it is important to optimize utility computation. In this paper, we first give a basic algorithm for HUI identification, the core of which is a utility computation procedure. Subsequently, a novel candidate tree structure is proposed for storing candidate itemsets, and a candidate tree-based algorithm is developed for fast HUI identification, in which there is an efficient utility computation procedure. Extensive experimental results show that the candidate tree-based algorithm outperforms the basic algorithm and the performance of two-phase algorithms, integrating the candidate tree algorithm as their second step, can be significantly improved.

Keywords: high utility itemset; utility computation

1. Introduction

In recent years, high utility itemset (HUI) mining [1] has became one of the most significant problems in the area of data mining. The problem derives from the frequent itemset mining problem [2], but the former considers the values of itemsets like profits, and is different from the latter that only takes the frequencies of itemsets into account. Efficient mining of high utility itemsets usually plays an important role in many real-life applications such as market analysis [3–7].

Many algorithms for high utility itemset mining adopt a two-phase frame [8–11], as shown in Figure 1. These algorithms first generate candidate itemsets, from which they subsequently identify high utility itemsets. Previous works pay much attention to reducing the number of candidate itemsets, which can result in performance improvement. However, these works neglect the identification process. A elaborate identification process plays an important role in performance improvement. This work focuses on the fast identification of high utility itemsets from candidates.
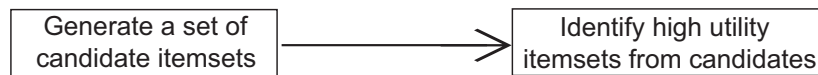
```
┌─────────────────────────┐           ┌─────────────────────────┐
│    Generate a set of    │           │    Identify high utility │
│    candidate itemsets   │ ────────> │   itemsets from candidates│
└─────────────────────────┘           └─────────────────────────┘
```

Figure 1. A two-phase frame for high utility itemset mining.

1.1. Problem Definition

Let $\mathcal{I} = \{i_1, i_2, i_3, \ldots, i_n\}$ be a set of items and *DB* be a transaction database. *DB* is composed of two tables: a utility table and a transaction table. Each item in $\mathcal{I}$ has a utility value in the utility table. Each transaction labeled with a Tid in the transaction table is a subset of $\mathcal{I}$, in which each item is associated with a count value. Tables 1 and 2 show a sample database.

Table 1. The utility table of a sample database.

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

Table 2. The transaction table of a sample database.

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

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

Definition 2. *The internal utility of item i in transaction T, denoted as iu(i, T), is the count value of i in T in the transaction table.*

Definition 3. *The utility of item i in transaction T, denoted as u(i, T), is the product of eu(i) and iu(i, T), namely u(i, T) = eu(i) × iu(i, T).*

For example, for the database in Table 1, eu(a) = 2, iu(a, T4) = 3, and u(a, T4) = eu(a) × iu(a, T4) = 2 × 3 = 6. An itemset is a subset of $\mathcal{I}$ and is called a *k*-itemset if it contains *k* items.

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

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

Definition 6. *An itemset is called* a high utility itemset *if its utility exceeds a user-specified minimum utility threshold denoted as "minutil".*

Note that if $X \nsubseteq T$, u(X, T) = 0. When a minutil is a percentage, high utility itemsets are those which utilities exceed the product of the minutil and the total utility of *DB*. For example, for the sample database, its total utility is u(T1, T1) + u(T2, T2) + u(T3, T3) + u(T4, T4) + u(T5, T5) + u(T6, T6) = 4 + 23 + 11 + 11 + 8 + 17 = 74, and u({a}) = u({a}, T1) + u({a}, T2) + u({a}, T4) + u({a}, T6) = 2 + 4 + 6 + 4 = 16, u({abd}) = u({abd}, T2) + u({abd}, T6) = 22 + 12 = 34, u({abcd}) = u({abcd}, T2) =23. If the minutil is equal to 40%, {abd} is a high utility itemset while {a} and {abcd} are not. Given a database and a minutil, the problem of high utility itemset mining is to find all high utility itemsets from the database.

1.2. Previous Solutions

After the formal introduction of the problem in [1] as above, a number of algorithms for high utility itemset mining have been proposed, such as TP [8], FSH [12], DCG [13], FUM [14], DCG+ [14], IHUPTWU [9], UP-Growth [10], and UP-Growth+ [11]. These algorithms employ a uniform two-phase framework as follows. Firstly, they generate a set of candidate itemsets from a mined database by roughly overestimating the utilities of all itemsets, and the set is a superset of the set of all high utility itemsets. Secondly, the exact utilities of all candidate itemsets are computed by a database scan, and thereby high utility itemsets are identified.

The two major costs in these algorithms are candidate generation and utility computation. It is obvious that the fewer candidate itemsets an algorithm generates, the lower the candidate generation and utility computation costs in the algorithm. Therefore, previous works put much effort into how to reduce the number of candidate itemsets. Recent two-phase algorithms such as UP-Growth+ have been able of efficiently reducing candidates. Table 3 shows the numbers of candidate itemsets generated by TP, FUM, UP-Growth, and UP-Growth+, given database *chain* and minutil 0.06%. The numbers of candidates in TP and FUM were taken from [14]. We implemented the last two algorithms and obtained the numbers of candidates in them. The database *chain* will be introduced in Section 5.

Table 3. Numbers of candidate itemsets in *Chain* (minutil = 0.06%).

Algorithm (Year)	TP (2005)	FUM (2007)	UP-Growth (2010)	UP-Growth+ (2012)
#Candidates	15,343	11,959	4485	4464

The running time of these algorithms mainly consists of the time for candidate generation in phase I and that for exact utility computation in phase II. Although the candidate generation time can be reduced significantly due to the decrease in the number of generated candidate itemsets, the exact utility computation time is still very large for a mining task. For example, when the minutil is 0.004%, 0.005%, and 0.006%, respectively, the running times of the two phases of the UP-Growth+ algorithm for database *chain* are depicted in Figure 2. It is very clear that the utility computation time dominates the whole running time of the algorithm.
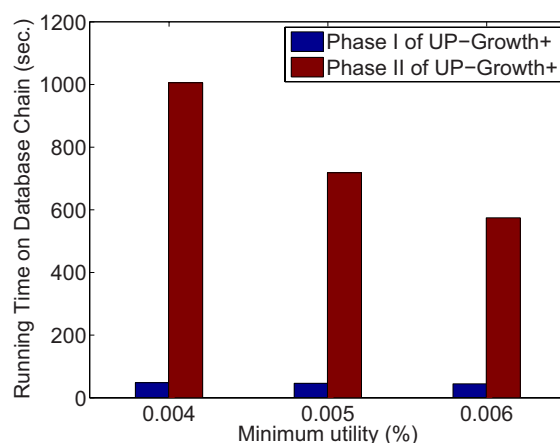

Figure 2. Running times of the two phases of UP-Growth+.

However, there is little effort to improve the performance of utility computation in previous works. To the best of our knowledge, a formal algorithm for exact utility computation is not even given in previous literature, although the algorithm should be simple.

1.3. Contributions

In this study, we focus on the fast identification of high utility itemsets from candidates, the core of which is the efficient utility computation for candidates. The main contributions of the paper are as follows.

- A basic algorithm for high utility itemset identification is formally presented.
- A novel structure called the candidate tree is proposed for storing candidate itemsets.
- A candidate tree-based algorithm is developed for the fast identification of high utility itemsets.
- Extensive experimental results that show the performance difference between the basic algorithm and the candidate tree-based algorithm are reported.

As shown in Figure 2, the running time of phase II dominates the total running time of a two-phase algorithm. The proposed structure and algorithms are devoted to the decrease in the time of phase II and thereby can result in performance improvement. The proposed structure and algorithms are all-purpose and can be integrated into any previous two-phase algorithm as its second step.

The rest of this paper is organized as follows. After the basic algorithm is introduced in Section 2, the candidate tree and related algorithm are proposed in Section 3 and analyzed in Section 4. Experimental results are reported in Section 5, and the paper ends with the conclusion of Section 6.

2. Basic Identification Algorithm

In this section, we show a basic identification algorithm (BIA) and discuss its core procedure.

2.1. Pseudo-Code of the BIA

Algorithm 1 shows the pseudo-code of the BIA.

Algorithm 1: Basic Identification Algorithm

Input: C is a set of candidate itemsets;
      DB is a transaction database;
      minutil is a minimum utility threshold.
Output: all high utility itemsets

1 **foreach** candidate itemset $c \in C$ **do**
2 | $utility[c] = 0$;
3 **end**
4 **foreach** transaction $t \in DB$ **do**
5 | **foreach** candidate itemset $c \in C$ **do**
6 | | $utility[c] = utility[c] + \mathbf{u}(c, t)$;
7 | **end**
8 **end**
9 **foreach** candidate itemset $c \in C$ **do**
10 | **if** $utility[c] \geq minutil$ **then**
11 | | output c;
12 | **end**
13 **end**

Firstly, a vector *utility* indexed by the names of candidates is initialized, and *utility*[c] stores the utility of candidate c. Subsequently, for each transaction, the algorithm accumulates the utility of each candidate in the vector. At last, the algorithm outputs those candidates, the utilities of which exceed the minimum utility threshold.

When both a set of candidates *C* and a database *DB* can be stored in memory, or when *C* can be stored in memory but *DB* cannot, the BIA works well. If *DB* can be stored in memory but *C* cannot, it is better to exchange the two loops in line 4 and line 5 for reducing the I/O cost.

2.2. Utility Computation

In the BIA, the core procedure is the computation of the utility of itemset c in transaction t in line 6, which is listed in Procedure 2.

Procedure 2: u(c, t)

Input: c is a candidate itemset;
       t is a transaction.
Output: the utility of c in t

1   $i = 1$;
2   $j = 1$;
3   $util = 0$;
4   **while** $i \leq length(c)$ **and** $j \leq length(t)$ **do**
5   **while** $j \leq length(t)$ **and** $c[i] > t[j]$ **do**
6         $j = j + 1$;
7   **end**
8   **if** $j > length(t)$ **or** $c[i] < t[j]$ **then**
9   **break**;
10   **else**   // $c[i]==t[j]$
11         $util = util + u(t[j], t)$;
12         $i = i + 1$;
13         $j = j + 1$;
14   **end**
15 **end**
16 **if** $i > length(c)$ **then**   // $c \subseteq t$
17   **return** $util$;
18 **else**
19   **return** 0;
20 **end**

In the procedure, *length*(c) and *length*(t) are the numbers of items in c and t, c[i] denotes the *i*th item in c and t[j] denotes the *j*th item in t. For each c[i], a search for it is performed in t. If there is such t[j] that t[j] is equal to c[i], the utility of the item in t, namely u(t[j], t), is added to variable *util* storing the accumulated utility for c. If the condition in line 16 is met, which means that t contains c, *util* is returned. The procedure is actually a two-way comparison procedure, in which the atom operations are item comparisons (lines 5 and 8) and utility accumulations (line 11).

The procedure is based on the assumption that items in both c and t are ordered. The items in a candidate itemset can be sorted before it is stored. The items in a transaction are generally ordered, and otherwise they can also be sorted after the transaction is loaded in memory. For u(t[j], t) in line 11 in the procedure, we can compute it once and employ many. For example, the sample database can be transformed into the view in Table 4.

Table 4. A database view.

Tid	Item	Util.	Item	Util.	Item	Util.	Item	Util.	Item	Util.
T1	a	2	c	1	f	1				
T2	a	4	b	3	c	1	d	15		
T3	b	3	c	2	d	5	e	1		
T4	a	6	b	3	f	2				
T5	b	6	c	2						
T6	a	4	b	3	d	5	e	1	g	4

3. Identifying HUIs by a Candidate-Tree

In the process of computing the utilities of candidate itemsets, two main operations are comparisons and accumulations. For example, $u(\{ab\}, T2) = 0 + u(a, T2) + u(b, T2) = 0 + 4 + 3 = 7$, and there are 2 comparisons and 2 accumulations. Suppose $\{ab\}$, $\{abc\}$, $\{abd\}$, and $\{abcd\}$ are candidates, and then $u(\{ab\}, T2) = 0 + u(a, T2) + u(b, T2)$, $u(\{abc\}, T2) = 0 + u(a, T2) + u(b, T2) + u(c, T2)$, $u(\{abd\}, T2) = 0 + u(a, T2) + u(b, T2) + u(d, T2)$, and $u(\{abcd\}, T2) = 0 + u(a, T2) + u(b, T2) + u(c, T2) + u(d, T2)$. It is obvious that there are many repeated comparisons and accumulations in these utility computations.

3.1. Candidate-Tree

To speed utility computation up, repeated comparisons and accumulations should be avoided. First of all, we can store all candidate itemsets in a candidate tree. A candidate tree is a modified prefix-tree [15], in which itemsets containing the same prefix share a common path. For example, the candidate tree in Figure 3 can represent itemsets $\{ab\}$, $\{abc\}$, $\{abd\}$, and $\{abcd\}$. Besides the pointers for maintaining the tree structure, each node in a candidate tree contains an *item* and an *util*. A node represents an itemset composed of the items in the path from the node to the root. The util of a node is used to store the utility of the itemset represented by the node. For example, the node numbered 5 in Figure 3 represents itemset $\{abd\}$. In a candidate tree, not all nodes represent candidate itemsets.
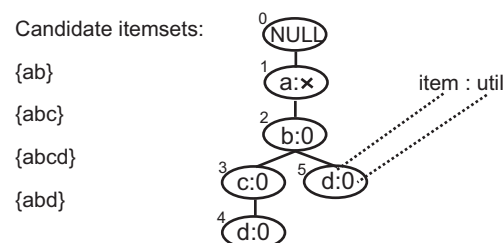
Candidate itemsets:

{ab}

{abc}

{abcd}

{abd}

Figure 3. Candidate itemsets and a candidate tree.

Definition 7. *In a candidate tree, a node is called* a count node *if it represents a candidate itemset.*

For the candidate tree in Figure 3, the nodes numbered 2, 3, 4, and 5 are count nodes, and the node numbered 1 is not.

The method of constructing a candidate tree is similar to the method of constructing a prefix-tree [15]. In the implementation of a candidate tree, the util of a count node is initialized with 0, and that of a node that is not a count node is initialized with -1. In this way, all count nodes of a candidate tree are marked during the candidate tree construction.

3.2. Fast HUI Identification

After a candidate tree is constructed, a fast identification algorithm (FIA) can efficiently compute the utilities of all candidates stored in the tree and subsequently identify high utility itemsets. The FIA is shown in Algorithm 3, the core procedure of which is shown in Procedure 4.

Algorithm 3: Fast Identification Algorithm

   Input: *root* is the root node of a candidate tree;
          DB is a transaction database;
          minutil is a minimum utility threshold.
   Output: all high utility itemsets
1 **foreach** transaction $t \in DB$ **do**
2 | **ComputeUtility**(t, 1, *root*, 0);
3 **end**
4 **IdentifyHUI**(*root*, $\varnothing$, *minutil*);

Procedure 4: ComputeUtility(t, k, n, *utility*)

   Input: *t* is a transaction;
          k indicates the position of an item in *t*;
          n is a node in the candidate tree;
          utility stores the sum of the utilities of the items contained in all *n*'s ancestor nodes in *t*.
1 **while** $k \leq length(t)$ **and** $t[k] < n.item$ **do**
2 | $k = k + 1$;
3 **end**
4 **if** $k > length(t)$ **or** $t[k] \neq n.item$ **then**
5 | **return** k;
6 **else**   // $t[k]{=}{=}n.item$
7 | $utility = utility + \mathbf{u}(t[k], t)$;
8 | **if** *n* is a count node **then**
9 | | $n.util = n.util + utility$;
10 | **end**
11 | $next = k + 1$;
12 | **foreach** child node *c* of *n* **do**
13 | | $next = $ **ComputeUtility**(t, *next*, *c*, *utility*);
14 | **end**
15 | **return** k+1;
16 **end**

Like Procedure 2, items in transactions and itemsets are considered as ordered in Procedure 4. In the procedure, *n.item* and *n.util* denote the item and util contained in *n*. Suppose node *n* represents itemset *X*, and then parameter *utility* stores u(*X-n.item*, *t*). Firstly, the procedure searches $t[k]$ ($k \leq length[t]$) for *n.item*. If *t* does not contain *n.item*, the subtree rooted at *n* is no longer checked (line 5). Otherwise, *utility* is updated and is added to *n.util* if *n* is a count node (line 9), and subsequently all child nodes of *n* are recursively processed. Parameter *k* keeps track of the position of an item in *t* before which each item in *t* is contained in an ancestor node of *n* and is no longer compared with *n.item*. After the subtree rooted at *n* is recursively processed, the utils of the count nodes in the subtree representing the itemsets contained in *t* are updated. To facilitate the understanding of Procedure 4, Figure 4 demonstrates the procedure when T2 in Table 4 and the candidate tree in Figure 3 are processed.

After all transactions in *DB* are processed, all high utility itemsets can be identified by a candidate tree traversal as shown in Procedure 5.
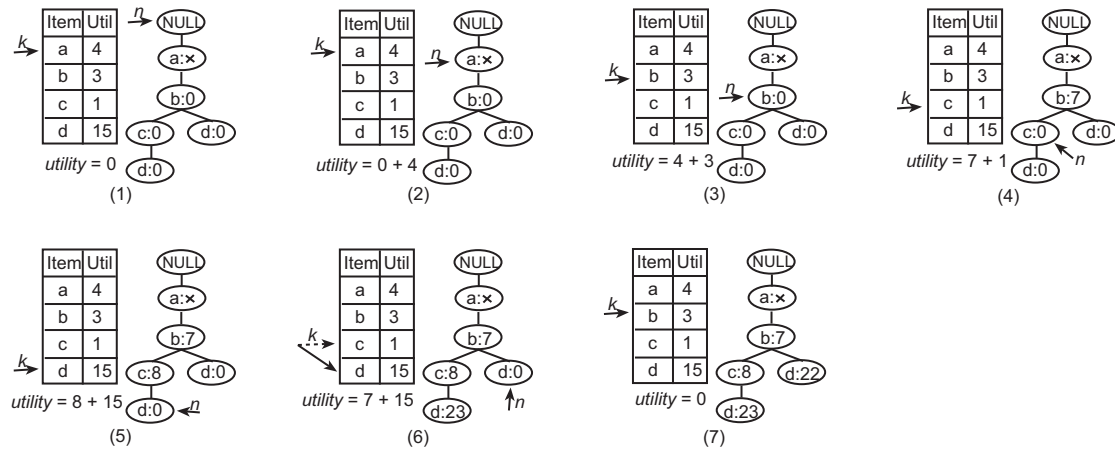
Figure 4. Utility computation on a candidate tree.

Procedure 5: IdentifyHUI(*n*, *X*, *minutil*)

Input: *n* is a node in the candidate tree;

   X is a prefix itemset;

   minutil is the minimum utility threshold.

1 *X* = *X* ∪ *n.item*;

2 **if** *n* is a count node **and** *n.util*≥*minutil* **then**

3 | output *X*;

4 **end**

5 **foreach** child node *c* of *n* **do**

6 | **IdentifyHUI**(*c*, *X* *minutil*);

7 **end**

---

4. Complexity Analysis

The main operations in the BIA and FIA are item comparisons and utility accumulations. Since items in a *k*-itemset *X* and a transaction *T* containing *m* items are ordered, for computing u(*X*, *T*), the comparison number denoted as *CN* holds in Properties 1 and 2, and the accumulation number denoted as *AN* holds in Properties 3 and 4.

Property 1. *If T contains X, then* $k \leq CN \leq m$.

Property 2. *If T does not contain X, then* $1 \leq CN \leq max(k, m)$, *and* $max(k, m)$ *denotes the larger between k and m.*

Property 3. *If T contains X, then* $AN = k$.

Property 4. *If T does not contain X, then* $0 \leq AN \leq (k-1)$.

Suppose there are a transaction with *m* items and *n* candidates that contain $s_1, s_2, s_3, \ldots, s_n$ items, respectively. The candidates have the same prefix itemset with *s* items ($s \leq s_i$, $1 \leq i \leq n$). To compute the utilities of the candidates in the transaction, the numbers of comparisons and accumulations performed in the BIA and FIA, on condition that all the candidates are or are not contained in the transaction, are listed in Table 5.

Table 5. Numbers of comparisons and accumulations. FIA: fast identification algorithm; BIA: basic identification algorithm.

Comparison Number	Least	Most
BIA (contained)	$\sum_{i=1}^{n} s_i$	$m + m + m + \ldots + m = m \times n$
FIA (contained)	$s + (s_1 - s) + \ldots + (s_n - s)$ $= (\sum_{i=1}^{n} s_i) - (n-1) \times s$	$m + (m - s) + (m - s) + \ldots + + (m - s)$ $= m \times (n + 1) - n \times s$
BIA (not contained)	$1 + 1 + 1 + \ldots + 1 = n$	$\sum_{i=1}^{n} max(m, s_i)$
FIA (not contained)	1	$max(m, s) + \sum_{i=1}^{n} max(m - s, s_i - s)$

Accumulation Number	Least	Most
BIA (contained)	$\sum_{i=1}^{n} s_i$	$\sum_{i=1}^{n} s_i$
FIA(contained)	$s + (s_1 - s) + (s_2 - s) + \ldots (s_n - s)$ $= (\sum_{i=1}^{n} s_i) - (n-1) \times s$	$s + (s_1 - s) + (s_2 - s) + \ldots (s_n - s)$ $= (\sum_{i=1}^{n} s_i) - (n-1) \times s$
BIA (not contained)	0	$(s_1 - 1) + (s_2 - 1) + \ldots (s_n - 1)$ $= (\sum_{i=1}^{n} s_i) - n$
FIA (not contained)	0	$s + (s_1 - s - 1) + \ldots (s_n - s - 1)$ $= (\sum_{i=1}^{n} s_i) - n - (n-1) \times s$

For example, when the utility of the candidate with s_i items is computed, for the BIA, the number of comparisons is s_i at least or m at most according Property 1, if the transaction contains the candidate. Then, the total number of comparisons for all the candidates is $(s_1 + s_2 + s_3 + \ldots + s_n)$ at least or $(m + m + \ldots + m = m \times n)$ at most, if the transaction contains these candidates. When these candidates are stored in a candidate tree, the n candidates can be considered as $(n + 1)$ candidates that contain s, $(s_1 - s), (s_2 - s), \ldots, (s_n - s)$ items respectively. Therefore, for the FIA, the total number of comparisons for the $(n + 1)$ candidates is $s + (s_1 - s) + (s_2 - s) + \ldots + (s_n - s) = (s_1 + s_2 + s_3 + \ldots + s_n) - (n - 1) \times s$ at least or $m + (m - s) + (m - s) + \ldots + (m - s) = m \times (n + 1) - n \times s$ at most. The remaining numbers in the figure can be analyzed similarly.

In the worst case, the complexities of the BIA and FIA are all O($m \times n$) with respect to comparisons, but compared with the BIA the number of comparisons in FIA factually decreases by $n \times s$, which is a large factor, especially for a large s. It is also observed that the number of accumulations in the FIA decreases by about $n \times s$ in the worst case, compared with that in the BIA.

5. Experiments

In this section, the BIA is compared with FIA.

We first implemented a famous algorithm UP-Growth+ [11] in C++. UP-Growth+ is a standard two-phase algorithm, and it first generates a set of candidate itemsets and subsequently computes the exact utilities of candidates to identify high utility itemsets. However, the utility computation of UP-Growth+ is not discussed in detail in [11]. Therefore, we integrated the BIA and FIA into UP-Growth+ as its second step, respectively. In the following, *BIA-UP-Growth+* denotes the combination of UP-Growth+ with the BIA, and *FIA-UP-Growth+* denotes the combination of UP-Growth+ with the FIA.

Eight databases were used in our experiments. The database *chain* was downloaded from NU-MineBench 2.0 [16], and the other databases were downloaded from the FIMI Repository [17]. Databases *accidents*, *chess*, *kosarak*, *mushroom*, and *retail* derived from the real world, and synthetic databases *T10I4D100K* and *T40I10D100K* were generated by the IBM Quest Synthetic Data Generation Code. Except for *chain*, the other databases do not provide the external utility and internal utility for each item, and thus we generated the utility and count values of each item as the settings in previous works [9–11]. The statistical information about these databases is shown in Table 6, including the size on disk, the number of transactions, the number of distinct items, the average number of items in a transaction, and the maximal number of items in the longest transaction(s). The experiments were

performed on a machine with a 2.8 GHz Intel Core i5 CPU, 4 GB of physical memory, and a 32-bit Linux operation system.

Table 6. Statistical information about databases.

Database	Size (MB)	#Trans	#Items	AvgLen	MaxLen
accidents	56.89	340,183	468	33.8	51
chain	60.63	1,112,949	46,086	7.3	170
chess	0.56	3196	75	37	37
kosarak	47.55	990,002	41,270	8.1	2498
mushroom	0.92	8124	119	23	23
retail	5.79	88,162	16,470	10.3	76
T10I4D100K	5.86	100,000	870	10.1	29
T40I10D100K	22.69	100,000	942	39.6	77

5.1. Running Time for Phase II

For each experimental database, it was transformed into a physical view in memory as in Table 4, and thereby u(*t*[*j*], *t*) in both Procedures 2 and 4 was directly available. After a set of candidate itemsets or a candidate tree was generated in memory, the utility computation time of the two algorithms was recorded, as depicted in Figure 5. We varied the minimum utility in the experiments. The lower the minimum utility is, the more high utility itemsets an algorithm generates, and thus the greater the running time is.
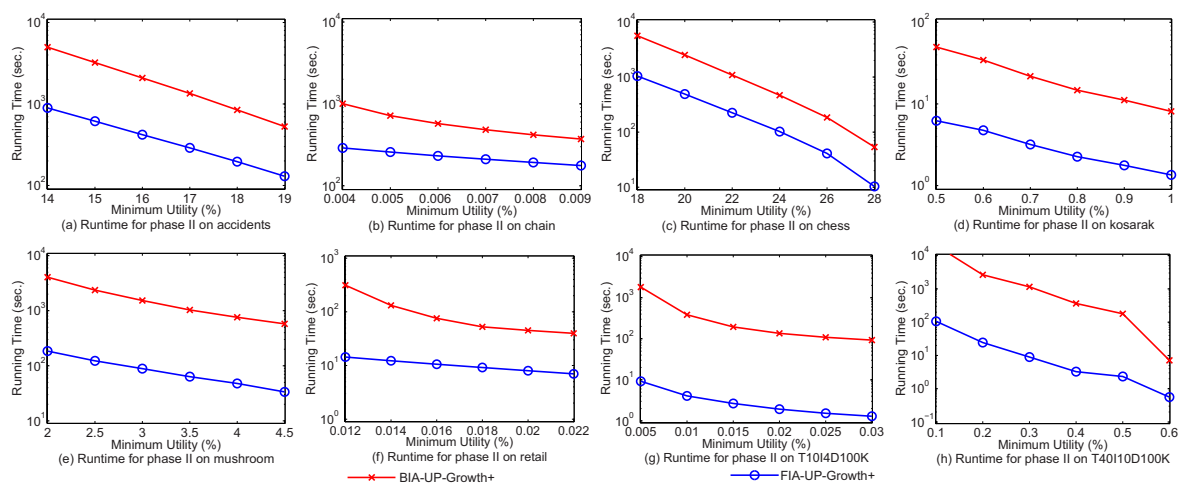

Figure 5. Performance comparison.

It can be observed that FIA-UP-Growth+ always outperforms BIA-UP-Growth+. For the databases *accidents*, *chain*, and *chess*, in Figure 5a–c, FIA-UP-Growth+ is several times faster than BIA-UP-Growth+. For the databases *kosarak*, *mushroom*, and *retail*, in Figure 5d–f, FIA-UP-Growth+ is about an order of magnitude faster than BIA-UP-Growth+. For databases *T10I4D100K* and *T40I10D100K*, in Figure 5g,h, FIA-UP-Growth+ is two orders of magnitude faster than BIA-UP-Growth+.

5.2. Running Time for Phase I

In phase I, the difference between BIA-UP-Growth+ and FIA-UP-Growth+ is that the former directly stores each generated candidate itemset in a memory pool, while the latter inserts each candidate itemset into a candidate tree immediately after generating it. Therefore, in theory, BIA-UP-Growth+ is faster than FIA-UP-Growth+ in the phase. The third column in Table 7 lists the first phase time of the two algorithms running on the eight databases for the lowest minutils in our

experiments, and in this case, the algorithms generate the largest numbers of candidate itemsets and high utility itemsets.

Table 7. Experimental results of BIA-UP-Growth+ (denoted as Basic) and FIA-UP-Growth+ (denoted as Fast).

Database/Minutil	Algorithm	Phase I (s)	Phase II (s)	Memory (KB)	#Candidates	#HUIs
accidents/14%	Basic	3.79	4981.55	8512	276,392	950
	Fast	3.83	895.26	5440		
chain/0.004%	Basic	48.03	1005.56	832	72,503	18,480
	Fast	53.20	290.07	1440		
chess/18%	Basic	13.32	5628.51	1,387,808	31,670,469	34,870
	Fast	18.21	1042.78	623,008		
kosarak/0.5%	Basic	10.91	49.76	64	3931	183
	Fast	10.93	6.22	96		
mushroom/2%	Basic	5.99	4065.73	675,328	16,681,768	3,583,596
	Fast	8.24	185.00	331,552		
retail/0.012%	Basic	0.83	313.11	4768	163,650	23,505
	Fast	1.17	14.40	4000		
T10I4DX/0.005%	Basic	1.51	1818.70	20,416	1,007,230	313,509
	Fast	3.48	9.23	20,736		
T40I10DX/0.1%	Basic	62.94	>>10,000	342,208	8,608,882	2,054,784
	Fast	48.81	106.25	179,104		

Even though there is a very large number of candidates, the time for constructing a candidate tree is small. For example, when the minutil is 18% for database *chess*, the first phase runtime of FIA-UP-Growth+ is 18.21 seconds and that of BIA-UP-Growth+ is 13.32 seconds, and then the time of constructing the candidate tree can be considered as 4.89 (=18.21 − 13.32) s. It is interesting that the first phase runtime of FIA-UP-Growth+ is even shorter than that of BIA-UP-Growth+ for database *T40I10D100K*, when the minutil is 0.1%. We believe the reason is that, for the mining task, the time of constructing the candidate tree is relatively short, while FIA-UP-Growth+ holds better data locality than BIA-UP-Growth+ due to the smaller memory consumption.

5.3. Memory Consumption

FIA-UP-Growth+ generates candidate itemsets as BIA-UP-Growth+ does [11], and thus they consume the same amount of memory for candidate generation. On the other hand, there is no considerable memory consumption in their second phases, namely in the FIA and BIA. Therefore, we paid attention to the memory consumption of the two algorithms for storing candidate itemsets, as shown in the fifth column in Table 7.

Since a candidate tree is a compact data structure [15], the size of a candidate tree-storing candidate itemsets is smaller than the size of a memory pool storing them, if the number of the candidate itemsets is large enough and thus there are many shared paths. For example, for databases *chess*, *mushroom*, and *T40I10D100K*, in Table 7, FIA-UP-Growth+ only consumes half the amount of memory BIA-UP-Growth+ does. However, a candidate tree also stores the tree structure information, namely pointers for linking nodes, and thus FIA-UP-Growth+ consumes more memory than BIA-UP-Growth+ if there is a small number of candidate itemsets.

5.4. Discussion

FIA-UP-Growth+ significantly outperforms BIA-UP-Growth+ in our experiments. The reasons are as follows.

Firstly, a high utility itemset mining algorithm generally generates a very large number of candidate itemsets, as shown in the sixth column in Table 7, and therefore there are numerous

comparisons and accumulations when computing their utilities. The numbers of comparisons and accumulations can be reduced efficiently if utility computation is performed on a candidate tree.

Secondly, using a candidate tree, the utility computation for the candidates sharing the same prefix but not contained in a transaction can be terminated once and for all. For example, for the candidate tree in Figure 3, when T1 in Table 4 is processed, the utility computation for the four candidates can be terminated immediately after two comparisons according to the FIA. If these candidates are stored in a memory pool, there are eight comparisons according to the BIA. Actually, for many mining tasks, the number of high utility itemsets is far less than the number of candidate itemsets, as shown in the last column in Table 7. Therefore, for a transaction, there should be a considerable number of candidates that are not contained in it.

Thirdly, if the number of candidate itemsets is so large that there are many shared paths, a candidate tree storing them occupies less memory than a memory pool storing them, and thereby the FIA can gain better data locality than the BIA.

Fourthly, although the first phase runtime of the algorithm integrating FIA is increased due to the candidate tree construction, the increase in the first phase runtime of the algorithm can be balanced by the decrease in the second phase runtime of the algorithm.

6. Conclusions

In this paper, we addressed the problem of identifying high utility itemsets from candidates. The high utility itemset identification is an indispensable part of most mining algorithms, but it is not discussed in these algorithms in detail. As a supplement to previous works, we first gave a basic identification algorithm, i.e., the BIA. Subsequently, we proposed a novel data structure called candidate tree for storing candidate itemsets and developed a candidate tree-based algorithm, i.e., the FIA, for the fast identification of high utility itemsets. The main operations in the BIA and FIA are comparisons and accumulations. For an identification task, the FIA performs fewer comparisons and has less accumulations than the BIA. Extensive experimental results show that (1) the time for high utility itemset identification dominates the whole running time for a mining algorithm; and (2) the FIA significantly outperforms the BIA in various databases.

It should be noted that FIA works well if a candidate tree can be completely in memory. However, this study does not consider the case that a tree is too large to be completely stored in memory. We plan to study the fast identification of high utility itemsets from candidates in disk in a future study.

Author Contributions: J.-F.Q. designed and developed the model. J.-F.Q., M.L. and C.X. wrote the manuscript. Z.W. carried out the experimental tests.

References

1. Yao, H.; Hamilton, H.J.; Butz, C.J. A Foundational Approach to Mining Itemset Utilities from Databases. In Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, FL, USA, 22–24 April 2004.
2. Agrawal, R.; Imieliński, T.; Swami, A. Mining association rules between sets of items in large databases. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, 25–28 May 1993; pp. 207–216.
3. Krishnamoorthy, S. Efficient Mining of High Utility Itemsets with Multiple Minimum Utility Thresholds. *Eng. Appl. Artif. Intell.* **2018**, *69*, 112–126. [CrossRef]
4. Zhang, L.; Fu, G.; Cheng, F.; Qiu, J.; Su, Y. A Multi-Objective Evolutionary Approach for Mining Frequent and High Utility Itemsets. *Appl. Soft Comput.* **2018**, *62*, 974–986. [CrossRef]
5. Mai, T.; Vo, B.; Nguyen, L.T.T. A Lattice-Based Approach for Mining High Utility Association Rules. *Inf. Sci.* **2017**, *399*, 81–97. [CrossRef]

6. Wu, J.M.-T.; Zhan, J.; Li, J.C.-W. An ACO-Based Approach to Mine High-Utility Itemsets. *Knowl-Based Syst.* **2017**, *116*, 102–113. [CrossRef]

7. Guo, Z.; Yue, X.; Yang, H.; Liu, K.; Liu, X. Enhancing social emotional optimization algorithm using local search. *Soft Comput.* **2017**, *21*, 7393–7404. [CrossRef]

8. Liu, Y.; Liao, W.; Choudhary, A.N. A Two-Phase Algorithm for Fast Discovery of High Utility Itemsets. In Proceedings of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2005, Hanoi, Vietnam, 18–20 May 2005; pp. 689–695.

9. Ahmed, C.F.; Tanbeer, S.K.; Jeong, B.; Lee, Y. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Trans. Knowl. Data Eng.* **2009**, *21*, 1708–1721. [CrossRef]

10. Tseng, V.S.; Wu, C.-W.; Shie, B.-E.; Yu, P.S. Up growth: An efficient algorithm for high utility itemset mining. In Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 25–28 July 2010; pp. 253–262.

11. Tseng, V.S.; Shie, B.-E.; Wu, C.-W.; Yu, P.S. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Trans. Knowl. Data Eng.* **2012**, *25*, 1772–1786. [CrossRef]

12. Li, Y.-C.; Yeh, J.-S.; Chang, C.-C. A fast algorithm for mining share-frequent itemsets. In Proceedings of the 7th Asia-Pacific Web Conference on Web Technologies Research and Development—APWeb 2005, Shanghai, China, 29 March–1 April 2005; pp. 417–428.

13. Li, Y.-C.; Yeh, J.-S.; Chang, C.-C. Direct candidates generation: A novel algorithm for discovering complete share-frequent itemsets. In Proceedings of the International Conference on Fuzzy Systems and Knowledge Discovery, Changsha, China, 27–29 August 2005; pp. 551–560.

14. Li, Y.-C.; Yeh, J.-S.; Chang, C.-C. Isolated Items Discarding Strategy for Discovering High Utility Itemsets. *Data Knowl. Eng.* **2008**, *64*, 198–217. [CrossRef]

15. Han, J.; Pei, J.; Yin, Y.; Mao, R. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.* **2004**, *8*, 53–87. [CrossRef]

16. NU-MineBench: A Data Mining Benchmark Suite. Available online: http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html (accessed on 8 April 2018).

17. Frequent Itemset Mining Dataset Repository. Available online: http://fimi.ua.ac.be/ (accessed on 8 April 2018).