Efficient Incremental High Utility Itemset Mining

Philippe Fournier-Viger
University of Moncton
Moncton, NB, Canada
philippe.fv@gmail.com

Jerry Chun-Wei Lin
Harbin Institute of technology,
Shenzhen Grad. School
Shenzhen, China
jerrylin@ieee.org

Ted Gueniche
University of Moncton
Moncton, NB, Canada
ted.gueniche@gmail.com

Prashant Barhate
Infosys Ltd.
Jaegaon, India
prashantbarhate1990@gmail.com

ABSTRACT

High-utility itemset mining (HUIM) in transaction databases is an important data mining task with wide applications. However, most HUIM algorithms assume the unrealistic assumption that databases are static. To address this issue, algorithms have been designed to maintain high-utility itemsets in dynamic databases. However, these incremental algorithms still remain very costly in terms of execution time. In this paper, we address this issue by proposing an algorithm named EIHI (Efficient Incremental High-utility Itemset miner), which introduces several novel ideas to efficiently maintain high-utility itemsets in dynamic databases.

CCS Concepts

• Information systems → Data mining;

Keywords

High-utility itemset mining; incremental pattern mining

1. INTRODUCTION

High-Utility Itemset Mining (HUIM) [2, 3, 8, 14] is an emerging data mining task that extends Frequent Itemset Mining (FIM) [1] by considering the case where items can appear more than once in each transaction and where each item has a weight (e.g. unit profit). Therefore, it can be used to discover itemsets having a high-utility (e.g. high profit), that is High-Utility Itemsets (HUIs). Several algorithms have been proposed to mine high-utility itemsets in transaction databases [2, 3, 5, 6, 8, 14]. However, most of them assume that databases are static. A few algorithms have been designed for maintaining high-utility itemsets in dynamic databases [2, 12]. However, these incremental algorithms remain very costly in terms of execution time. In this paper, we address this issue by proposing an algorithm named EIHI (Efficient Incremental High-utility Itemset miner), which introduces several novel ideas to efficiently maintain high-utility itemsets in dynamic databases.

2. PROBLEM DEFINITION

Let $I$ be a finite set of items (symbols). An itemset $X$ is a finite set of items such that $X \subseteq I$. A transaction database is a set of transactions $D = \{T_1, T_2, ..., T_n\}$ such that for each transaction $T_c$, $T_c \subseteq I$ and $T_c$ has a unique identifier $c$ called its TID (Transaction ID). Each item $i \in I$ is associated with a positive number $p(i)$, called its external utility (e.g. unit profit). Every item $i$ appearing in a transaction $T_c$ has a positive number $q(i, T_c)$, called its internal utility (e.g. purchase quantity).

For example, consider the database in Table 1, which will be used as the running example in the rest of this paper. Transaction $T_2$ indicates that items $a, c, e$ and $g$ appear in this transaction with an internal utility of respectively 2, 6, 2 and 5. Table 2 indicates that external utilities of these items are respectively 5, 1, 3 and 1.

The utility of an item $i$ in a transaction $T_c$ is denoted as $u(i, T_c)$ and defined as $p(i) \times q(i, T_c)$ if $i \in T_c$. The utility of an itemset $X$ in a transaction $T_c$ is denoted as $u(X, T_c)$ and defined as $u(X, T_c) = \sum_{i \in X} u(i, T_c)$ if $X \subseteq T_c$. The utility of an itemset $X$ in a database $D$ is denoted as $u(X)$ and defined as $u(X) = \sum_{T_c \in D} u(X, T_c)$, where $g(X)$ is the set of transactions containing $X$. For example, $u(\{a, c\}, T_2) = 16$ and $u(\{a, c\}) = 28$. An itemset $X$ is a high-utility itemset if its utility $u(X)$ is no less than a user-specified minimum utility threshold $\minutil$ given by the user (i.e. $u(X) \geq \minutil$). Otherwise, $X$ is a low-utility itemset. The problem of high-utility itemset mining in a database $D$ is to discover the set $H$ of all high-utility itemsets [3, 8, 14].

For example, if $\minutil = 30$, the high-utility itemsets are $\{b, d\}, \{a, c, e\}, \{b, c, d\}, \{b, c, e\}, \{b, d, e\}, \{a, b, c, d, e, f\}$ with respectively a utility of 30, 31, 34, 31, 36, 40 and 30.

Definition 1. The problem of maintaining high-utility itemsets in a dynamic database is defined as follows [12]. Let be a database $D$. A database $D'$ is an update of database $D$ if $D' = D \cup N$, where $N$ is a non empty set of transactions.
such that $N \subseteq D$. Let $\text{minutil}$, $D$ and $H$, respectively be a user-specified minimum utility threshold, a database, and the set of high-utility itemsets found in $D$. Let $D'$ be an update of database $D$. The problem of incremental high-utility itemset mining is to find $H'$ the set of high-utility itemsets in $D'$, given $\text{minutil}$, $D$ and $H$.

For example, consider $D$ to be the running example and $\text{minutil} = 30$. Let $D'$ be the database $D$ that is updated by inserting transaction $T_6 = (a, 2)(b, 5)(f, 10)$. In the updated database $D'$, the high-utility itemsets are $\{b, d\}$, $\{a, c, e\}$, $\{b, c, d\}$, $\{b, d, e\}$, $\{a, b, c, d, e, f\}$ and $\{d\}$, respectively have a utility of 50, 31, 34, 31, 40, 30 and 30. In this problem, the example of incremental high-utility itemset mining is to find that a new high-utility itemset $\{d\}$ has been updated to 50.

### Table 1: A Transaction Database $D$

<table>
<thead>
<tr>
<th>TID</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>(a, 1)(c, 1)(d, 1)</td>
</tr>
<tr>
<td>$T_2$</td>
<td>(a, 2)(c, 6)(e, 2)(g, 5)</td>
</tr>
<tr>
<td>$T_3$</td>
<td>(a, 1)(b, 2)(e, 1)(d, 6)(c, 1)(f, 5)</td>
</tr>
<tr>
<td>$T_4$</td>
<td>(b, 4)(c, 3)(d, 3)(e, 1)</td>
</tr>
<tr>
<td>$T_5$</td>
<td>(b, 2)(c, 2)(e, 1)(g, 2)</td>
</tr>
</tbody>
</table>

### Table 2: External Utility Values

<table>
<thead>
<tr>
<th>Item</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3. RELATED WORK

HUIM is harder than FIM since the utility is not monotonic or anti-monotonic [2, 14], i.e. the utility of an itemset may be lower, equal or higher than the utility of its subsets. Thus, strategies used in FIM to prune the search space based on the anti-monotonicity of the frequency cannot be applied to the utility measure to discover high-utility itemsets. Several HUIM algorithms circumvent this problem by overestimating the utility of itemsets using the TWU measure [2, 14], which is anti-monotonic. The transaction utility of a transaction $T_i$ is defined as $TU(T_i) = \sum_{x \in T_i} u(x, T_i)$. The transaction-weighted utilization (TWU) of an itemset $X$ is defined as $TWU(X) = \sum_{T_i \subseteq X} TU(T_i)$. For example, consider item $a$. $TWU(a) = TU(T_1) + TU(T_2) + TU(T_3) = 8 + 27 + 30 = 65$. The following property of the TWU is used to prune the search space.

**Property 1** (Pruning using the TWU). For any itemset $X$, if $TWU(X) \lt \text{minutil}$, then $X$ is a low-utility itemset as well as its all supersets.

Two phase algorithms such as IHUP [2], UP-Growth+ [14], PB [7], and BAHUI [13] utilize property 1 to prune the search space. Those algorithms however suffers from the problem of generating a large amount of candidates. Recently, algorithms that mine high-utility itemsets using a single phase and avoid the problem of candidate generation were proposed, such as HUI-Miner [8] and FHM [3], which were shown to outperform state-of-the-art two phase algorithms. FHM is to our knowledge the current best algorithm for HUIM. In HUI-Miner and FHM, each itemset is associated with a structure named utility-list [3, 8]. Utility-lists allow calculating the utility of an itemset by making join operations with utility-lists of smaller itemsets.

Utility-lists are defined as follows. Let $\succ$ be a total order on items from $I$, and $X$ be an itemset. The remaining utility of $X$ in a transaction $T_i$ is defined as $re(X, T_i) = \sum_{x \in T_i \land x \notin X} u(i, T_i)$. The utility-list of an itemset $X$ in a database $D$ is a set of tuples such that there is a tuple $(c, iutil, rutil)$ for each transaction $T_i$ containing $X$. The $iutil$ and $rutil$ elements of a tuple respectively are the utility of $X$ in $T_i$ ($u(X, T_i)$) and the remaining utility of $X$ in $T_i$ ($re(X, T_i)$). For example, assume the lexicographical order. The utility-list of $\{a, c\}$ is $\{(T_2, 16, 5), (T_3, 8, 5)\}$.

To discover high-utility itemsets, HUI-Miner and FHM perform a database scan to create utility-lists of patterns containing single items. Then, larger patterns are obtained by joining utility-lists of smaller patterns [8]. Pruning the search space and calculating the utility of an itemset is respectively done using the following properties.

**Property 2** (Pruning using utility-lists). Let $X$ be an itemset. Let the extensions of $X$ be the itemsets that can be obtained by appending an item $i$ to $X$ such that $i \succ x, \forall x \in X$. The remaining utility upper-bound of $X$ is defined as $reu(X) = u(X) + re(X)$, and can be computed by summing the $iutil$ and $rutil$ values in the utility-list of $X$. If $reu(X) \lt \text{minutil}$, then $X$ is a low-utility itemset as well as all its extensions [8].

**Property 3** (Utility using utility-list). The utility of an itemset $X$ can be calculated as the sum of the $iutil$ values in its utility-list [8].

To maintain high-utility itemsets in databases where new transactions are inserted, IHUP [2] was proposed. Then, the FUP-HUI-INS [10] and the PRE-HUI-INS [11] were proposed to update the discovered high-utility itemsets based on the FUP concept and pre-large concept, respectively. A limitation of these algorithms is that they are two phase algorithms, and thus generate a large amount of candidates. To avoid this problem, a one-phase algorithm named HUI-LIST-INS was proposed [12] that extends the FHM algorithm. It is to our knowledge the state-of-the-art algorithm for incremental high utility itemset mining.

### 4. THE EIHI ALGORITHM

In this section, we present our proposal, the EIHI algorithm. We first describe the main procedure, which is inspired by the FHM [3] algorithm. This procedure is a batch algorithm. We then explain how it is adapted to maintain HUIMs in an updated database. We call this new algorithm the EIHI algorithm. The main procedure (Algorithm 1) takes as input a transaction database with utility values and the $\text{minutil}$ threshold. The algorithm scans the database to calculate the TWU of each item. Then, it identifies the set $I^*$ of all items having a TWU no less than $\text{minutil}$. The TWU values of items are then used to establish a total order $\succ$ on items, which is the order of ascending TWU values [3, 8]. A second database scan is then performed. Items in transactions are reordered according to $\succ$, the utility-list of each item $i \in I^*$ is built and a structure named EUCS is built [3]. This structure stores the TWU of all pairs of
items \( \{a, b\} \) such that \( u(\{a, b\}) \neq 0 \). Then, the depth-first search exploration of itemsets starts by calling the recursive procedure \( \text{Search} \) with the empty itemset \( \emptyset \), the set of single items \( I^* \), \( \text{minutil} \) and the EUCS.

**Algorithm 1: The EIHII Algorithm**

**input**: \( D \): a transaction database, \( \text{minutil} \): a user-specified threshold  
**output**: the set of high-utility itemsets

1. Scan \( D \) to calculate the TWU of single items;  
2. \( I^* \leftarrow \) each item \( i \) such that \( \text{TWU}(i) \geq \text{minutil} \);  
3. Let \( \succ \) be the total order of TWU ascending values on \( I^* \);  
4. Scan \( D \) to build the utility-list of each item \( i \in I^* \) and build the EUCS structure;  
5. \( \text{Search} (\emptyset, I^*, \text{minutil}, \text{EUCS}) \);

The \( \text{Search} \) procedure (Algorithm 2) takes as input (1) an itemset \( P \), (2) extensions of \( P \) having the form \( Pz \) meaning that \( Pz \) was previously obtained by appending an item \( z \) to \( P \), (3) \( \text{minutil} \) and (4) the EUCS. The search procedure operates as follows. For each extension \( Px \) of \( P \), if the sum of the \( iutil \) values of the utility-list of \( Px \) is no less than \( \text{minutil} \), then \( Px \) is a high-utility itemset and it is output (cf. property 3). Then, if the sum of \( iutil \) and \( rutil \) values in the utility-list of \( Px \) are no less than \( \text{minutil} \), it means that extensions of \( Px \) should be explored (cf. property 2). This is performed by merging \( Px \) with all extensions \( Py \) of \( P \) such that \( y \succ x \) and \( \text{TWU}(\{x, y\}) \geq \text{minutil} \), to form extensions of the form \( Pxy \) containing \( |Px| + 1 \) items. The utility-list of \( Pxy \) is then constructed as in FHM by calling the \( \text{Construct} \) procedure (cf. Algorithm 3) to join the utility-lists of \( P \), \( Px \) and \( Py \) [3]. Then, a recursive call to the \( \text{Search} \) procedure with \( Px \) is done to calculate its utility and explore its extension(s). The \( \text{Search} \) procedure starts from single items and recursively explores the search space to discover all high-utility itemsets [3].

We next explain how the algorithm is modified to maintain high-utility itemsets in a dynamic database. We first introduce a few novel and very important ideas.

**The HUI-tree structure.** The first important idea is the following. Let be an update \( D' \) of a database \( D \), such that \( D' = D \cup N \), where \( N \) is a non empty set of transactions. It can be observed that the set of high-utility itemsets \( H' \) found in \( D' \) is always a superset of the set of high-utility itemsets \( H \) found in \( D \). Moreover, it can be observed that the utilities of itemsets in \( H \) may only increase or stay the same in \( H' \) as a result of the insertion of the new transactions. For example, in the running example, the utility of itemset \( \{b, d\} \) is updated from 30 to 50 when transaction \( T_6 \) is added. It is thus important to have an efficient mechanism for storing itemsets found in \( H \) to be able to quickly update their utility. The solution adopted in EIHII to this problem is to store all itemsets from \( H \) in a trie-like structure that we call the HUI-trie. This structure is a trie, where each node represents an item and each itemset is represented by a path starting from the root and ending by an inner node or a leaf. Moreover, each node representing the last item of an itemset is annotated with the utility of that itemset. For example, Fig. 1 show the HUI-trie structure contructed with the HUIs found in the database of Table 1, that is \{b, a\}, \{a, c\}, \{b, c, d\}, \{b, c\}, \{b, d\}, \{b, c, d, e\}, \{a, b, c, d, e, f\}, having utilities of respectively 30, 31, 34, 31, 36, 40 and 30.

Inserting an itemset \( P \) in the HUI-trie structure is very efficient, as it requires to traverse/create \( |P| + 1 \) nodes in the tree (starting from the root). Searching for an itemset \( P \) in the HUI-trie structure is also very efficient. It requires to only traverse one path in the tree. Furthermore, to increase efficiency of searching in the HUI-trie, the list of child nodes of each node is sorted according to the total order \( \succ \). This allows to perform a binary search at each node when looking for the child node corresponding to a given item.

![Figure 1: The HUI-trie structure](image-url)
Efficiently mining the updated database. The second important idea in the EIHI algorithm is about how to efficiently mine itemsets in $D'$. For the original database $D$, one only needs to apply Algorithm 1 to find the set of high-utility itemsets $H$. However, how to find the set $H'$ in $D'$, given $D$ and $H$? A naive solution would be to apply Algorithm 1 on database $D'$. However, this would be inefficient since it would mine high-utility itemsets from scratch. Moreover, another reason is that numerous itemsets appearing in $D$ may not appear in $D'$. This lead to the following important property, which was not used in HUI-LIST-INS.

**Property 4.** Let be an itemset $x \in H$ appearing in $D$ but not appearing in $N$, where $D' = D \cup N$. The utility of $x$ in $D'$ is the same as the utility of $x$ in $D$. Thus, if $x$ is a high-utility itemset in $D$, it is also a high-utility itemset in $D'$. Similarly, if $x$ is a low-utility itemset in $D$, it is also a low-utility itemset in $D'$.

Based on property 4, to design an efficient algorithm to find $H'$, one should avoid exploring itemsets not appearing in $N$. We next explain how this idea is implemented in EIHI. The first modification is to the TWU calculation of single items in Line 1 of Algorithm 1. When the algorithm is run on $D'$, the TWU of items in $D$ has been already calculated when the algorithm was run on $D$. By keeping this information, it is only necessary to scan transactions in $N$ to update the TWU of all items and obtain the TWU of items in $D'$.

The second modification is to how the algorithm determine the set of item $I'$ to be used for generating itemsets in Line 2 of Algorithm 1. The set $I'$ is redefined by adding the condition that items from $I'$ must appear in $N$. In other words, items that are appearing in $D$ but not in $N$ are not used to generate itemsets (based on property 4).

The third modification is to how the $\succ$ order is defined in Line 3 of Algorithm 1. As previously explained, when the algorithm is run on $D$, a total order $\succ$ is established on items from $D$ based on the order of increasing TWU of items. But in the database $D'$ some new items may appear and the TWU of items may change, which may lead to a different total order. To ensure that the pruning properties still hold, it is necessary to use a total order for $D'$ that is compatible with the total order used for $D$. Thus, the algorithm is modified to keep the total order from $D$ into memory and only insert the new items from $D'$ in this total order.

The fourth modification is to how utility-lists are constructed for each item in $I'$ in Line 4 of Algorithm 1. A naive approach would be to build the utility-list of each item in $I'$ for $D'$ from scratch, which would require to scan all transactions in $D'$. However, since the algorithm was previously run on $D$, the utility-lists of items in $D$ have already been built. Therefore, it is only necessary to add elements in utility-lists for transactions in $N$. In our implementation, we thus assign to each item two utility-lists: (1) a utility-list for $D$ and (2) a utility-list for $N$. Using these two utility-lists is equivalent to using a single utility-list for $D'$. To build the utility-lists for $N$, only transactions from $N$ are read.

The fifth modification is to how the EUCS structure is built in Line 4 of Algorithm 1. A naive approach for building the EUCS would be to construct it for $D'$ by scanning the whole database $D'$. A better approach is to update the EUCS by scanning only the transactions $N$ and adding the transaction utilities of items appearing in these transactions to the TWU of pairs of items stored in the EUCS to obtain the updated TWU of each pair of items.

The sixth modification is with respect to how the algorithm determines if an itemset is high-utility. Algorithm 2 is modified as follows. Let $ul_D(Px)$ denotes the utility-list of an itemset $Px$ in transactions $D$. Line 2 is modified by checking if the sum of iutils values in $ul_D(Px)$ plus the sum of the iutils values in $ul_N(Px)$ is greater than minutil. This condition is to ensure that only itemsets that are high-utility in $D'$ are output.

The seventh modification is Line 5 of the Search procedure. This pruning condition is modified as if the sum of iutils and rutils values in $ul_D(Px)$ plus the sum of the iutils and rutils values in $ul_N(Px)$ is greater than minutil. This is to ensure than an itemset and its extensions are only pruned if the sum of iutil and rutil values in $D'$ is no less than minutil (according to property 2). This pruning condition is formalized as follows.

**Property 5 (Pruning condition 1).** For any itemset $Px$, if the sum of iutils and rutils values in $ul_D(Px)$ plus the sum of the iutils and rutils values in $ul_N(Px)$ is less than minutil, then $Px$ and its extensions are low-utility itemsets.

The eight modification is a new pruning condition. In Algorithm 1, the utility-list of an itemset $Pxy$ is created by combining the utility-lists of itemsets $P$, $Px$, and $Py$ using the Construct procedure. In EIHI, each itemset has utility-list for $D$ and a utility-list for $N$. Thus, the Construct procedure has to be called twice to construct the utility-lists of $Pxy$ for $D$ and $N$. A novel pruning condition introduced in EIHI is the following.

**Property 6 (Pruning condition 2).** For any itemset $Pxy$, if the utility-list of $Pxy$ in $N$ is empty ($ul_N(Pxy)$), then the itemset $Pxy$ and all its extension do not need to be explored.
Rationale. If the utility-list of $Pxy$ is empty, it means that $Pxy$ does not appear in $N$. By property 4, it is thus not necessary to consider $Pxy$. Since $Pxy$ does not appear in $N$, it follows that extensions of $Pxy$ also do not appear in $N$. Thus, any extension of $Pxy$ also do not need to be explored.

The above pruning condition is incorporated in Line 11 of the Search procedure. If the utility-list of $Pxy$ is empty, then $Pxy$ is not added to the set $Extensio nsOfPx$. Thanks to this condition, only itemsets that appears in $N$ will be considered by the search procedure. A further optimization is to construct the utility-list of $Pxy$ in $N$ first before constructing the utility-list of $Pxy$ in $D$. If the utility-list of $Pxy$ in $N$ is empty, the utility-list of $Pxy$ in $D$ does not need to be constructed.

The ninth important modification is to how a high-utility itemset $Px$ is output in Line 3 of the Search procedure. For each high-utility itemset $Px$ that is found, the EIHI algorithm first search if this itemset is in the proposed HUI-trie structure. If yes, it means that $Px$ was previously found when the algorithm was applied to $D$. In that case, the utility of $Px$ is updated in the HUI-trie. Otherwise, it means that $Px$ is a new high-utility itemset that was low-utility in $D$. Thus, $Px$ is inserted in the HUI-trie with its calculated utility. When the algorithm terminates, all updated HUIs are in the HUI-trie.

The tenth modification is that at the end of the algorithm, the utility-lists in $D$ and in $N$ are merged together to form a single utility-list for $D'$. This is to prepare the algorithm for the next execution where the database $D'$ will be updated.

Finally, the last modification is to incorporate the LA-prune optimization, which was proposed in HUP-Miner [9], in the Construct procedure. This optimization is compatible with any utility-list based algorithm and is not modified here. Therefore, it is not described and the interested reader can refer to the paper about HUP-Miner for more details.

Based on the above explanations and properties, it is easy to see that the algorithm is correct and complete for the problem of incremental high-utility itemset mining.

5. EXPERIMENTAL EVALUATION

Experiments were performed on a computer with a third generation 64 bit Core i5 processor running Windows 7 and 8 GB of RAM. We compared the performance of EIHI with the state-of-the-art algorithm HUI-LIST-INS. Experiments were carried on four datasets. Let $NT$, $NI$ and $AL$ respectively denote the number of transactions, number of items and average transaction length of a dataset.

The datasets are retail ($NT = 88, 162, NI = 16, 470, AL = 10.3$), foodmart ($NT = 4, 141, NI = 1, 559, AL = 4.4$), chess ($NT = 3, 196, NI = 75, AL = 35$), and mushroom ($NT = 8, 124, NI = 120, AL = 23$). The foodmart dataset contains real utility values. For the other datasets, external utilities and internal utilities were respectively generated in the [1, 1000] and [1, 5] intervals as in previous work [2, 3, 8, 14].

Algorithms were run on each dataset, while decreasing the $minutil$ threshold until a clear winner was observed. Furthermore, for each dataset, we run each algorithm three times for each $minutil$ value to test a different number of updates to the database. The notation $Y-x$ indicates that the algorithm $Y$ was run $x$ times on a given dataset by splitting the dataset into $x$ parts. For example, for a dataset of 1,000 transactions, EIHI-5 indicates the performance of EIHI when it is applied five times (to transactions 1 to 200, 201 to 400, 401 to 600, 601 to 800 and 801 to 1,000). The comparison of execution times is shown in Fig. 2.

It can be observed that EIHI is always considerably faster than HUI-LIST-INS on all datasets for the same number of updates and $minutil$ values (up to 220 times faster than HUI-LIST-INS). The largest gap in terms of execution time between EIHI and HUI-LIST-INS is for the real shopping datasets (retail and foodmart). For dense datasets, the gap is smaller because items appear in almost every transactions and thus the pruning condition of EIHI is less effective.

It can also be observed that the gap in terms of execution time between the proposed EIHI algorithm and HUI-LIST-INS increases considerably as the number of update increases. For example, on foodmart with 5, 25 and 50 updates, the EIHI algorithm is respectively 16, 124 and 220 times faster than HUI-LIST-INS.

In terms of memory usage, EIHI has very similar memory usage to HUI-LIST-INS. The reason is that both algorithm are utility-list based algorithms based on the FHM algorithm and that the new pruning conditions introduced in EIHI are designed to reduce execution time rather than memory usage. For retail, foodmart, chess, mushroom and T10I6D100K, the maximum memory usage of the algorithms is respectively about 500 MB, 200 MB, 100, 500 and 800 MB.

6. CONCLUSION

We have presented EIHI, a novel algorithm for maintaining high-utility itemsets in dynamic databases. It introduces numerous ideas to more efficiently maintain high-utility itemsets in dynamic databases. Experimental results show that EIHI is up to two orders of magnitude faster than the state-of-art HUI-LIST-INS algorithm and is more scalable with respect to the number of updates. The source code of the EIHI and HUI-LIST-INS algorithms as well as all datasets used in this paper are available as part of the SPMF open-source data mining library http://goo.gl/rIKub.

For future work, we plan to explore other problem related to the maintenance of patterns in dynamic databases. In particular, we are interested in mining high-utility sequential pattern [15] and sequential rules [16].

7. ACKNOWLEDGEMENTS

This research was partially supported by the Natural Scientific Research Innovation Foundation in Harbin Institute of Technology under grant HIT.NSRIF.2014100, by the Tencent Project under grant CCF-TencentRAGR20140114, and by the National Natural Science Foundation of China (NSFC) under grant No.61503092.

8. REFERENCES

Figure 2: Comparison of execution times