EFIM: A Highly Efficient Algorithm for High-Utility Itemset Mining

Souleymane Zida[1], Philippe Fournier-Viger[1], Jerry Chun-Wei Lin[2], Cheng-Wei Wu[3], Vincent S. Tseng[3]

[1] Dept. of Computer Science, University of Moncton, Canada
[2] School of Computer Science and Technology, Harbin Institute of Technology Shenzhen Graduate School, China
[3] Dept. of Computer Science, National Chiao Tung University, Taiwan
esz2233@umoncton.ca, philippe.fournier-viger@umoncton.ca, jerrylin@ieee.org, silvemoonfox@gmail.com, vtseng@cs.nctu.edu.tw

Abstract. High-utility itemset mining (HUIM) is an important data mining task with wide applications. In this paper, we propose a novel algorithm named EFIM (EFficient high-utility Itemset Mining), which introduces several new ideas to more efficiently discovers high-utility itemsets both in terms of execution time and memory. EFIM relies on two upper-bounds named *sub-tree utility* and *local utility* to more effectively prune the search space. It also introduces a novel array-based utility counting technique named *Fast Utility Counting* to calculate these upper-bounds in linear time and space. Moreover, to reduce the cost of database scans, EFIM proposes efficient database projection and transaction merging techniques. An extensive experimental study on various datasets shows that EFIM is in general two to three orders of magnitude faster and consumes up to eight times less memory than the state-of-art algorithms d^{2}HUP, HUI-Miner, HUP-Miner, FHM and UP-Growth+.

Keywords: high-utility mining, itemset mining, pattern mining.

1 Introduction

Frequent Itemset Mining (FIM) [1] is a popular data mining task that is essential to a wide range of applications. Given a transaction database, FIM consists of discovering frequent itemsets. i.e. groups of items (itemsets) appearing frequently in transactions [1, 14]. However, an important limitation of FIM is that it assumes that each item cannot appear more than once in each transaction and that all items have the same importance (weight, unit profit or value). To address these issues, the problem of *High-Utility Itemset Mining* (HUIM) has been defined [2, 6–9, 11, 12]. As opposed to FIM [1, 14], HUIM considers 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*. HUIM has a wide range of applications [2, 7, 12]. The problem of HUIM is more difficult than the problem

of FIM because the utility of an itemset is neither monotonic or anti-monotonic (a HUI may have a superset or subset with lower, equal or higher utility) [2, 7, 12]. Thus techniques to prune the search space developed in FIM based on the anti-monotonicity of the support cannot be directly applied to HUIM.

Several algorithms have been proposed for HUIM. A popular approach to HUIM is to discover high-utility itemsets in two phases. This approach is adopted by algorithms such as PB [6], Two-Phase [9], BAHUI [11], UP-Growth and UP-Growth+ [12]. However, the two-phase model suffers from the problem of generating a huge amount of candidates and repeatedly scanning the database. Recently, to avoid the problem of candidate generation, the HUI-Miner [7] and d^{2}HUP [8] algorithms were proposed to mine high-utility itemsets directly using a single phase. Then, improved versions of HUI-Miner named HUP-Miner [8] and FHM [2] were proposed, and are to our knowledge the current best algorithms for HUIM. However, despite all these research efforts, the task of high-utility itemset mining remains very computationally expensive.

In this paper, we address this need for more efficient HUIM algorithms by proposing a one-phase algorithm named EFIM (EFficient high-utility Itemset Mining), which introduces several novel ideas to greatly decrease the time and memory required for HUIM. First, we propose two new techniques to reduce the cost of database scans. EFIM performs database projection and merges transactions that are identical in each projected database using a linear time and space implementation. Both operations reduce the size of the database as larger itemsets are explored. Second, we propose two new upper-bounds on the utility of itemsets named *sub-tree utility* and *local utility* to more effectively prune the search space. Third, we introduce a novel array-based utility counting technique named *Fast Utility Counting* (FAC) to calculate these upper-bounds in linear time and space.

We conducted an extensive experimental study to compare the performance of EFIM with the state-of-the-art algorithms d^{2}HUP, HUI-Miner, HUP-Miner, FHM and UP-Growth+ on various datasets. Results show that EFIM is in general two to three orders of magnitude faster than these algorithms, consumes up to eight times less memory.

The rest of this paper is organized as follows. Sections 2, 3, 4, 5 and 6 respectively presents the problem of HUIM, the related work, the EFIM algorithm, the experimental evaluation and the conclusion.

2 Problem Statement

The problem of high-utility itemset mining is defined as follows. 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 multiset 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. It contains five transactions ($T_1, T_2..., T_5$). 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 the external utility of these items are respectively 5, 1, 3 and 1.

Table 1. A transaction database

TID	Transaction
T_1	$(a,1)(c,1)(d,1)$
T_2	$(a,2)(c,6)(e,2)(g,5)$
T_3	$(a,1)(b,2)(c,1)(d,6)(e,1)(f,5)$
T_4	$(b,4)(c,3)(d,3)(e,1)$
T_5	$(b,2)(c,2)(e,1)(g,2)$

Table 2. External utility values

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

Definition 1 (Utility of an item/itemset). 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* is denoted as $u(X)$ and defined as $u(X) = \sum_{T_c \in g(X)} u(X, T_c)$, where $g(X)$ is the set of transactions containing X.

For example, the utility of item a in T_2 is $u(a, T_2) = 5 \times 2 = 10$. The utility of the itemset $\{a, c\}$ in T_2 is $u(\{a, c\}, T_2) = u(a, T_2) + u(c, T_2) = 5 \times 2 + 1 \times 6 = 16$. Furthermore, the utility of the itemset $\{a, c\}$ is $u(\{a, c\}) = u(\{a, c\}, T_1) + u(\{a, c\}, T_2) + u(\{a, c\}, T_3) = u(a, T_1) + u(c, T_1) + u(a, T_2) + u(c, T_2) + u(a, T_3) + u(c, T_3) = 5 + 1 + 10 + 6 + 5 + 1 = 28$.

Definition 2 (Problem definition). An itemset X is a *high-utility itemset* if $u(X) \geq minutil$. Otherwise, X is a *low-utility itemset*. The *problem of high-utility itemset mining* is to discover all high-utility itemsets.

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

3 Related Work

HUIM is harder than FIM since the utility measure is not monotonic or anti-monotonic [9, 12], 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 *Transaction-Weighted Utilization* (TWU) measure [6, 9, 11, 12], which is anti-monotonic, and defined as follows.

Definition 3 (Transaction weighted utilization (TWU)). The *transaction utility* of a transaction T_c is defined as $TU(T_c) = \sum_{x \in T_c} u(x, T_c)$. The TWU of an itemset X is defined as $TWU(X) = \sum_{T_c \in g(X)} TU(T_c)$.

Example 1. The TU of transactions T_1, T_2, T_3, T_4 and T_5 for our running example are respectively 8, 27, 30, 20 and 11. $TWU(a) = TU(T_1) + TU(T_2) + TU(T_3) = 8 + 27 + 30 = 65$.

For any itemset X, it can be shown that $TWU(X) \geq u(Y) \forall Y \supseteq X$ (the TWU of X is an upper-bound on the utility of X and its supersets) [9]. The following properties of the TWU is used to prune the search space.

Property 1 (Pruning search space using TWU). For any itemset X, if $TWU(X) < minutil$, then X is a low-utility itemset as well as all its supersets [9].

Algorithms such as PB [6], Two-Phase [9], BAHUI [11], UP-Growth and UP-Growth+ [12] utilize Property 1 to prune the search space. They operate in two phases. In the first phase, they identify candidate high-utility itemsets by calculating their TWUs. In the second phase, they scan the database to calculate the exact utility of all candidates to filter low-utility itemsets. UP-Growth is one of the fastest two-phase algorithm. Recent two-phase algorithms such as PB and BAHUI only provide a small speed improvement.

Recently, algorithms that mine high-utility itemsets using a single phase were proposed to avoid the problem of candidate generation. The d^2HUP and HUI-Miner algorithms were reported to be respectively up to 10 and 100 times faster than UP-Growth [7, 8]. Then, improved versions of HUI-Miner named HUP-Miner [8] and FHM [2] were proposed to reduce the number of join operations performed by HUI-Miner, and are to our knowledge the current best algorithm for HUIM. In HUI-Miner, HUP-Miner and FHM, each itemset is associated with a structure named *utility-list* [2, 7]. 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.

Definition 4 (Remaining utility). Let $\succ$ be a total order on items from I, and X be an itemset. The *remaining utility* of X in a transaction T_c is defined as $re(X, T_c) = \sum_{i \in T_c \wedge i \succ x \forall x \in X} u(i, T_c)$.

Definition 5 (Utility-list). 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_c containing X. The *iutil* and *rutil* elements of a tuple respectively are the utility of X in T_c ($u(X, T_c)$) and the remaining utility of X in T_c ($re(X, T_c)$).

For example, assume the lexicographical order (i.e. $e \succ d \succ c \succ b \succ a$). The utility-list of $\{a, e\}$ is $\{(T_2, 16, 5), (T_3, 8, 5)\}$.

To discover high-utility itemsets, HUI-Miner, HUP-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 (see [7, 8] for details). Pruning the search space is done using the following properties.

Definition 6 (Remaining utility upper-bound). 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.

Property 2 (Pruning search space using utility-lists). If $reu(X) < minutil$, then X is a low-utility itemset as well as all its extensions [7].

One-phase algorithms [2, 7, 8, 10] are faster and generally more memory efficient than two phase algorithms. However, mining HUIs remains very computationally expensive. For example, HUI-Miner, HUP-Miner, and FHM still suffer from a high space and time complexity. The size of each utility-list is in the worst case $O(n)$, where n is the number of transactions (when a utility-list contains an entry for each transaction). The complexity of building a utility-list is also very high [2]. In general, it requires to join three utility-lists of smaller itemsets, which thus requires $O(n^3)$ time in the worst case.

4 The EFIM Algorithm

We next present our proposal, the EFIM algorithm. It is a one-phase algorithm, which introduces several novel ideas to reduce the time and memory required for HUIM. Subsection 4.1 briefly reviews definitions related to the depth-first search of itemsets. Subsection 4.2 and 4.3 respectively explain how EFIM reduces the cost of database scans using an efficient implementation of database projection and transaction merging. Subsection 4.4 presents two new upper-bounds used by EFIM to prune the search space. Subsection 4.5 presents a new array-based utility counting technique named *Fast Utility Counting* to efficiently calculate these upper-bounds in linear time and space. Finally, subsection 4.6 gives the pseudocode of EFIM.

4.1 The Search Space

Let $\succ$ be any total order on items from I. According to this order, the search space of all itemsets 2^I can be represented as a *set-enumeration tree*. For example, the set-enumeration tree of $I = \{a, b, c, d\}$ for the lexicographical order is shown in Fig. 1. The EFIM algorithm explores this search space using a depth-first search starting from the root (the empty set). During this depth-first search, for any itemset α, EFIM recursively appends one item at a time to α according to the $\succ$ order, to generate larger itemsets. In our implementation, the $\succ$ order is defined as the order of increasing TWU because it generally reduces the search space for HUIM [2, 7, 12]. However, we henceforth assume that $\succ$ is the lexicographical order, for the sake of simplicity. We next introduce definitions related to the depth-first search exploration of itemsets.

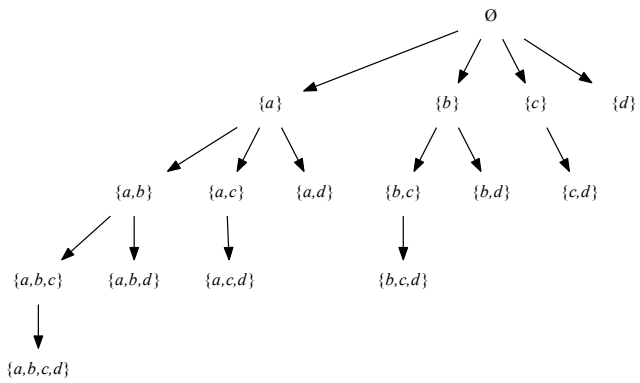

Fig. 1. Set-enumeration tree for $I = \{a, b, c, d\}$

Definition 7 (Items that can extend an itemset). Let be an itemset α. Let $E(\alpha)$ denote the set of all items that can be used to extend α according to the depth-first search, that is $E(\alpha) = \{z | z \in I \wedge z \succ x, \forall x \in \alpha\}$.

Definition 8 (Extension of an itemset). Let be an itemset α. An itemset Z is an *extension* of α (appears in a sub-tree of α in the set-enumeration tree) if $Z = \alpha \cup W$ for an itemset $W \in 2^{E(\alpha)}$ such that $W \neq \emptyset$. Furthermore, an itemset Z is a *single-item extension* of α (is a child of α in the set-enumeration tree) if $Z = \alpha \cup \{z\}$ for an item $z \in E(\alpha)$.

For example, consider the database of our running example and $\alpha = \{d\}$. The set $E(\alpha)$ is $\{e, f, g\}$. Single-item extensions of α are $\{d, e\}$, $\{d, f\}$ and $\{d, g\}$. Other extensions of α are $\{d, e, f\}$, $\{d, f, g\}$ and $\{d, e, f, g\}$.

4.2 Reducing the Cost of Database Scans using Projections

As we will explain, EFIM performs database scans to calculate the utility of itemsets and upper-bounds on their utility. To reduce the cost of database scans, it is desirable to reduce the database size. An observation is that when an itemset α is considered during the depth-first search, all items $x \notin E(\alpha)$ can be ignored when scanning the database to calculate the utility of itemsets in the sub-tree of α, or upper-bounds on their utility. A database without these items is called a *projected database*.

Definition 9 (Projected database). The *projection of a transaction T using an itemset α* is denoted as α-T and defined as α-$T = \{i | i \in T \wedge i \in E(\alpha)\}$. The *projection of a database D using an itemset α* is denoted as α-D and defined as the multiset α-$D = \{\alpha\text{-}T | T \in D \wedge \alpha\text{-}T \neq \emptyset\}$.

For example, consider database D of the running example and $\alpha = \{b\}$. The projected database α-D contains three transactions: α-$T_3 = \{c, d, e, f\}$, α-$T_4 = \{c, d, e\}$ and α-$T_5 = \{c, e, g\}$.

Database projections generally greatly reduce the cost of database scans since transactions become smaller as larger itemsets are explored. However, an important issue is how to implement database projection efficiently. A naive and inefficient approach is to make physical copies of transactions for each projection. An efficient approach used in EFIM is to sort each transaction in the original database according to the $\succ$ total order beforehand. Then, a projection is performed as a *pseudo-projection*, that is each projected transaction is represented by an offset pointer on the corresponding original transaction. The complexity of calculating the projection α-D of a database D is linear in time and space with respect to the number of transactions. The proposed database projection technique generalizes the concept of database projection from pattern mining [14] for the case of transactions with internal/external utility values. Note that FP-growth based HUIM algorithms [6, 12] and hyper-link based HUIM algorithms [8] also perform projections but differently than the proposed EFIM algorithm.

4.3 Reducing the Cost of Database Scans by Transaction Merging

To further reduce the cost of database scans, EFIM also introduce an efficient transaction merging technique. It is based on the observation that transaction databases often contain identical transactions. The technique consists of identifying these transactions and to replace them with single transactions. In this context, a transaction T_a is said to be *identical to* a transaction T_b if it contains the same items as T_b (i.e. $T_a = T_b$) (but not necessarily the same internal utility values).

Definition 10 (Transaction merging). *Transaction merging* consists of replacing a set of identical transactions $Tr_1, Tr_2, ...Tr_m$ in a database D by a single new transaction $T_M = Tr_1 = Tr_2 = ... = Tr_m$ where the quantity of each item $i \in T_M$ is defined as $q(i, T_M) = \sum_{k=1...m} q(i, Tr_k)$.

But to achieve higher database reduction, we also merge transactions in projected databases. This generally achieves a much higher reduction because projected transactions are smaller than original transactions, and thus are more likely to be identical.

Definition 11 (Projected transaction merging). *Projected transaction merging* consists of replacing a set of identical transactions $Tr_1, Tr_2, ...Tr_m$ in a database α-D by a single new transaction $T_M = Tr_1 = Tr_2 = ... = Tr_m$ where the quantity of each item $i \in T_M$ is defined as $q(i, T_M) = \sum_{k=1...m} q(i, Tr_k)$.

For example, consider database D of our running example and $\alpha = \{c\}$. The projected database α-D contains transactions α-$T_1 = \{d\}$, α-$T_2 = \{e, g\}$, α-$T_3 = \{d, e, f\}$, α-$T_4 = \{d, e\}$ and α-$T_5 = \{e, g\}$. Transactions α-T_2 and α-T_5 can be replaced by a new transaction $T_M = \{e, g\}$ where $q(e, T_M) = 3$ and $q(g, T_M) = 7$.

Transaction merging is obviously desirable. However, a key problem is to implement it efficiently. The naive approach to identify identical transactions is

to compare all transactions with each other. But this is inefficient ($O(n^2)$, where n is the number of transactions). To find identical transactions in $O(n)$ time, we propose the following approach. We initially sort the original database according to a new total order $\succ_T$ on transactions. Sorting is achieved in $O(n\ log(n))$ time, and is performed only once.

Definition 12 (Total order on transactions). The $\succ_T$ order is defined as the lexicographical order when the transactions are read backwards. Formally, let be two transactions $T_a = \{i_1, i_2, ...i_m\}$ and $T_b = \{j_1, j_2, ...j_k\}$. The total order $\succ_T$ is defined by four cases. The first case is that $T_b \succ T_a$ if both transactions are identical and T_b is greater than the TID of T_a. The second case is that $T_b \succ_T T_a$ if $k > m$ and $i_{m-x} = j_{k-x}$ for any integer x such that $0 \leq x < m$. The third case is that $T_b \succ_T T_a$ if there exists an integer x such that $0 \leq x < \min(m, k)$, where $j_{k-x} \succ i_{m-x}$ and $i_{m-y} = j_{k-y}$ for all integer y such that $x < y < \min(m, k)$. The fourth case is that otherwise $T_a \succ_T T_b$.

For example, let be transactions $T_x = \{b, c\}$, $T_y = \{a, b, c\}$ and $T_z = \{a, b, e\}$. We have that $T_z \succ_T T_y \succ_T T_x$.

A database sorted according to the $\succ_T$ order provides the following property (proof omitted due to space limitation).

Property 3 (Order of identical transactions in a $\succ_T$ sorted database). Let be a $\succ_T$ sorted database D and an itemset α. Identical transactions appear consecutively in the projected database α-D.

Using the above property, all identical transactions in a (projected) database can be identified by only comparing each transaction with the next transaction in the database. Thus, using this scheme, transaction merging can be done very efficiently by scanning a (projected) database only once (linear time). It is interesting to note that transaction merging as proposed in EFIM cannot be implemented efficiently in utility-list based algorithms (e.g. FHM and HUP-Miner) and hyperlink-based algorithms (e.g. d^{2}HUP) due to their database representations.

4.4 Pruning Search Space using Sub-tree Utility and Local Utility

To propose an efficient HUIM algorithm, a key problem is to design an effective pruning mechanism. For this purpose, we introduce in EFIM two new upper-bounds on the utility of itemsets named *sub-tree utility* and *local utility*. The key difference with previous upper-bounds is that they are defined w.r.t the sub-tree of an itemset α in the search-enumeration tree.

Definition 13 (Sub-tree utility). Let be an itemset α and an item z that can extend α according to the depth-first search ($z \in E(\alpha)$). The *Sub-tree Utility* of z w.r.t. α is $su(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\})} u(i, T)]$.

Example 2. Consider the running example and $\alpha = \{a\}$. We have that $su(\alpha, c)$ $= (5+1+2) + (10+6+11) + (5+1+20) = 61$, $su(\alpha, d) = 25$ and $su(\alpha, e) = 34$.

The following theorem of the sub-tree utility is proposed in EFIM to prune the search space (proof ommited due to space limitation).

Property 4 (Overestimation using the sub-tree utility). Let be an itemset α and an item $z \in E(\alpha)$. The relationship $su(\alpha, z) \geq u(\alpha \cup \{z\})$ holds. And more generally, $su(\alpha, z) \geq u(Z)$ holds for any extension Z of $\alpha \cup \{z\}$.

Theorem 1 (Pruning a sub-tree using the sub-tree utility). Let be an itemset α and an item $z \in E(\alpha)$. If $su(\alpha, z) < minutil$, then the single item extension $\alpha \cup \{z\}$ and its extensions are low-utility. In other words, the sub-tree of $\alpha \cup \{z\}$ in the set-enumeration tree can be pruned.

Thus, using Theorem 1, we can prune some sub-trees of an itemset α, which reduces the number of sub-trees to be explored. To further reduce the search space, we also identify items that should not be explored in any sub-trees.

Definition 14 (Local utility). Let be an itemset α and an item $z \in E(\alpha)$. The *Local Utility* of z w.r.t. α is $lu(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + re(\alpha, T)]$.

Example 3. Consider the running example and $\alpha = \{a\}$. We have that $lu(\alpha, c)$ $= (8 + 27 + 30) = 65$, $lu(\alpha, d) = 30$ and $lu(\alpha, e) = 57$.

Property 5 (Overestimation using the local utility). Let be an itemset α and an item $z \in E(\alpha)$. Let Z be an extension of α such that $z \in Z$. The relationship $lu(\alpha, z) \geq u(Z)$ holds.

Theorem 2 (Pruning an item from all sub-trees using the local utility). Let be an itemset α and an item $z \in E(\alpha)$. If $lu(\alpha, z) < minutil$, then all extensions of α containing z are low-utility. In other words, item z can be ignored when exploring all sub-trees of α.

The relationship between the proposed upper-bounds and the main ones used in previous work is the following.

Property 6 (Relationships between upper-bounds). Let be an itemset $Y = \alpha \cup \{z\}$. The relationship $TWU(Y) \geq lu(\alpha, z) \geq reu(Y) = su(\alpha, z)$ holds.

Given, the above relationship, it can be seen that the proposed local utility upper-bound is a tighter upper-bound on the utility of Y and its extensions compared to the TWU, which is commonly used in two-phase HUIM algorithms. Thus the local utility can be more effective for pruning the search space. Besides, one can ask what is the difference between the proposed *su* upper-bound and the *reu* upper-bound of HUI-Miner and FHM since they are mathematically equivalent. The major difference is that *su* is calculated when the depth-first search is at itemset α in the search tree rather than at the child itemset Y. Thus, if $su(\alpha, z) < minutil$, EFIM prunes the whole sub-tree of z including node Y rather than only pruning the descendant nodes of Y. Thus, using *su* instead of *reu* is more effective for pruning the search space. In the rest of the paper, for a given itemset α, we respectively refer to items having a sub-tree utility and local-utility no less than *minutil* as *primary* and *secondary items*.

Definition 15 (Primary and secondary items). Let be an itemset α. The *primary items of* α is the set of items defined as $Primary(\alpha) = \{z | z \in E(\alpha) \wedge su(\alpha, z) \geq minutil\}$. The *secondary items of* α is the set of items defined as $Secondary(\alpha) = \{z | z \in E(\alpha) \wedge lu(\alpha, z) \geq minutil\}$. Because $lu(\alpha, z) \geq su(\alpha, z)$, $Primary(\alpha) \subseteq Secondary(\alpha)$.

For example, consider the running example and $\alpha = \{a\}$. $Primary(\alpha) = \{c, e\}$. $Secondary(\alpha) = \{c, d, e\}$.

4.5 Calculating Upper-Bounds Efficiently using Fast Utility Counting

In the previous subsection, we introduced two new upper-bounds on the utility of itemsets to prune the search space. We now present a novel efficient array-based approach to compute these upper-bounds in linear time and space that we call Fast Utility Counting (FUC). It relies on a novel array structure called utility-bin.

Definition 16 (Utility-Bin). Let be the set of items I appearing in a database D. A *utility-bin array* U for database D is an array of length $|I|$, having an entry denoted as $U[z]$ for each item $z \in I$. Each entry is called a *utility-bin* and is used to store a utility value (an integer in our implementation, initialized to 0).

A utility-bin array can be used to efficiently calculate the following upper-bounds in $O(n)$ time (recall that n is the number of transactions), as follows.

Calculating the TWU of all items. A utility-bin array U is initialized. Then, for each transaction T of the database, the utility-bin $U[z]$ for each item $z \in T$ is updated as $U[z] = U[z] + TU(T)$. At the end of the database scan, for each item $k \in I$, the utility-bin $U[k]$ contains $TWU(k)$.

Calculating the sub-tree utility w.r.t. an itemset α. A utility-bin array U is initialized. Then, for each transaction T of the database, the utility-bin $U[z]$ for each item $z \in T \cap E(\alpha)$ is updated as $U[z] = U[z] + u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \succ z} u(i, T)$. Thereafter, we have $U[k] = su(\alpha, k)\ \forall k \in I$.

Calculating the local utility w.r.t. an itemset α. A utility-bin array U is initialized. Then, for each transaction T of the database, the utility-bin $U[z]$ for each item $z \in T \cap E(\alpha)$ is updated as $U[z] = U[z] + u(\alpha, T) + re(\alpha, T)$. Thereafter, we have $U[k] = lu(\alpha, k)\ \forall k \in I$.

Thus, by the above approach, the three upper-bounds can be calculated for all items that can extend an itemset α with only one (projected) database scan. Furthermore, it can be observed that utility-bins are a very compact data structure ($O(|I|)$ size). To utilize utility-bins more efficiently, we propose three optimizations. First, all items in the database are renamed as consecutive integers. Then, in a utility-bin array U, the utility-bin $U[i]$ for an item i is stored in the i-th position of the array. This allows to access the utility-bin of an item in $O(1)$ time. Second, it is possible to reuse the same utility-bin array multiple times by reinitializing it with zero values before each use. This avoids creating

multiple arrays and thus reduces memory usage. In our implementation, only three utility-bin arrays are created, to respectively calculate the TWU, sub-tree utility and local utility. Third, when reinitializing a utility-bin array to calculate the sub-tree utility or the local utility of single-item extensions of an itemset α, only utility-bins corresponding to items in $E(\alpha)$ are reset to 0, for faster reinitialization of the utility-bin array.

4.6 The Proposed Algorithm

In this subsection, we present the EFIM algorithm, which combines all the ideas presented in the previous section. The main procedure (Algorithm 1) takes as input a transaction database and the *minutil* threshold. The algorithm initially considers that the current itemset α is the empty set. The algorithm then scans the database once to calculate the local utility of each item w.r.t. α, using a utility-bin array. Note that in the case where $\alpha = \emptyset$, the local utility of an item is its TWU. Then, the local utility of each item is compared with *minutil* to obtain the secondary items w.r.t to α, that is items that should be considered in extensions of α. Then, these items are sorted by ascending order of TWU and that order is thereafter used as the $\succ$ order (as suggested in [2, 7]). The database is then scanned once to remove all items that are not secondary items w.r.t to α since they cannot be part of any high-utility itemsets by Theorem 2. If a transaction becomes empty, it is removed from the database. Then, the database is scanned again to sort transactions by the $\succ_T$ order to allow $O(n)$ transaction merging, thereafter. Then, the algorithm scans the database again to calculate the sub-tree utility of each secondary item w.r.t. α, using a utility-bin array. Thereafter, the algorithm calls the recursive procedure *Search* to perform the depth first search starting from α.

Algorithm 1: The EFIM algorithm

input : D: a transaction database, *minutil*: a user-specified threshold
output: the set of high-utility itemsets

1 $\alpha = \emptyset$;
2 Calculate $lu(\alpha, i)$ for all items $i \in I$ by scanning D, using a utility-bin array;
3 $Secondary(\alpha) = \{i | i \in I \wedge lu(\alpha, i) \geq minutil\}$;
4 Let $\succ$ be the total order of TWU ascending values on $Secondary(\alpha)$;
5 Scan D to remove each item $i \notin Secondary(\alpha)$ from the transactions, and delete empty transactions;
6 Sort transactions in D according to $\succ_T$;
7 Calculate the sub-tree utility $su(\alpha, i)$ of each item $i \in Secondary(\alpha)$ by scanning D, using a utility-bin array;
8 $Primary(\alpha) = \{i | i \in Secondary(\alpha) \wedge su(\alpha, i) \geq minutil\}$;
9 `Search` (α, D, $Primary(\alpha)$, $Secondary(\alpha)$, *minutil*);

The *Search* procedure (Algorithm 2) takes as parameters the current itemset to be extended α, the α projected database, the primary and secondary items w.r.t α and the *minutil* threshold. The procedure performs a loop to consider each single-item extension of α of the form $\beta = \alpha \cup \{i\}$, where i is a primary item w.r.t α (since only these single-item extensions of α should be explored according to Theorem 1). For each such extension β, a database scan is performed to calculate the utility of β and at the same time construct the β projected database. Note that transaction merging is performed whilst the β projected database is constructed. If the utility of β is no less than *minutil*, β is output as a high-utility itemset. Then, the database is scanned again to calculate the sub-tree and local utility w.r.t β of each item z that could extend β (the secondary items w.r.t to α), using two utility-bin arrays. This allows determining the primary and secondary items of β. Then, the *Search* procedure is recursively called with β to continue the depth-first search by extending β. Based on properties and theorems presented in previous sections, it can be seen that when EFIM terminates, all and only the high-utility itemsets have been output.

Complexity. A rough analysis of the complexity is as follows. To process each primary itemset α encountered during the depth-first search, EFIM performs database projection, transaction merging and upper-bound calculation in $O(n)$ time. In terms of space, utility-bin arrays are created once and require $O(|I|)$ space. The database projection performed for each primary itemset α requires at most $O(n)$ space. In practice, this is small considering that projected databases become smaller as larger itemsets are explored, and are implemented using offset pointers.

Algorithm 2: The *Search* procedure

input : α: an itemset, α-D: the α projected database, $Primary(\alpha)$: the primary items of α, $Secondary(\alpha)$: the secondary items of α, the *minutil* threshold

output: the set of high-utility itemsets that are extensions of α

foreach *item* $i \in Primary(\alpha)$ **do**
  $\beta = \alpha \cup \{i\}$;
  Scan α-D to calculate $u(\beta)$ and create β-D; `// uses transaction merging`
  if $u(\beta) \geq minutil$ **then** output β;
  Calculate $su(\beta, z)$ and $lu(\beta, z)$ for all item $z \in Secondary(\alpha)$ by scanning β-D once, using two utility-bin arrays;
  $Primary(\beta) = \{z \in Secondary(\alpha) | su(\beta, z) \geq minutil\}$;
  $Secondary(\beta) = \{z \in Secondary(\alpha) | lu(\beta, z) \geq minutil\}$;
  `Search` (β, β-D, $Primary(\beta)$, $Secondary(\beta)$, *minutil*);
end

5 Experimental Results

We performed experiments to evaluate the performance of the proposed EFIM algorithm. Experiments were carried out on a computer with a fourth generation 64 bit core i7 processor running Windows 8.1 and 16 GB of RAM. We compared the performance of EFIM with the state-of-the-art algorithms UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM.

Algorithms were implemented in Java and memory measurements were done using the Java API. Experiments were performed using a set of standard datasets used in the HUIM literature for evaluating HUIM algorithms, namely (*Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*). Table 3 summarizes their characteristics. *Foodmart* contains real external/internal utility values. For other datasets, external/internal utility values have been respectively generated in the $[1, 000]$ and $[1, 5]$ intervals using a log-normal distribution, as done in previous state-of-the-art HUIM studies [2, 7, 12]. The datasets and the source code of the compared algorithms can be downloaded as part of the SPMF data mining library `http://goo.gl/rIKIub` [3].

Table 3. Dataset characteristics

Dataset	# Transactions	# Distinct items	Avg. trans. length
Accident	340,183	468	33.8
BMS	59,601	497	4.8
Chess	3,196	75	37.0
Connect	67,557	129	43.0
Foodmart	4,141	1,559	4.4
Mushroom	8,124	119	23.0

Influence of the *minutil* threshold on execution time. We first compare execution times of the various algorithms. We ran the algorithms on each dataset while decreasing the *minutil* threshold until algorithms were too slow, ran out of memory or a clear winner was observed. Execution times are shown in Fig. 2. Note that for UP-Growth+, no result is shown for the connect dataset and that some results are missing for the chess dataset because UP-Growth+ exceeded the 16 GB memory limit. It can be observed that EFIM clearly outperforms UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM on all datasets. EFIM is in general about two to three orders of magnitude faster than these algorithms. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*, EFIM is respectively up to 15,334, 2, 33,028, –, 17 and 3,855 times faster than UP-Growth+, 154, 741, 323, 22,636, 2 and 85 times faster than HUP-Miner, 89, 1,490, 109, 2,587, 1 and 15 times faster than d^{2}HUP, 236, 2,370, 482, 10,586, 3 and 110 times faster than HUI-Miner and 145, 227, 321, 6,606, 1 and 90 times faster than FHM. An important reason why EFIM performs so well is that the proposed upper-bounds allows EFIM to prune a larger part of the search space compared

to other algorithms (as will be shown). The second reason is that the proposed transaction merging technique often greatly reduces the cost of database scans. It was observed that EFIM on *Chess*, *Connect* and *Mushroom*, EFIM is up to 116, 3,790 and 55 and times faster than a version of EFIM without transaction merging. For other datasets, transaction merging reduces execution times but by a lesser amount (EFIM is up to 90, 2 and 2 times than a version of EFIM without transaction merging on *Accident*, *BMS* and *Foodmart*). The third reason is that the calculation of upper-bounds is done in linear time using utility-bins. It is also interesting to note that transaction merging cannot be implemented efficiently in utility-list based algorithms such as HUP-Miner, HUI-Miner and FHM, due to their vertical database representation, and also for hyperlink-based algorithms such as the d^{2}HUP algorithm.

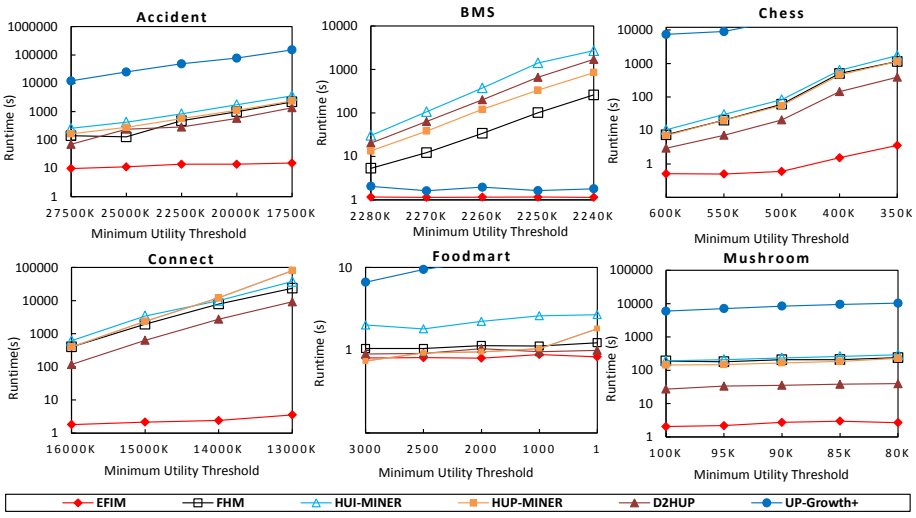

Fig. 2. Execution times on different datasets

Influence of the *minutil* threshold on memory usage. In terms of memory usage, EFIM also clearly outperforms other algorithms as shown in Table 4. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*, EFIM uses 1.8, 4.4, 14.9, 27.0, 1.3 and 6.5 times less memory than the second fastest algorithm (d^{2}HUP). Moreover, EFIM uses 1.6, 9.2, 4.6, 8.1, 3.2 and 3.1 times less memory than the third fastest algorithm (FHM). It is also interesting that EFIM utilizes less than 100 MB on four out of the six datasets, and never more than 1 GB, while other algorithms often exceed 1 GB.

A reason why EFIM is so memory efficient is that it use a simple database representation, which does not requires to maintain much information in memory (only pointers for pseudo-projections). Other algorithms relies on complex structures such as tree-structures (e.g. UPGrowth+) and list-structures (e.g.

HUP-Miner, HUI-Miner and FHM), which requires additional memory. Moreover, projected databases generated by EFIM are often very small due to transaction merging. Another reason is that the number of projected databases created by EFIM is small, because EFIM visits less nodes of the search-enumeration tree (as we will show later). EFIM is also more efficient than two-phase algorithms such as UPGrowth+ since it is a one-phase algorithm. Lastly, another important characteristic of EFIM in terms of memory efficiency is that it reuses some of its data structures. For example, FAC only requires to create three arrays that are reused to calculate the upper-bounds of each itemset considered during the depth-first search.

Table 4. Comparison of maximum memory usage (MB)

Dataset	HUI-MINER	FHM	EFIM	UP-Growth+	HUP-Miner	d^2HUP
Accident	1,656	1,480	**895**	765	1,787	1,691
BMS	210	590	**64**	64	758	282
Chess	405	305	**65**	–	406	970
Connect	2,565	3,141	**385**	–	1,204	1,734
Foodmart	808	211	**64**	819	68	84
Mushroom	194	224	**71**	1,507	196	468

Comparison of the number of visited nodes. We also performed an experiment to compare the ability at pruning the search space of EFIM to other algorithm. Table 5 shows the number of nodes of the search-enumeration tree (itemsets) visited by EFIM, UP-Growth+, HUP-Miner, d^2HUP, HUI-Miner and FHM for the lowest *minutil* values on the same datasets. It can be observed that EFIM is much more effective at pruning the search space than the other algorithms, thanks to its proposed sub-tree utility and local utility upper-bounds.

Table 5. Comparison of visited node count

Dataset	HUI-MINER	FHM	EFIM	UP-Growth+	HUP-Miner	d^2HUP
Accident	131,300	128,135	**51,883**	3,234,611	113,608	119,427
BMS	2,205,782,168	212,800,883	**323**	91,195	205,556,936	220,323,377
Chess	6,311,753	6,271,900	**2,875,166**	–	6,099,484	5,967,414
Connect	3,444,785	3,420,253	**1,366,893**	–	3,385,134	3,051,789
Foodmart	55,172,950	1,880,740	**233,231**	233,231	1,258,820	233,231
Mushroom	3,329,191	3,089,819	**2,453,683**	13,779,114	3,054,253	2,919,842

6 Conclusion

We have presented a novel algorithm for high-utility itemset mining named EFIM. It relies on two new upper-bounds named *sub-tree utility* and *local utility*. It also introduces a novel array-based utility counting approach named *Fast Utility Counting* to calculate these upper-bounds in linear time and space. Moreover, to reduce the cost of database scans, EFIM introduces techniques for database projection and transaction merging, also performed in linear time and space. An extensive experimental study on various datasets shows that EFIM is in general two to three orders of magnitude faster and consumes up to eight times less memory than the state-of-art algorithms UP-Growth+, HUP-Miner, d^2HUP, HUI-Miner and FHM. The source code of all algorithms and datasets used in the experiments can be downloaded as part of the SPMF data mining library `goo.gl/rIKIub` [3]. For future work, we will extend EFIM for popular variations of the HUIM problem such as mining closed+ high-utility itemset [13], generators of high-utility itemsets [5], and on-shelf high-utility itemsets [4].

Acknowledgement This work is financed by a National Science and Engineering Research Council (NSERC) of Canada research grant.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. Intern. Conf. Very Large Databases, pp. 487–499 (1994)
2. Fournier-Viger, P., Wu, C.-W., Zida, S., Tseng, V. S.: FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proc. 21st Intern. Symp. on Methodologies for Intell. Syst., pp. 83–92 (2014)
3. Fournier-Viger, P, Gomariz, A., Gueniche, T., Soltani, A., Wu., C.-W., Tseng, V. S.: SPMF: a Java Open-Source Pattern Mining Library. Journal of Machine Learning Research, 15, 3389–3393 (2014)
4. Fournier-Viger, P., Zida, S.: Foshu: Faster On-Shelf High Utility Itemset Mining with or without negative unit profit. Proc. 30th ACM Symposium on Applied Computing, pp. 857–864 (2015)
5. Fournier-Viger, P., Wu, C.-W., Tseng, V. S. : Novel Concise Representations of High Utility Itemsets using Generator Patterns. Proc. of 10th Intern. Conference on Advanced Data Mining and Applications, pp. 30–43 (2014)
6. Lan, G. C., Hong, T. P., Tseng, V. S.: An efficient projection-based indexing approach for mining high utility itemsets. Knowl. and Inform. Syst. 38(1), 85–107 (2014)
7. Liu, M., Qu, J.: Mining high utility itemsets without candidate generation. In: Proc. 22nd ACM Intern. Conf. Info. and Know. Management, pp. 55–64 (2012)
8. Krishnamoorthy, S.: Pruning strategies for mining high utility itemsets. Expert Systems with Applications, 42(5), 2371-2381 (2015)
9. Liu, Y., Liao, W., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Proc. 9th Pacific-Asia Conf. on Knowl. Discovery and Data Mining, pp. 689–695 (2005)

10. Liu, J., Wang, K., Fung, B.: Direct discovery of high utility itemsets without candidate generation. Proc. 12th IEEE Intern. Conf. Data Mining, pp. 984–989 (2012)
11. Song, W., Liu, Y., Li, J.: BAHUI: Fast and memory efficient mining of high utility itemsets based on bitmap. Intern. Journal of Data Warehousing and Mining. 10(1), 1–15 (2014)
12. Tseng, V. S., Shie, B.-E., Wu, C.-W., Yu., P. S.: Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans. Knowl. Data Eng. 25(8), 1772–1786 (2013)
13. Tseng, V., Wu, C., Fournier-Viger, P., Yu, P.: Efficient algorithms for mining the concise and lossless representation of closed+ high utility itemsets. IEEE Trans. Knowl. Data Eng. 27(3), 726–739 (2015)
14. Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: Proc. ICDM'04 Workshop on Frequent Itemset Mining Implementations, CEUR (2004)
15. Zida, S., Fournier-Viger, P., Wu, C.-W., Lin, J. C. W., Tseng, V.S.: Efficient mining of high utility sequential rules. in: Proc. 11th Intern. Conf. Machine Learning and Data Mining, pp. 1–15 (2015)