EFIM-Closed: Fast and Memory Efficient Discovery of Closed High-Utility Itemsets

Philippe Fournier-Viger[1], Souleymane Zida[2], Jerry Chun-Wei Lin[3], Cheng-Wei Wu[4], Vincent S. Tseng[4],

[1] School of Natural Sciences and Humanities, Harbin Institute of Technology Shenzhen Graduate School, China
[2] Department of Computer Science, University of Moncton, Canada
[3] School of Computer Science and Technology, Harbin Institute of Technology Shenzhen Graduate School, China
[4] Department of Computer Science, National Chiao Tung University, Taiwan
philfv@hitsz.edu.cn, esz2233@umoncton.ca, jerrylin@ieee.org, silvemoonfox@gmail.com, tsengsm@mail.ncku.edu.tw

Abstract. Discovering high-utility temsets in transaction databases is a popular data mining task. A limitation of traditional algorithms is that a huge amount of high-utility itemsets may be presented to the user. To provide a concise and lossless representation of results to the user, the concept of closed high-utility itemsets was proposed. However, mining closed high-utility itemsets is computationally expensive. To address this issue, we present a novel algorithm for discovering closed high-utility itemsets, named EFIM-Closed. This algorithm includes novel pruning strategies named *closure jumping*, *forward closure checking* and *backward closure checking* to prune non-closed high-utility itemsets. Furthermore, it also introduces novel utility upper-bounds and a transaction merging mechanism. Experimental results shows that EFIM-Closed can be more than an order of magnitude faster and consumes more than an order of magnitude less memory than the previous state-of-art CHUD algorithm.

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

1 Introduction

High Utility Itemset Mining (*HUIM*) [2, 3, 6, 8, 9, 7, 10, 14] is a popular data mining task for discovering useful patterns in customer transaction databases. It consists of discovering itemsets that yield a high utility (e.g. high profit), that is *High Utility Itemsets* (*HUIs*). Besides customer transaction analysis, HUIM also has applications in other domains such as click stream analysis and biomedicine [2, 3, 8, 10]. HUIM can be viewed as an extension of the problem of *Frequent itemset Mining* (*FIM*) [1], where a weight (e.g unit profit) may be assigned to each item, and where purchase quantities of items in transactions are not restricted to binary values. HUIM is generally viewed as a difficult problem, because the *utility* measure used in HUIM is neither *monotonic* or *anti-monotonic*, unlike

the *support* measure in FIM. That is, the utility of an itemset may be greater, smaller or equal to the utility of its subsets. For this reason, efficient search space pruning techniques developped in FIM cannot be used in HUIM.

Several algorithms have been proposed for HUIM [2, 3, 6, 8, 9, 7, 10, 14]. However, an important limitation of traditional HUIM algorithms is that they often produce a huge amount of high-utility itemsets. Hence, it can be very time-consuming for users to analyze the output of these algorithms. Moreover, this makes HUIM algorithms suffer from long execution times and even fail to run due to huge memory consumption or lack of storage space. To address this issue, it was recently proposed to mine a concise and lossless representation of all HUIs named closed high-utility itemsets (CHUIs) [11]. The concept of CHUI extends the concept of *closed patterns* [13, 12] from FIM. A CHUI is a HUI having no proper supersets that are HUIs and appear in the same number of transactions [11]. This latter representation is interesting since it is lossless (it allows deriving all HUIs). Furthermore, it is also meaningful for real applications since it only discovers the largest HUIs that are common to groups of customers. However, CHUI mining can be very computationally expensive [11].

In this paper, we address the need for a more efficient CHUI mining algorithm by proposing an algorithm named EFIM-Closed (EFficient high-utility Itemset Mining - Closed), based on the strict constraint that for each itemset in the search space, all operations for that itemset should be performed in linear time and space. EFIM-Closed propose three strategies to discover CHUIs efficiently: closure jumping (CJU), forward closure checking (FCC) and backward closure checking (BCC). To reduce the cost of database scans, EFIM-Closed relies on two efficient techniques named *High-utility Database Projection* (HDP) and *High-utility Transaction Merging* (HTM). Also, the proposed EFIM-Closed algorithm includes two new upper-bounds on the utility of itemsets named *sub-tree utility* and *local utility* to effectively prune the search space, and an efficient *Fast Utility Counting* (FAC) technique to compute them. An experimental study show that EFIM-Closed is up to 71 times faster and consumes up to 18 times less memory than the state-of-the-art CHUD algorithm, and has excellent scalability.

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-Closed algorithm, the experimental evaluation and the conclusion.

2 Problem Statement

This section introduces the problem of closed high-utility itemset mining. 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

The utility of an item i in a transaction T_c is denoted as $u(i, T_c)$ and defined as $p(i) \times q(i, T_c)$. 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 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. The *support* of an itemset X in a database D is denoted as $sup(X)$ and defined as $|g(X)|$. For example, the utility of item a in T_2 is $u(a, T_2) = 5 \times 2 = 10$, and its support is 1. The utility of itemset $\{a, c\}$ is $u(\{a, c\}) = u(\{a, c\}, T_1) + u(\{a, c\}, T_2) + u(\{a, c\}, T_3) = u(a, T_1) + u(c, T_1) + u(a, T_2) + u(c, T_2) + u(a, T_3) + u(c, T_3) = 5 + 1 + 10 + 6 + 5 + 1 = 28$.

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*. A HUI X is a *closed high-utility itemset* (*CHUI*) [11] iff there exists no HUI Y such that $X \subset Y$ and $sup(X) = sup(Y)$. The *problem of (closed) high-utility itemset mining* is to discover all (closed) high-utility itemsets, given a threshold *minutil*, set by the user [11]. 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. Among those, the closed high-utility itemsets are $\{a, b, c, d, e, f\}$,{b,c,d,e}, $\{b, c, e\}$ and $\{a, c, e\}$.

3 Related Work

A key challenge in HUIM is that search space prune techniques used in FIM cannot be used in HUIM because the utility measure is neither monotonic nor anti-monotonic [2, 9, 10]. Several HUIM algorithms circumvent this problem by overestimating the utility of itemsets using the concept of *Transaction-Weighted Utilization* (TWU) measure [2, 6, 9, 7, 10, 14], defined as follows.

Definition 1 (Transaction weighted utilization). The *transaction utility* of a transaction T_c is the sum of the utilities of items from T_c in that transaction.

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

For example, 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. The TWU of single items a, b, c, d, e, f and g are respectively 65, 61, 96, 58, 88, 30 and 38. The following property of the TWU is commonly used in HUIM to prune the search space.

Property 1 (Pruning using the TWU). Let be an itemset X. If $TWU(X) < minutil$, then X is a low-utility itemset as well as all its supersets [9].

Many HUIM algorithms [2, 6, 9, 7, 10, 11, 14] 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. Recently, algorithms that mine high-utility itemsets using a single phase were proposed to avoid the problem of candidate generation [3, 8], and were shown to outperform previous algorithms. One-phase algorithms rely mainly on the concept of *remaining utility* to prune the search space.

Definition 2 (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 $re(X, T_c) = \sum_{i \in T_c \wedge i \succ x \forall x \in X} u(i, T_c)$. The *remaining utility* of X in a database is defined as $re(X) = \sum_{T_c \in D} re(X, T_c)$.

Property 2 (Pruning using remaining utility). 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 an itemset X is defined as $reu(X) = u(X) + re(X)$. If $u(X) + reu(X) < minutil$, then X is a low-utility itemset as well as all its extensions [3, 8].

A crucial problem in HUIM is that the set of HUIs is often very large. To address this issue, it was proposed to mine the concise and representative subset of *closed HUIs* [11]. But mining CHUIs can be very computationally expensive. To address this issue, we next introduce a novel more efficient algorithm.

4 The EFIM-Closed Algorithm

The proposed EFIM-Closed algorithm is a highly efficient algorithm for closed HUI mining. It is a one phase algorithm designed using the strict design constraint that for each itemset in the search space, all operations for that itemset should be performed in linear time and space. This section is organized as follows. Subsection 4.1 introduces preliminary definitions related to the depth-first search of itemsets. Subsection 4.2 explains how EFIM-Closed reduces the cost of database scans. Subsection 4.3 explains how EFIM-closed prune low-utility itemsets in the search space. Subsection 4.4 explains how EFIM-Closed prunes non closed HUIs. Finally, subsection 4.5 put all the pieces together, and gives the full pseudocode of EFIM-Closed.

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-Closed 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-Closed 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, 3, 8, 10]. 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.

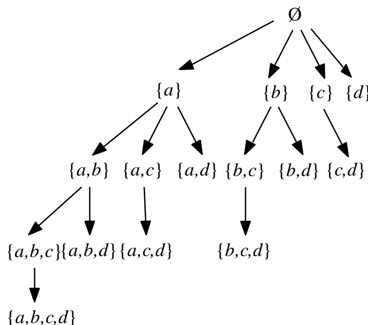

Fig. 1: Set-enumeration tree for $I = \{a, b, c, d\}$

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 $E(\alpha) = \{z | z \in I \wedge z \succ x, \forall x \in \alpha\}$. 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 $W \neq \emptyset$. 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 Scanning the Database Efficiently

As we will later explain, EFIM-Closed 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. In EFIM-Closed this is performed by two techniques.

High-utility Database Projection (HDP). This technique is based on the observation 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 3 (Projected database). The *projection of a transaction T using an itemset α* is denoted as α-T and defined as $\alpha\text{-}T = \{i | i \in T \wedge i \in E(\alpha)\}$. The *projection of a database D using an itemset α* is denoted as α-D and defined as the multiset $\alpha\text{-}D = \{\alpha\text{-}T | T \in D \wedge \alpha\text{-}T \neq \emptyset\}$.

For example, consider database D of the running example and $\alpha = \{b\}$. The projected database α-D contains three transactions: $\alpha\text{-}T_3 = \{c, d, e, f\}$, $\alpha\text{-}T_4 = \{c, d, e\}$ and $\alpha\text{-}T_5 = \{c, e, g\}$. Database projections generally greatly reduce the cost of database scans since transactions become smaller as larger itemsets are explored. To implement database projection efficiently, each transaction in the original database is sorted beforehand according to the $\succ$ total order. Then, each 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 performing a projection is $o(nw)$, where n is the number of transactions and w is their average length.

High-utility Transaction Merging (HTM). To further reduce the cost of database scans, EFIM-Closed also introduce an efficient transaction merging technique named *High-utilty Transaction Merging* (HTM). HTM is based on the observation that transaction databases often contain identical transactions (transactions containing exactly the same items, but not necessarily the same internal utility values). The technique consists of replacing a set of identical transactions $Tr_1, Tr_2, ...Tr_m$ in a (projected) 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 $\alpha\text{-}T_1 = \{d\}$, $\alpha\text{-}T_2 = \{e, g\}$, $\alpha\text{-}T_3 = \{d, e, f\}$, $\alpha\text{-}T_4 = \{d, e\}$ and $\alpha\text{-}T_5 = \{e, g\}$. Transactions $\alpha\text{-}T_2$ and $\alpha\text{-}T_5$ 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. To find identical transactions in $O(nw)$ time, we initially sort the original database according to a new total order $\succ_T$ on transactions defined as the $\succ$ order when the transactions are read backwards. For example, let be transactions $T_x = \{b, c\}$, $T_y = \{a, b, c\}$ and $T_z = \{a, b, e\}$. We have that $T_z \succ_T T_y \succ_T T_x$. Sorting is achieved in $O(nw\ log(nw))$ time. This cost is negligible because it is performed only once.

A database sorted according to the $\succ_T$ order provides the following property. For a database D or any projected database α-D, identical transactions always appear consecutively in the projected database α-D. This property holds because (1) transactions are sorted according to the $\succ$ order when read backwards and (2) projections always remove the smallest items of a transactions according to the $\succ$ order. 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-Closed is not performed in any other one-phase HUIM algorithms.

4.3 Pruning Low-Utility Itemsets

To propose an efficient CHUI mining algorithm, a key problem is to design an effective mechanism for pruning low-utility itemsets in the search space. For this purpose, we introduce two new upper-bounds on the utility of itemsets.

The subtree-utility and local utility upper-bounds. The first upper-bound is defined as follows.

Definition 4 (Sub-tree utility). Let be an itemset α and an item z such that $z \in E(\alpha)$. The *Sub-tree Utility* of z w.r.t. α is $su(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\})} u(i, T)]$.

For example, if $\alpha = \{a\}$, 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 allows EFIM-Closed to prune the search space (proof omitted due to space limitation).

Theorem 1 (Pruning using 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.

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

Definition 5 (Local utility). Let be an itemset α and an item $z \in E(\alpha)$. The *Local Utility* of z w.r.t. α is $lu(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})} [u(\alpha, T) + re(\alpha, T)]$.

For example, if $\alpha = \{a\}$, we have that $lu(\alpha, c) = (8 + 27 + 30) = 65$, $lu(\alpha, d) = 30$ and $lu(\alpha, e) = 57$. The following property can be used for pruning low-utility itemsets (proof ommitted due to space limitation).

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

The relationships between the proposed upper-bounds and the main ones used in previous work are the following. Let be an itemset α, an item z and an itemset $Y = \alpha \cup \{z\}$. It can be demonstrated easily that the relationship $TWU(Y) \geq lu(\alpha, z) \geq reu(Y) = su(\alpha, z)$ holds. Thus, the 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 such as

CHUD. About the su upper-bound, one can ask what is the difference between this upper-bound and the reu upper-bound used by some HUIM algorithms 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-Closed 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*. Formally, the *primary items of an itemset* 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 instance, consider that $\alpha = \{a\}$. $Primary(\alpha) = \{c, e\}$. $Secondary(\alpha) = \{c, d, e\}$.

Calculating Upper-Bounds and Support Efficiently using Fast Utility Counting (FUC). In the previous paragraphs, 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 6 (Utility-Bin). Let be the set of items I appearing in a database D. A *utility-bin array* U for a 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 stores 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 and the support 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, $U[k] = su(\alpha, k)\ \forall k \in E(\alpha)$.

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 $U[k] = lu(\alpha, k)\ \forall k \in E(\alpha)$.

Calculating the support 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 $U[z] = U[z] + 1$. Thereafter, we have $U[k] = sup(\alpha \cup \{k\})\ \forall k \in E(\alpha)$.

This approach for calculating upper-bounds and the support is highly efficient. For an itemset α, this approach allows to calculate the three upper-bounds

and the support for all single extensions of α in linear time by performing a single (projected) database scan. In terms of memory, 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 greatly reduces memory usage. In our implementation, only four utility-bin arrays are created, to respectively calculate the TWU, sub-tree utility, local utility and support. This is a reason why the memory usage of EFIM-Closed is very low compared to the state-of-the-art CHUD algorithm, as it will be shown in the experimental section. 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.

4.4 Pruning Non Closed HUIs

We now explain the techniques used by EFIM-closed to prune non closed HUIs. To find only CHUIs, a naive approach would be to keep all HUIs found until now into memory. Then, every time that a new HUI is found, the algorithm would compare the HUI with previously found HUIs to determine if (1) the new HUI is included in a previously found HUI or (2) if some previously found HUI(s) are included in the new HUI. The drawback of this approach is that it can consume a large amount of memory if the number of patterns is large, and it becomes very time consuming if a very large number of HUIs is found, because a very large number of comparisons would have to be performed. In this paper, we present new checking mechanisms that can determine if a HUI is closed without having to compare a new pattern with previously found patterns. It is inspired by a similar mechanism used in sequential pattern mining [13]. The mechanism is based on two separate checks, which we respectively name *backward extension checking* check and *forward-extension checking*, and are defined as follows.

Definition 7 (Forward/backward extensions). *Let be an itemset $\beta = \alpha \cup \{i\}$. The itemset β is said to have a forward extension if there exists an item $z \succ i$ such that $z \in E(\beta)$ and $sup(\beta \cup \{z\}) = sup(\beta)$. The itemset β is said to have a* backward extension *if there exists an item $z \prec i$ such that $z \notin \beta$ and $sup(\beta \cup \{z\}) = sup(\beta)$.*

The EFIM-Closed algorithms determine if an itemset is closed as follows.

Property 3 (Identifying non closed itemsets). An itemset $\beta = \alpha \cup \{i\}$ is a CHUI if it is a HUI and it has no backward and forward extension. **Rationale.** By definition, an itemset is not closed if it has a superset $Y = \beta \cup \{z\}$ with the same support. The additional item z can respect either $z \succ i$ or $z \prec i$, which correspond respectively to the cases checked by forward and backward extensions.

The above property only allows to decide if a HUI is closed or not. To also prune the search space of non closed HUIs, the following property is used.

Property 4 (Backward extension pruning). The whole subtree of an itemset $\beta = \alpha \cup \{i\}$ can be pruned during the depth-first search if β has a backward extension.
Rationale. Because there exists a backward extension with an item z, and z thus appear in all transactions where β appears, it follows that all itemsets in the sub-tree of β also have a backward extension with z, and thus are not CHUIs.

Furthermore, we also introduce a second property for pruning the search space that we name *closure jumping.*

Property 5 (Closure jumping property). Let be an itemset β and a projected database β-D. If $sup(\beta) = sup(\beta \cup \{z\})$ for all item $z \in E(\beta)$, then the itemset $\beta \cup E(\beta)$ is the only closed itemset in the sub-tree of β. The whole sub-tree of β can thus be pruned and $\beta \cup E(\beta)$ can be output if it is a HUI.

This property can be easily demonstrated, and is very powerful. It allows to go directly from an itemset β to its closure and prune the rest of its sub-tree.

4.5 The Algorithm

In this subsection, we present the proposed EFIM-Closed algorithm, which combines all the ideas presented in the previous subsections. 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. 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, 3, 8, 11]). 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 $O(nw)$ 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.

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

Algorithm 1: The EFIM-Closed 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, $Primary(\alpha)$, $Secondary(\alpha)$, $minutil$);

database is constructed. If β has a backward extension, no extensions of β will be explored (by Property 4). Otherwise, the projected database of β is scanned to calculate the support, 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 three utility-bin arrays. This allows determining the primary and secondary items of β. If all items that can extend β have the same support as β, the closure jumping optimization is performed to directly output $\beta \cup \bigcup_{z \succ i \wedge z \in E(\alpha)} \{z\}$ if it is a HUI and prune the subtree of β. Otherwise, the *Search* procedure is recursively called with β to continue the depth-first search by extending β. If no extension of β have the same support as β and the utility of β is no less than $minutil$, β is output as a CHUI (by Property 3). Based on properties and theorems presented in previous sections, it can be seen that when EFIM-Closed terminates, all and only the CHUIs have been output.

Complexity. The complexity of EFIM-Closed is briefly analyzed as follows. In terms of time, a $O(nw\ log(nw))$ sort is performed. This cost is negligible since it is performed only once. Then, to process each primary itemset α encountered during the depth-first search, EFIM-Closed performs database projection, transaction merging, backward/forward extension checking and upper-bound calculation in linear time and space ($O(nw)$). Thus, the time complexity of EFIM-Closed is proportional to the number of itemsets in the search space, and it is linear for each itemset.

5 Experimental Results

We performed experiments to evaluate the performance of the proposed EFIM-Closed 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. The performance of EFIM-Closed was compared with the state-of-the-art CHUD algorithm. Moreover, to evaluate the influence of the design decisions in EFIM-

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

1 **foreach** *item* $i \in Primary(\alpha)$ **do**
2 $\quad \beta = \alpha \cup \{i\}$;
3 $\quad$ Scan α-D to calculate $u(\beta)$ and create β-D; `// with transaction merging`
4 $\quad$ **if** β *has no backward extension* **then**
5 $\quad\quad$ Calculate $sup(\beta, z)$, $su(\beta, z)$ and $lu(\beta, z)$ for all item $z \in Secondary(\alpha)$ by scanning β-D once, using three utility-bin arrays;
6 $\quad\quad$ **if** $sup(\beta) = sup(\beta \cup \{z\}) \forall z \succ i \wedge z \in E(\alpha)$ **then**
7 $\quad\quad\quad$ Output $\beta \cup \bigcup_{z \succ i \wedge z \in E(\alpha)} \{z\}$ if it is a HUI; `// closure jumping`
8 $\quad\quad$ **else**
9 $\quad\quad\quad Primary(\beta) = \{z \in Secondary(\alpha) | su(\beta, z) \geq minutil\}$;
10 $\quad\quad\quad Secondary(\beta) = \{z \in Secondary(\alpha) | lu(\beta, z) \geq minutil\}$;
11 $\quad\quad\quad$ `Search` (β, β-D, $Primary(\beta)$, $Secondary(\beta)$, $minutil$);
12 $\quad\quad\quad$ **if** β *has no forward extension and* $u(\beta) \geq minutil$ **then** output β;
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 **end**

Closed, we also compared with a version of EFIM-Closed named EFIM(nop), where transaction merging (HTM), closure jumping, and search space pruning using the sub-tree utility were respectively deactivated, and a version named EFIM(lu), where only the sub-tree utility is desactivated. Algorithms were implemented in Java. Experiments were performed using standard datasets used in the HUIM literature for evaluating HUIM algorithms, namely *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*. These datasets have varied characteristics representing the main types of databases (sparse, dense, long transactions). For these datasets, the number of transactions/number of distinct items/average transaction length are: *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). *Foodmart* contains real external/internal utility values. For other datasets, external and internal utility values have been generated in the $[1, 000]$ and $[1, 5]$ intervals using a log-normal distribution, as done in previous work $[2, 3, 8, 10, 11]$. The datasets and the source code of the compared algorithms can be downloaded at `https://goo.gl/ZaeD6O`.

Influence of the ***minutil*** **threshold on execution time.** 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. Results are shown in Fig. 2. It can be seen that EFIM-Closed clearly outperforms CHUD on all datasets. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart* and *Mushroom*, EFIM-Closed is respectively up to 71 times, 3 times, 36 times, 2 times, 69 times

and 9 times faster than CHUD. The main reasons why EFIM-Closed performs so well are, as we will show in the following experiments, that (1) the proposed sub-tree utility and local-utility upper-bounds allows EFIM-Closed to prune a larger part of the search space compared to CHUD, and that (2) the proposed HTM transaction merging technique greatly reduce the cost of database scans. Beside, the efficient calculation of the proposed upper-bounds and support in linear time using utility-bins also contribute to the time efficiency of EFIM-Closed. A reason why EFIM-Closed is so memory efficient is that it uses a simple database representation, which does not requires to maintain much information in memory (only pointers for pseudo-projections). Moreover, EFIM-Closed is also more efficient because it is a one-phase algorithm (it does not need to maintain candidates in memory), while CHUD is a two-phase algorithm. Lastly, another important characteristic of EFIM-Closed in terms of memory efficiency is that it reuses some of its data structures. As explained in section 4.3, EFIM-Closed uses a very efficient mechanism called Fast Array Counting for calculating upper-bounds. FAC only requires to create four arrays that are then reused to calculate the upper-bounds and support of each itemset considered during the depth-first search.

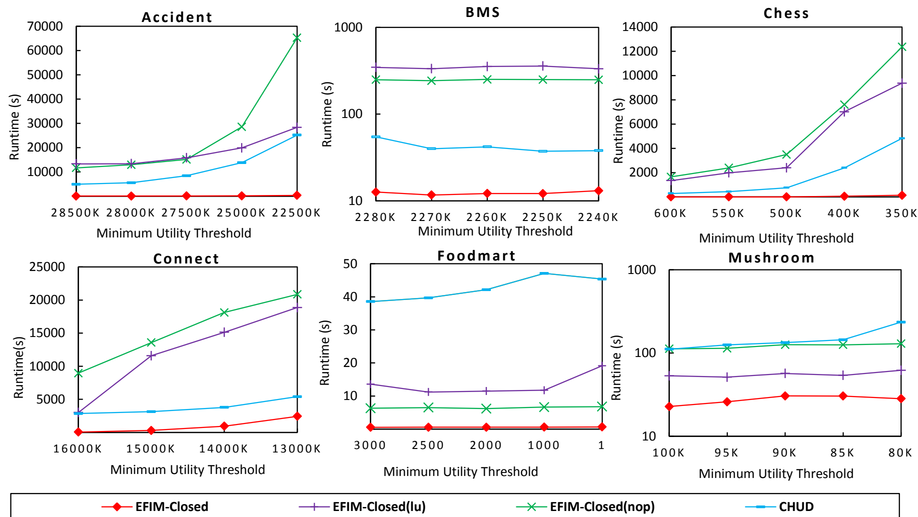

Fig. 2: Execution times on different datasets

Influence of the *minutil* threshold on memory usage. In terms of memory usage, EFIM-Closed clearly outperforms CHUD. The maximum memory usage for EFIM-Closed/CHUD on each dataset are (in megabytes): *Accident* (895 / 2,603), *BMS* (64 / 707), *Chess* (65 / 327), *Connect* (385 / 1,504), *Foodmart* (64 / 215) and *Mushroom* (71 / 1,308). It is also interesting that EFIM-Closed

utilizes less than 100 MB on four out of the six datasets, and never more than 1 GB, while CHUD often exceeds 1 GB.

Influence of transaction merging on execution time. In terms of optimizations, the proposed transaction merging and closure jumping techniques used in EFIM-Closed sometimes greatly increases its performance in terms of execution time. This allows EFIM-Closed to perform very well on dense or large datasets such as *Accidents*, *Chess*,*Chess*, *Connect* and *Mushroom*). For example, for *Accidents* and $minutil = 22500K$, EFIM-Closed terminates in 6 minutes while CHUD terminates in almost 7 hours. On dense datasets or datasets having long transactions, transaction merging and closure jumping is very effective as projected transactions are more likely to be identical. This can be clearly seen by comparing the runtime of EFIM-Closed and EFIM(nop). On *Accidents*, *Chess*, *Connect* and *Mushroom*, EFIM-Closed is up to 183, 90, 9 and 5 times faster than EFIM(nop). For other datasets, transaction merging also reduces execution times but usually by a lesser amount (EFIM-Closed is up to 19, 10 times faster than EFIM(nop) on *BMS* and *Foodmart*). It is also interesting to note that transaction merging could not be implemented efficiently in CHUD because it uses a vertical database representation.

Comparison of the number of visited nodes. We also performed an experiment to compare the ability at pruning the search space of EFIM-Closed to CHUD. For the same datasets and the lowest *minutil* values, the EFIM-Closed / CHUD algorithms visited the following number of nodes: *Accident* (1,341 / 29,932), *BMS* (7 / 27), *Chess* (348,633 / 7,759,252), *Connect* (19,336 / 218,059), *Foodmart* (6,680 / 6,680) and *Mushroom* (8,017 / 17,621). It can be observed that EFIM-Closed is much more effective at pruning the search space than CHUD, thanks to its proposed sub-tree utility and local utility upper-bounds. For example, on *Chess*, EFIM-Closed visits 22 times less nodes.

6 Conclusion

We have presented an efficient algorithm named EFIM-Closed for closed high-utility itemset mining. It relies on two new upper-bounds named *sub-tree utility* and *local utility*, and an array-based utility counting approach named *Fast Utility Counting*. Moreover, to reduce the cost of database scans, EFIM-Closed proposes two efficient techniques named *High-utility Database Projection* and *High-utility Transaction Merging*. Lastly, to discover only closed HUIs, three mechanisms are proposed: (1) forward closure checking, (2) backward closure checking, and (3) closure jumping. Experimental results shows that EFIM-Closed can be up to 71 times faster and consumes up to 18 times less memory than the state-of-the-art CHUD algorithm. Source code and datasets are available as part of the SPMF data mining library [5] at `http://www.philippe-fournier-viger.com/spmf/`. For future work, we will consider extending ideas introduced in EFIM-closed for top-k HUI mining [4], and high-utility sequent pattern and sequential rule mining [15].

Acknowledgement This research was partially supported by National Natural Science Foundation of China (NSFC) under grant No.61503092.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. Int. Conf. Very Large Databases, pp. 487–499, (1994)
2. Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., Lee, Y.-K.: Efficient tree structures for high-utility pattern mining in incremental databases. IEEE Trans. Knowl. Data Eng. 21(12), 1708–1721 (2009)
3. 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)
4. Fournier-Viger, P., Gomariz, A., Gueniche, T., Mwamikazi, E., Thomas, R.: Efficient Mining of Top-K Sequential Patterns. In: Proc. 9th Intern. Conf. on Advanced Data Mining and Applications Part I, pp. 109–120, Springer (2013)
5. Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu., C., Tseng, V. S.: SPMF: a Java Open-Source Pattern Mining Library. Journal of Machine Learning Research (JMLR), 15, pp. 3389-3393 (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. 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)
8. 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)
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)
   Shie, B.-E., Yu, P.S., Tseng, V.S.: Efficient algorithms for mining maximal high utility itemsets from data streams with different models. Expert Syst. Appl. 39(17), pp. 12947-12960 (2012)
10. 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)
11. 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)
12. T. Uno, M. Kiyomi, H. Arimura, "LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," *Proc. ICDM'04 Workshop on Frequent Itemset Mining Implementations*, CEUR, 2004.
13. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. IEEE Trans. on Knowledge Data Engineering 19(8), 10421056 (2007)
14. Yun, U., Ryang, H., Ryu, K. H.: High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates. Expert Syst. with Appl. 41(8), 3861–3878 (2014)
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 (MLDM 2015), pp. 1–15 (2015)