Knowl Inf Syst
DOI 10.1007/s10115-016-0986-0

REGULAR PAPER

EFIM: a fast and memory efficient algorithm for high-utility itemset mining

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

Received: 28 January 2016 / Revised: 23 July 2016 / Accepted: 25 August 2016
© Springer-Verlag London 2016

Abstract In recent years, high-utility itemset mining has emerged as an important data mining task. However, it remains computationally expensive both in terms of runtime and memory consumption. It is thus an important challenge to design more efficient algorithms for this task. In this paper, we address this issue by proposing a novel algorithm named EFIM (EFficient high-utility Itemset Mining), which introduces several new ideas to more efficiently discover high-utility itemsets. EFIM relies on two new upper bounds named *revised 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 named *High-utility Database Projection* and *High-utility Transaction Merging* (HTM), also performed in linear time. An extensive experimental study on various datasets shows that EFIM is in general two to three orders of magnitude faster than the state-of-art algorithms d^{2}HUP, HUI-Miner, HUP-Miner, FHM and UP-Growth+ on dense datasets and performs quite well on sparse datasets. Moreover, a key advantage of EFIM is its low memory consumption.

Keywords Pattern mining · Itemset mining, High-utility mining · Fast Utility Counting, High-utility database merging and projection

✉ Philippe Fournier-Viger
philfv8@yahoo.com

1 Department of Computer Science, University of Moncton, Moncton, NB, Canada

2 School of Natural Sciences and Humanities, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, GD, China

3 School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, GD, China

4 Department of Computer Science, National Chiao Tung University, Hsinchu City, Taiwan

Published online: 02 September 2016

1 Introduction

Frequent Itemset Mining (FIM) [1] is a popular data mining task. Given a transaction database, FIM consists of discovering frequent itemsets, i.e., groups of items (itemsets) appearing frequently in a customer transaction database [1,9,10,28,33]. FIM is essential to many applications. A classical application of FIM is market basket analysis. In this context, frequent itemsets are discovered and then used by retail store managers to co-promote frequently purchased itemsets [1]. Although much work has been done on FIM, a fundamental 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). These two assumptions often do not hold in real applications. For example, consider a database of customer transactions. It is common that a customer will buy several unit of the same product (e.g., a customer may purchase several bottles of juice), and not all items have the same unit profit (e.g., selling a diamond yields more profit than selling a bottle of juice). Traditional FIM algorithms do not consider information about the purchase quantities of items and unit profits of items. Thus, FIM algorithms would discard this information and only find frequent itemsets, rather than finding those yielding a high profit. As a result, many uninteresting frequent itemsets generating a low profit may be discovered, and many rare itemsets generating a high profit may be missed.

To address this issue, the problem of *High-Utility Itemset Mining* (HUIM) has been defined [2,6,11,12,16,18,24,26,29,32]. As opposed to FIM [1,9,10,28], 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). The goal of HUIM is to discover itemsets having a high-utility (a high importance, such as a high profit), that is *High-Utility Itemsets*. High-utility itemset mining has emerged as an important research topic in data mining in recent years and has inspired several other important data mining tasks such as high-utility sequential pattern mining [30,31,34]. Beside market basket analysis, HUIM and its variations have a wide range of applications such as mobile commerce [26], click stream analysis [3,25], biomedicine [17] and cross-marketing [2,26].

The problem of HUIM is widely recognized as more difficult than the problem of FIM. In FIM, the *downward-closure property* states that the support (frequency) of an itemset is *anti-monotonic* [1], that is, supersets of an infrequent itemset are infrequent and subsets of a frequent itemset are frequent. This property, also called the *Apriori property*, is very powerful to prune the search space. But in HUIM, the utility of an itemset is neither monotonic or anti-monotonic. That is, a HUI may have a superset or a subset having a lower, equal or higher utility [2,6,16,26]. Thus, techniques that have been developed in FIM to prune the search space based on the anti-monotonicity of the support cannot be directly applied in HUIM, to prune the search space.

Many studies have been carried to develop efficient HUIM algorithms [2,6,11,12,16,18,24,26,32]. A popular approach to HUIM is to discover high-utility itemsets in two phases using the Transaction-Weighted-Downward Closure model. This approach is adopted by algorithms such as IHUP [2], PB [12], Two-Phase [18], BAHUI [24], UP-Growth and UP-Growth+ [26] and MU-Growth [32]. These algorithms first generate a set of candidate high-utility itemsets by overestimating their utility in Phase 1. Then, in Phase 2, the algorithms scan the database to calculate the exact utilities of candidates and filter low-utility itemsets. However, the two-phase model suffers from the problem of generating a huge amount of candidates in Phase 1 and repeatedly scanning the database to calculate their utility in Phase 2. Recently, to avoid the problem of candidate generation, more efficient approaches were

proposed in the HUI-Miner [16] and d^{2}HUP [15] algorithms to mine high-utility itemsets directly using a single phase, thus avoiding the problem of candidate generation. The d^{2}HUP and HUI-Miner algorithms were reported to be, respectively, up to 10 and 100 times faster than the state-of-the-art two-phase algorithms [15,16]. Then, improved versions of HUI-Miner named HUP-Miner [11] and FHM [6] were proposed. FHM and HUP-Miner were shown to be up to 6 times faster than HUI-Miner 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 computationally expensive both in terms of execution time and memory usage [6,11,16].

In this paper, we address this need for more efficient HUIM algorithms by proposing a novel one-phase algorithm named EFIM (EFficient high-utility Itemset Mining). Authors were invited to publish an extended version in knowledge-based systems. This latter introduces several novel ideas to greatly improve the performance of the task of HUIM both in terms of memory and execution time.

The major contributions of this paper are the following.

- First, we propose a novel one-phase algorithm named EFIM. The overall design of the EFIM algorithm is based on the principle that for each itemset in the search space, all operations for that itemset should be performed in linear time and space.
- Second, the EFIM algorithm introduces two efficient techniques to reduce the cost of database scans named *High-utility Database Projection* (HDP) and *High-utility Transaction Merging* (HTM). These techniques, respectively, perform database projections and merges transactions that are identical in each projected database using a linear time and space implementation. Both techniques reduce the size of the database as larger itemsets are explored, and thus considerably decrease the cost of database scans.
- Third, the proposed EFIM algorithm includes two new upper bounds on the utility of itemsets named the *revised sub-tree utility* and *local utility* to more effectively prune the search space. We show that these upper bounds are more effective at pruning the search space than the TWU and remaining utility upper bounds, commonly used in previous work.
- Fourth, we introduce a novel array-based utility counting technique named *Fast Utility Counting* (FAC) to calculate these upper bounds in linear time and space for all extensions of an itemset.
- Fifth, we conduct an extensive experimental study to compare the performance of EFIM with five state-of-the-art algorithms, namely d^{2}HUP, HUI-Miner, HUP-Miner, FHM and UP-Growth+. Both sparse and dense datasets having various characteristics are used in the experiments in terms of transaction length, number of distinct items and types of data. Results show that EFIM is in general two to three orders of magnitude faster than these algorithms on dense datasets and performs quite well on sparse dataset. Moreover, a key advantage of EFIM is its low memory consumption. It consumes up to eight times less memory than the other algorithms.

Note that this paper is an extension of a conference paper published in the proceedings of the 14th Mexican International Conference on Artificial Intelligence [35]

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.

Table 1 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

2 Problem statement

The problem of high-utility itemset mining was introduced by Yao et al. [29]. It is defined as follows.

Definition 2.1 (*Transaction database*) 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 $D = \{T_1, T_2, \ldots, T_n\}$ 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*. The external utility of an item represents its relative importance to the user. Every item i appearing in a transaction T_c has a positive number $q(i, T_c)$, called its *internal utility*. In the context of market basket analysis, the external utility typically represents an item's unit profit (for example, the sale of 1 unit of bread yields a 1 dollar profit), while the internal utility represents the purchase quantity of an item in a transaction (for example, a customer has bought 2 units of bread).

Example 2.1 Consider the database in Table 1, which will be used as the running example. It contains five transactions $(T_1, T_2, \ldots, T_5)$. Transaction T_2 indicates that items a, c, e and g appear in this transaction with an internal utility (e.g., purchase quantity) of, respectively, 2, 6, 2 and 5. Table 2 indicates that the external utility (e.g., unit profit) of these items are, respectively, 5, 1, 3 and 1.

Definition 2.2 (*Utility of an item*) The utility of an item i in a transaction T_c is denoted as $u(i, T_c)$ and defined as $p(i) \times q(i, T_c)$. It represents the profit generated by the sale of the item i in the transaction T_c.

Example 2.2 The utility of item a in T_2 is $u(a, T_2) = 5 \times 2 = 10$.

Definition 2.3 (*Utility of an itemset in a transaction*) 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 $X \subseteq T_c$. Otherwise $u(X, T_c) = 0$.

Example 2.3 The utility of the itemset $\{a, c\}$ in T_2 is $u(\{a, c\}, T_2) = u(a, T_2) + u(c, T_2) = 5 \times 2 + 1 \times 6 = 16$.

Definition 2.4 (*Utility of an itemset in a database*) 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. It represents the profit generated by the sale of the itemset X in the transaction T_c.

Example 2.4 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 2.5 (*High-utility itemset*) 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*.

Definition 2.6 (*Problem definition*) The *problem of high-utility itemset mining* is to discover all high-utility itemsets, given a threshold $minutil$ set by the user. For the application of market basket analysis, the problem of high-utility itemset mining can be interpreted as finding all sets of items that have generated a profit not less than $minutil$.

Example 2.5 If $minutil = 30$, the high-utility itemsets in the database of the running example are listed in Table 3.

It is interesting to note that FIM is the special case of the problem of HUIM where all internal utility and external utility values are set to 1 (i.e., all purchase quantities and unit profit values are assumed to be equal).

3 Related work

HUIM is widely recognized as a much harder problem than FIM since the utility measure is not monotonic or anti-monotonic [2,18,26], 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. As a result, the first algorithm for high-utility itemset mining named UMining was not a complete algorithm (it could miss some high-utility itemsets by pruning the search space). To circumvent the fact that the utility is not anti-monotonic and to find the full set of high-utility itemsets, several HUIM algorithms used a measure called the *Transaction-Weighted Utilization* (TWU) measure [2,12,18,24,26,32], which is an upper bound on the utility of itemsets, and is anti-monotonic. Calculating an upper bound such as the TWU on the utility of itemsets is interesting because if that upper bound is lower than

Table 3 High-Utility itemsets for $minutil = 30$

Itemset	Utility
$\{b, d\}$	30
$\{a, c, e\}$	31
$\{b, c, d\}$	34
$\{b, c, e\}$	31
$\{b, d, e\}$	36
$\{b, c, d, e\}$	40
$\{a, b, c, d, e, f\}$	30

$minutil$ for some itemsets, it can be concluded that these itemsets are not high-utility itemsets (since the TWU is an upper bound on their utility values). Thus, the TWU can be used to safely prune itemsets in the search space. The TWU upper bound is defined as follows.

Definition 3.1 (*Transaction utility*) The *transaction utility* of a transaction T_c is the sum of the utilities of items from T_c in that transaction, i.e., $TU(T_c) = \sum_{x \in T_c} u(x, T_c)$. In other words, the transaction utility of a transaction T_c is the total profit generated by that transaction.

Definition 3.2 (*Transaction-Weighted Utilization*) Let there be an itemset X. The *Transaction-Weighted Utilization* (TWU) of X is defined as the sum of the transaction utilities of transactions containing X and is denoted as $TWU(X)$. Formally, $TWU(X) = \sum_{T_c \in g(X)} TU(T_c)$. The TWU represents the total profit generated by the transactions containing the itemset X.

Example 3.1 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. The TWU of single items a, b, c, d, e, f and g are, respectively, 65, 61, 96, 58, 88, 30 and 38. Consider item a. $TWU(a) = TU(T_1) + TU(T_2) + TU(T_3) = 8 + 27 + 30 = 65$.

As previously mentioned, the TWU is interesting because it is an upper bound on the utility of itemsets and can thus be used to prune the search space. The following properties of the TWU have been proposed to prune the search space.

Property 3.1 (Overestimation using the TWU) *Let be an itemset X. The TWU of X is no less than its utility ($TWU(X) \geq u(X)$). Moreover, the TWU of X is no less than the utility of its supersets ($TWU(X) \geq u(Y) \forall Y \supset X$). The proof is provided in [18]. Intuitively, since the TWU of X is the sum of the profit of transactions where X appears, the TWU must be greater or equal to the utility of X and any of its supersets.*

Property 3.2 (Pruning the search space using the TWU) *For any itemset X, if $TWU(X) < minutil$, then X is a low-utility itemset as well as all its supersets. This directly follows from the previous property [18].*

Algorithms such as IHUP [2], PB [12], Two-Phase [18], BAHUI [24], UP-Growth and UP-Growth+ [26] and MU-Growth [32] utilize Property 3.2 as main property to prune the search space. They operate in two phases. In the first phase, they identify candidate high-utility itemsets by calculating their TWUs to prune the search space. If an itemset has a TWU greater than $minutil$, it may be a high-utility itemsets and is thus considered as a candidate. If an itemset has a TWU lower than $minutil$, it is discarded, as it cannot be a high-utility itemset. Then, in the second phase, these algorithms scan the database to calculate the exact utility of all candidates to filter those that are low-utility itemsets. Among the two-phase algorithms, UP-Growth is one of the fastest. It uses a tree-based algorithm inspired by the FP-Growth algorithm for FIM [9]. It was shown to be up to 1000 times faster than Two-Phase and IHUP. More recent two-phase algorithms such as PB, BAHUI and MU-Growth have introduced various optimizations and different design but only provide a small speed improvement over Two-Phase or UP-Growth (MU-Growth is only up to 15 times faster than UP-Growth).

Recently, algorithms that mine high-utility itemsets using a single phase were proposed to avoid the problem of candidate generation. These algorithms use upper bounds that are tighter than the TWU to prune the search space and can immediately obtain the exact utility of any itemset to decide if it should be output. The d^{2}HUP and HUI-Miner algorithms were reported to be, respectively, up to 10 and 100 times faster than UP-Growth [15,16]. Then, improved

versions of HUI-Miner named HUP-Miner [11] and FHM [6] were proposed to reduce the number of join operations performed by HUI-Miner. FHM introduces a novel strategy that consists of precalculating the TWU of all pairs of items to prune the search space, while HUP-Miner introduces the idea of partitioning the database and a mechanism called LA-Prune to stop calculating the upper bound of an itemset early. FHM and HUP-Miner were shown to be up to 6 times faster than HUI-Miner and are to our knowledge the current best algorithms for HUIM. The HUI-Miner, HUP-Miner and FHM are vertical algorithms. They associate a structure named *utility-list* [6,16] to each itemset. Utility-lists allow calculating the utility of an itemset by making join operations with utility-lists of smaller itemsets. Moreover, utility-lists also allow calculating an upper bound called *remaining utility* on the utilities of its supersets, to prune the search space. This upper bound is equivalent to the upper bound used by the d^{2}HUP algorithm. Similar to the TWU, the *remaining utility* upper bound is used to prune the search space. The next paragraphs introduces this upper bound and the utility-list structure. Then, an explanation of how it is used to prune the search space is provided.

Definition 3.3 (*Remaining utility*) Let $\succ$ be a total order on items from I (e.g., the lexicographical order), and X be an itemset. The *remaining utility* of X in a transaction T_c is defined as $re(X, T_c) = \sum_{i \in T_c \wedge i \succ x \forall x \in X} u(i, T_c)$. Since high-utility itemset mining algorithms append items one at a time to itemsets to generate larger itemsets by following the total order $\succ$, the remaining utility of an itemset X in a transaction T_c can be interpreted as the amount of profit that the itemset X could gain if other items from T_c were appended to X (according to the $\succ$ order).

Definition 3.4 (*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 ($re(X, T_c)$). The $iutil$ element of an itemset X in a transaction T_c can be interpreted as the profit generated by X in that transaction, while the remaining utility of X in T_c represents the profits of other items that could be appended to X in that transaction when following the $\succ$ order.

Example 3.2 For the running example described in this paper, assume that $\succ$ is 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 the utility-lists of patterns containing single items. Then, utility-lists of larger patterns are constructed by joining utility-lists of smaller patterns (see [11,16] for details). Pruning the search space is done using the following properties.

Definition 3.5 (*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. The proof that the sum of $iutil$ and $rutil$ values of an itemset X is an upper bound on the utility of X and its extensions is provided in [16]. Intuitively, this upper bound is the total profit generated by X (the $iutil$ values) plus the sum of the profit for items that could still be appended to X when following the $\succ$ order for appending items (the sum of $rutil$ values).

Example 3.3 For example, $re(\{a, e\}) = u(\{a, e\}) + re(\{a, e\}) = u(\{a, e\}, T_2) + re(\{a, e\}, T_2) + u(\{a, e\}, T_3) + re(\{a, e\}, T_3) = (16 + 8) + (8 + 5) = 37$. Since this sum is an upper bound on the utility of $\{a, e\}$ and its extensions, it can be concluded that the utility of $\{a, e\}$ and its extensions cannot be greater than 37.

Property 3.3 (Pruning the search space using utility-lists) *If* $reu(X) < minutil$*, then X is a low-utility itemset as well as all its extensions* [16]. *These latter can thus be pruned. This property directly follows from the previous definition. This property is used by the* d^2HUP*, HUI-Miner and FHM algorithms to prune the search space.*

One-phase algorithms are faster than previous algorithms because they discover itemsets in one phase, thus avoiding the problem of candidate generation found in two-phase algorithms. Moreover, one-phase algorithms have introduced the remaining utility upper bound, which is a tighter upper bound on the utility of itemsets than the TWU. Thus, it can be used to prune a larger part of the search space. However, mining HUIs remains a very computationally expensive task both in terms of memory consumption and execution time. For example, HUI-Miner, HUP-Miner and FHM still suffer from a high space and time complexity to process each itemset in the search space. 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). Creating the lists for itemsets can thus require a significant amount of memory, since more than one list need to be maintained in memory during the depth-first search. Moreover, in terms of execution time, the complexity of building a utility-list is also high [6]. In general, it requires to join three utility-lists of smaller itemsets. A naive implementation requires $O(n^3)$ time in th worst case, while a better implementation may require $O(3n)$ time. FHM and HUP-Miner introduce strategies to reduce the number of join operations performed by HUI-Miner. However, joining utility-lists remains the main performance bottleneck in terms of execution, and storing utility-lists remains the main issue in terms of memory consumption [6]. Lastly, another limitation of list-based algorithms such as HUI-Miner and HUP-Miner is that they may consider itemsets not appearing in the database, as they explore the search space of itemsets by combining smaller itemsets, without scanning the database. The d^2HUP algorithm [15] uses a pattern-growth approach to avoid considering itemsets not appearing in the database, but uses a hyper-structure-based approach that can still consume quite a significant amount of memory (as it will be shown in the experimental evaluation of this paper).

To summarize, all the above algorithms except Two-Phase utilize a depth-first search to explore the search space of high-utility itemsets, as it is generally more efficient than using a breadth-first search in the field of itemset mining [9,10,28,33]. High-utility itemset mining algorithms differ in many aspects: the type of database representation that is used (vertical vs horizontal), whether they perform a single phase or two phases, what kind of data structure is used to maintain information about transactions and itemsets (e.g., hyperlink structures, utility-lists, tree-based structures), the choice of upper bound(s) and strategies to prune the search space and how these upper bounds are calculated.

4 The EFIM algorithm

As stated in the introduction, the goal of this paper is to improve the efficiency of HUIM. In this section, we present our proposal, the EFIM algorithm. Four key design ideas have guided the overall design of EFIM:

- It is necessary to design a one-phase algorithm to avoid the problem of candidate generation of two-phase algorithms.
- It is necessary to use a "pattern-growth approach" to avoid considering patterns that may not appear in the database, and techniques should be used to reduce the cost of database scans.

– It is necessary to use a tighter upper bound to prune the search space of itemsets more efficiently. This upper bound should be tighter than the remaining utility upper bound used in the state-of-the-art algorithms.
– It is necessary to perform low-complexity operations for processing each itemset in the search space, both in terms of space and time. In EFIM, we use the very strict 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. Section 4.1 introduces preliminary definitions related to the depth-first search of itemsets. Sections 4.2 and 4.3, respectively, explain how EFIM reduces the cost of database scans using two novel efficient techniques named *High-utility Database Projection* (HDP) and *High-utility Transaction Merging* (HTM), performed in linear time and space. Section 4.4 presents two new upper bounds used by EFIM to prune the search space. Section 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, Sect. 4.6 gives the pseudocode of EFIM, discusses the overall complexity of the algorithm, and briefly presents an example of how the algorithm is applied for the running example.

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* [22]. 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 [2,6,16,26]. However, we henceforth assume that $\succ$ is the lexicographical order in the running example, to make the examples easier to understand, for the convenience of the reader. We next introduce three definitions related to the depth-first search exploration of itemsets.

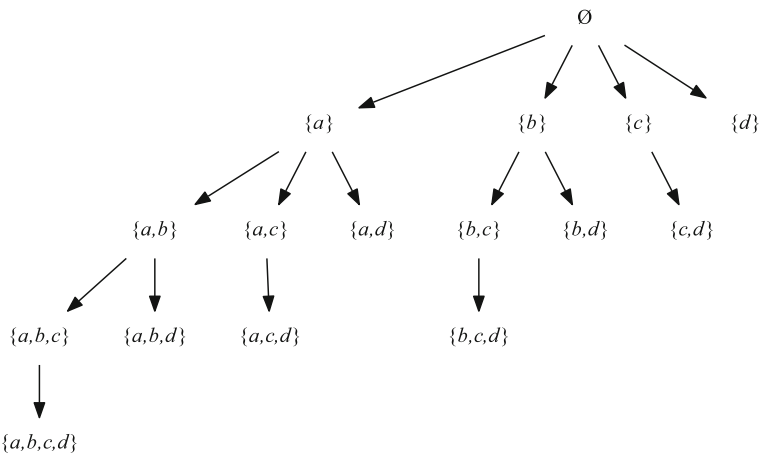

Fig. 1 Set-enumeration tree for $I = \{a, b, c, d\}$

Definition 4.1 (*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 4.2 (*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 $W \neq \emptyset$.

Definition 4.3 (*Single-item extension of an itemset*) Let be an itemset α. 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 $z \in E(\alpha)$.

Example 4.1 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 high-utility database projection (HDP)

As we will later 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. In EFIM this is performed by a novel technique called *High-utility Database Projection* (HDP).

HDP 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 4.4 (*Projected transaction*) The *projection of a transaction T using an itemset α* is denoted as $\alpha - \mathrm{T}$ and defined as $\alpha - \mathrm{T} = \{i | i \in T \wedge i \in E(\alpha)\}$.

Definition 4.5 (*Projected database*) The *projection of a database D using an itemset α* is denoted as $\alpha - \mathrm{D}$ and defined as the multiset $\alpha - \mathrm{D} = \{\alpha - \mathrm{T} | T \in D \wedge \alpha - \mathrm{T} \neq \emptyset\}$.

Example 4.2 Consider database D of the running example and $\alpha = \{b\}$. The projected database $\alpha - D$ contains three transactions: $\alpha - T_3 = \{c, d, e, f\}$, $\alpha - T_4 = \{c, d, e\}$ and $\alpha - T_5 = \{c, e, g\}$.

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. The efficient approach used in EFIM called HDP is performed as follows. It requires to sort items in each transaction in the original database according to the $\succ$ total order beforehand. 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 calculating the projection $\alpha - \mathrm{D}$ of a database D is linear in time and space ($O(nw)$ where n and w are, respectively, the number of transactions and the average transaction length). However, as larger itemsets are explored, the size of projected databases decrease.

The proposed database projection technique is a generalization of the concept of database projection used in frequent pattern mining [20,28] for the case of transactions with internal/external utility values. Note that FP-Growth-based HUIM algorithms [2,12,26,32] and

hyperlink-based HUIM algorithms [15] also perform some form of projections but differently than in the EFIM algorithm since they use different database representations (fp-tree and hyperlink representations, respectively).

4.3 Reducing the cost of database scans by high-utility transaction merging (HTM)

To further reduce the cost of database scans, EFIM also introduces an efficient transaction merging technique named *High-utilty Transaction Merging* (HTM). HTM is based on the observation that transaction databases often contain identical transactions. The technique consists of identifying these transactions and then replace them with single transactions, while combining their utilities.

Definition 4.6 (*Identical transactions*) A transaction T_a is *identical to* a transaction T_b if it contains the same items as T_b (i.e., $T_a = T_b$). It is important to note that in this definition, two identical transactions are not required to have the same internal utility values.

Definition 4.7 (*Transaction merging*) *Transaction merging* consists of replacing a set of identical transactions $Tr_1, Tr_2, \ldots, Tr_m$ in a database D by a single new transaction $T_M = Tr_1 = Tr_2 = \cdots = Tr_m$ where the quantity of each item $i \in T_M$ is defined as $q(i, T_M) = \sum_{k=1\ldots m} q(i, Tr_k)$.

Merging identical transactions reduce the size of the database. But this reduction is small if the database contains few identical transactions. For example, in the database of the running example, no transactions can be merged. 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 4.8 (*Projected transaction merging*) *Projected transaction merging* consists of replacing a set of identical transactions $Tr_1, Tr_2, \ldots, Tr_m$ in a projected database $\alpha - \mathrm{D}$ by a single new transaction $T_M = Tr_1 = Tr_2 = \cdots = Tr_m$ where the quantity of each item $i \in T_M$ is defined as $q(i, T_M) = \sum_{k=1\ldots m} q(i, Tr_k)$.

Example 4.3 Consider database D of our running example and $\alpha = \{c\}$. The projected database $\alpha - D$ contains transactions $\alpha - T_1 = \{d\}$, $\alpha - T_2 = \{e, g\}$, $\alpha - T_3 = \{d, e, f\}$, $\alpha - T_4 = \{d, e\}$ and $\alpha - T_5 = \{e, g\}$. Transactions $\alpha - T_2$ and $\alpha - T_5$ 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 because it requires $O((nw)^2)$ time. To find identical transactions in $O(nw)$ time, we propose the following novel approach. We initially sort the original database according to a new total order $\succ_T$ on transactions. Sorting is achieved in $O(nw\ log(nw))$ time. However, this cost is generally negligible compared to the other operations performed by the algorithm because it is performed only once.

Definition 4.9 (*Total order on transactions*) The $\succ_T$ order is defined as the lexicographical order when the transactions are read backwards. Formally, let there be two transactions $T_a = \{i_1, i_2, \ldots, i_m\}$ and $T_b = \{j_1, j_2, \ldots, j_k\}$. 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 the TID of 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 $T_a \succ_T T_b$.

Example 4.4 Consider three 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.

Property 4.1 (Transaction order in an $\succ_T$ sorted database) *Let there be a $\succ_T$ sorted database D and an itemset α. Identical transactions appear consecutively in the projected database $\alpha - D$.*

Proof Because (1) transactions are sorted in lexicographical order when read backwards and (2) projections always removes the smallest items of a transaction according to the lexicographical order, it is clear that the property holds.

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, a (projected) database can be scanned only once to merge all identical transactions in the database. During this scan, each transaction is only compared with the next one to determine if they should be merged. Thus, the number of comparisons is $n - 1$, where n is the number of transactions. Each comparison between two transactions can be performed in linear time using a two-way comparison. Thus, the overall cost of merging all transactions in a projected database is $O(nw)$

This efficient implementation of transaction merging has to our knowledge not been used in previous work. In particular, it is interesting to note that transaction merging as proposed in EFIM is not performed in any other one-phase HUIM algorithms. The reason is that it cannot be implemented efficiently in utility-list-based algorithms such as HUP-Miner, HUI-Miner and FHM, because these latter use a vertical database representation, rather than a horizontal representation. Moreover, transaction merging cannot be implemented easily in hyperlink-based algorithms such as d^{2}HUP, due to the representation of projected databases using hyperlinks over the original database.

4.4 Pruning the search space using sub-tree utility and local utility upper bounds

The previous subsection has explained techniques used in the proposed EFIM algorithm for reducing the cost of database scans. This subsection now discusses another key issue for designing an efficient HUIM algorithm, which is to design an effective mechanism for pruning itemsets in the search space. For this purpose, we introduce in EFIM two new upper bounds on the utility of itemsets named *revised sub-tree utility* and *local utility*. As we will explain, these upper bounds have similarities to the remaining utility and TWU upper bounds. But a key difference with these upper bounds is that the proposed upper bounds are defined w.r.t the sub-tree of an itemset α in the search-enumeration tree. Moreover, as we will explain, some items are ignored when calculating these upper bounds. This makes the proposed upper bounds more tight, and thus the proposed algorithm more effective at pruning the search space.

The proposed local utility upper bound is defined as follows.

Definition 4.10 (*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 4.5 Consider the running example and $\alpha = \{a\}$. We have that $lu(\alpha, c) = (8+27+30) = 65$, $lu(\alpha, d) = 30$ and $lu(\alpha, e) = 57$.

The following theorem of the local utility is proposed in EFIM to prune the search space.

Property 4.2 (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.*

Proof Let Y denote the itemset $\alpha \cup \{z\}$. The utility of Z is equal to $u(Z) = \sum_{T \in g(Z)} u(Z, T) = \sum_{T \in g(Z)} [u(\alpha, T) + u(Z \backslash \alpha, T)]$. The local utility of α w.r.t to z is equal to $lu(\alpha, z) = \sum_{T \in g(Y)} [u(\alpha, T) + re(\alpha, T)]$. Because $g(Z) \subseteq g(Y)$ and $Z \backslash Y \subseteq E(\alpha)$, it follows that $u(Z \backslash \alpha, T) \leq re(\alpha, T)$ and thus that $lu(\alpha, z) \geq u(Z)$. □

Example 4.1 (Pruning an item in 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.

Thus, by using Theorem 4.1, some items can be pruned from all sub-trees of an itemset α, which reduces the number of itemsets to be considered. To further reduce the search space, we also identify whole sub-trees of α that can be pruned by proposing another upper bound named the sub-tree utility, which is defined as follows.

Definition 4.11 (*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 4.6 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.

Property 4.3 (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 $\alpha \cup \{z\}$.*

Proof Suppose α is an itemset and $Y = \alpha \cup \{z\}$ is an extension of α. The utility of Y is equal to $u(Y) = \sum_{T \in g(Y)} u(Y, T)$. The sub-tree utility of α w.r.t. z is equal to $su(\alpha, z) = \sum_{T \in g(Y)} [u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\})} u(i, T)] = \sum_{T \in g(Y)} u(Y, T) + \sum_{T \in g(Y)} \sum_{i \in T \wedge i \in E(\alpha \cup \{z\})} u(i, T) \geq \sum_{T \in g(Y)} u(Y, T)$. Thus, $su(\alpha, z) \geq u(Y)$. Now let's consider another itemset Z that is an extension of Y. The utility of Z is equal to $u(Z) = \sum_{T \in g(Z)} u(Y, T) + \sum_{T \in g(Z)} u(Z \backslash Y, T)$. Because $g(Z) \subseteq g(Y)$ and $Z \backslash Y \subseteq E(\alpha)$, it follows that $su(\alpha, z) \geq u(Z)$.

Example 4.2 (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.

The relationships between the proposed upper bounds and the main ones used in previous work are the following.

Property 4.4 (Relationships between upper bounds) *Let be an itemset α, an item z and an itemset $Y = \alpha \cup \{z\}$. The relationship $TWU(Y) \geq lu(\alpha, z) \geq reu(Y) = su(\alpha, z)$ holds.*

Proof $TWU(Y) = \sum_{T \in g(Y)} TU(T)$ and $lu(\alpha \cup \{z\}) = \sum_{T \in g(Y)} [u(\alpha, T) + re(\alpha, T)]$. Since $u(\alpha, T) + re(\alpha, T)$ cannot be greater than $TU(T)$ for any transaction T, the relationship $TWU(Y) \geq lu(\alpha \cup \{z\})$ holds. Moreover, because $reu(Y) = \sum_{T \in g(Y)} [u(Y, T) + re(\alpha \cup \{z\}, T)]$ and $re(\alpha, T) \geq re(\alpha \cup \{z\}, T)$, the relationship $lu(\alpha, z) \geq reu(Y)$ holds. Lastly, by definition $su(\alpha, z) = \sum_{T \in g(Y)} [u(Y, T) + \sum_{i \in T \wedge i \in E(Y)} u(i, T)]$ and $re(Y, T)$ is equal to $\sum_{i \in T \wedge i \in E(Y)} u(i, T)$, the relationship $reu(Y) = su(\alpha, z)$ 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.

About the *su* upper bound, one can ask what is the difference between this upper bound and the *reu* upper bound of HUI-Miner and FHM since they are mathematically equivalent. The major difference between the remaining utility upper bound and the proposed *su* upper bound is that the *su* upper bound 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. This is illustrated in Fig. 2, which compares the nodes pruned in the sub-tree of Y using the *su* and *reu* upper bounds. Thus, as explained here, using *su* instead of *reu* upper bound, is more effective for pruning the search space.

Moreover, we make the *su* upper bound even tighter by redefining it as follows. This leads to the final *revised sub-tree utility upper bound* used in the proposed EFIM algorithm.

Definition 4.12 (*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 $lu(\alpha, z) \geq su(\alpha, z)$, $Primary(\alpha) \subseteq Secondary(\alpha)$.

Example 4.7 Consider the running example and $\alpha = \{a\}$. $Primary(\alpha) = \{c, e\}$. $Secondary(\alpha) = \{c, d, e\}$. This means that w.r.t. α, only the sub-trees rooted at nodes

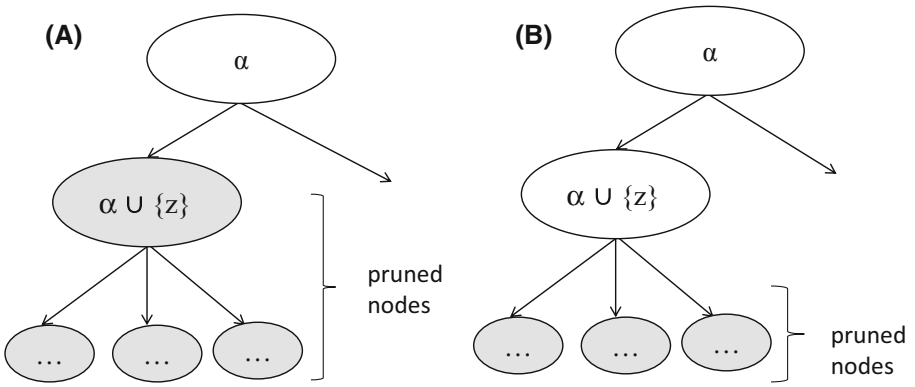

Fig. 2 Comparison of the *su* and *reu* upper bounds

$\alpha \cup \{c\}$ and $\alpha \cup \{e\}$ should be explored. Furthermore, in these subtrees, no items other than c, d and e should be considered.

The redefined (tighter) su upper bound is defined as follows.

Definition 4.13 (*Redefined Sub-tree utility*) Let be an itemset α and an item z. The redefined sub-tree utility of item z w.r.t. itemset α is defined as: $su(\alpha, z) = \sum_{T \in g(\alpha \cup \{z\})}[u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \in E(\alpha \cup \{z\}) \wedge \ i \in Secondary(\alpha)} u(i, T)]$.

The difference between the su upper bound and the redefined su upper bound is that in the latter, items not in $Secondary(\alpha)$ will not be included in the calculation of the su upper bound. Thus, this redefined upper bound is always less than or equal to the original su upper bound and the reu upper bound. It can be easily proven that the redefined su upper bound preserves the pruning Theorem of the su upper bound. In the rest of the paper, it is assumed that the redefined su upper bound is used instead of the original su upper bound.

We also illustrate the difference between the reu and (revised) su upper bound with an example.

Example 4.8 Consider the lexicographical order, $minutil = 40$, and the itemset $\{b\}$. The utility of $\{b\}$ is 16 but it is necessary to explore its extensions since $su(\{b\}) \geq minutil$. The itemset $\{b\}$ can be extended with items $E(\{b\}) = \{c, d, e, f, g\}$. The local utilities of these items are, respectively: $lu(\alpha, \{c\}) = 56$, $lu(\alpha, \{d\}) = 45$, $lu(\alpha, \{e\}) = 56$, $lu(\alpha, \{f\}) = 25$, and $lu(\alpha, \{g\}) = 11$. Because f and g have local utilities lower than $minutil$, they will be excluded from the calculation of the (revised) sub-tree utility of any extensions of b. For example, consider itemset $\{b, d\}$ that extends itemset $\{b\}$. The remaining utility of $\{b, d\}$ is $reu(\{b, d\}) = 42$, while the sub-tree utility is $su(\{b, d\}) = 36$. By using the sub-tree utility for pruning the search space, the whole sub-tree of $\{b, d\}$ can be pruned (including node $\{b, d\}$) because $su(\{b, d\}) < minutil$, while using the remaining utility, $\{b, d\}$ and all its child nodes will need to be explored, because $reu(\{b, d\}) > minutil$. This simple example shows the effectiveness of the sub-tree utility for pruning the search space.

Lastly, it is interesting to note that the redefined su upper bound cannot be applied in vertical algorithms such as HUI-Miner and FHM, since transactions are not represented explicitly (these algorithms do not perform database scans once their utility-lists have been built). These latter algorithms store the remaining utility of an itemset in each transaction in their utility-list structure but cannot reduce the remaining utility at further stages of the depth-first search, as it would require to scan the original transactions. Having explained the proposed upper bounds used in the EFIM algorithm, the next section explains the novel method used in EFIM to calculate its upper bounds efficiently both in terms of time and space, using its horizontal database representation.

4.5 Calculating upper bounds efficiently using fast utility counting (FUC)

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 4.14 (*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 is used to store a utility value (an integer in our implementation, initialized to 0).

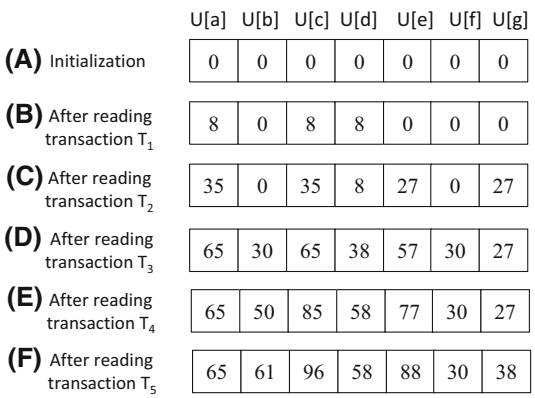

Fig. 3 Calculating the TWU using a utility-bin array

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 $TWU(k)$.

Example 4.9 For example, consider the database of the running example. In this example $I = \{a, b, c, d, e, f, g\}$. An utility-bin array U is constructed with 7 bins since there are 7 items, and it is filled with zeros, as illustrated in Fig. 3 A. Then, the database is scanned one transaction at a time. The first transaction is T_1, which has a transaction utility of 8. Because items a, c and d appear in transaction T_1, the value 8 is added to the utility-bins $U[a]$, $U[c]$ and $U[d]$. The result is shown in Fig. 3 B. Then, the next transaction T_2 is read, which has a transaction utility of 27. Because items a, c, e and g appear in transaction T_2, the value 27 is added to the utility-bins $U[a]$, $U[c]$ and $U[d]$. The result is shown in Fig. 3 C. Then, the same process is repeated for the remaining transactions. The content of the utility-bin array after reading transactions T_3, T_4 and T_5 are, respectively, shown in Fig. 3 D, E, and F. After the last transaction has been read, it is found that the TWU of items a, b, c, d, e, f and g are, respectively, 65, 61, 96, 58, 88, 30 and 38, according to their respective entries in the utility-bin array.

Calculating the sub-tree utility w.r.t. an itemset α Similarly, a utility-bin array can also be used to calculate the sub-tree utility efficiently. A utility-bin array U is first initialized by filling all bins with zeros. 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 $U[z] = U[z] + u(\alpha, T) + u(z, T) + \sum_{i \in T \wedge i \succ z \wedge i \in Secondary(\alpha)} u(i, T)$. When the last transaction has been processed, we have $U[k] = su(\alpha, k)\ \forall k \in E(\alpha)$.

Calculating the local utility w.r.t. an itemset α A utility-bin array U is initialized by filling all bins with zeros. 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 $U[z] = U[z] + u(\alpha, T) + re(\alpha, T)$. When the last transaction has been processed, we have $U[k] = lu(\alpha, k)\ \forall k \in E(\alpha)$.

This approach for calculating upper bounds is highly efficient. For an itemset α, this approach allows to calculate the three upper bounds for all single extensions of α in linear time by performing a single (projected) database scan. In comparison, in the HUI-Miner,

FHM and HUP-Miner algorithms, calculating upper bounds is done one itemset at a time by performing a costly join operation of up to three utility-list for each itemset, which may use up to $O(3n)$ time.

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 three utility-bin arrays are created, to, respectively, calculate the TWU, sub-tree utility and local utility. This is one of the reasons why the memory usage of EFIM is very low compared to other algorithms, as it will be shown in Experimental section. For example, the same utility-bin array can be reused to calculate the sub-tree utility for any itemset in the search space. Thus, no additional memory is required during the depth-first search to calculate the upper bounds once the three initial utility-bin arrays have been created.
- 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 (line 1). The algorithm then scans the database once to calculate the local utility of each item w.r.t. α, using a utility-bin array (line 2). 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 (line 3). Then, these items are sorted by ascending order of TWU and that order is thereafter used as the $\succ$ order (as suggested in [2,6,16]) (line 4). 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 4.1 (line 5). At the same time, items in each transaction are sorted according to $\succ$, and 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 (line 6). 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 (line 7 and 8). Thereafter, the algorithm calls the recursive procedure $Search$ to perform the depth-first search starting from α (line 9).

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 4.2) (line 1 to 9). 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 (line 3). Note that transaction merging is performed while the β projected database is constructed. If the utility of β is no less than $minutil$, β is output as a high-utility

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, sort items in each transaction according to $\succ$, 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` $(\alpha, D, Primary(\alpha), Secondary(\alpha), minutil)$;

itemset (line 4). 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 (line 5). This allows determining the primary and secondary items of β (line 6 and 7). Then, the *Search* procedure is recursively called with β to continue the depth-first search by extending β (line 8). 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.

Algorithm 2: The *Search* procedure

input : α: an itemset, $\alpha - $ 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 $\alpha - $ D to calculate $u(\beta)$ and create $\beta - $ D; `// uses transaction merging`
4 $\quad$ **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 $\beta - $ D once, using two utility-bin arrays;
5 $\quad Primary(\beta) = \{z \in Secondary(\alpha) | su(\beta, z) \geq minutil\}$;
6 $\quad Secondary(\beta) = \{z \in Secondary(\alpha) | lu(\beta, z) \geq minutil\}$;
7 $\quad$ `Search` $(\beta, \beta - $ D, $Primary(\beta), Secondary(\beta), minutil)$;
8 **end**

4.7 A detailed example

This subsection now provides a detailed example of how the EFIM algorithm is applied for the running example. Consider the database of the running example and $minutil = 30$. The main procedure (Algorithm 1) is applied as follows. Initially, α is set to the $\emptyset$ (line 1), and the database is scanned to calculate $lu(\alpha, i)$ for all items using a utility-bin array (line 2). This is performed as previously illustrated in Fig. 3. The result is that the lu values of items are $lu(\alpha, a) = 65$, $lu(\alpha, b) = 61$, $lu(\alpha, c) = 96$, $lu(\alpha, d) = 58$, $lu(\alpha, e) = 88$, $lu(\alpha, f) = 30$ and $lu(\alpha, g) = 38$. These values are equal to the TWU since the lu upper bound is equal to the TWU when $\alpha = \emptyset$. Based on these values, the set of secondary items is $Secondary(\alpha) = \{a, b, c, d, e, f\}$. Then, the order $\succ$ is established at the order of ascending TWU values (line 4), that is $f \prec g \prec d \prec b \prec a \prec e \prec c$. After that, the database is scanned to remove items not in $Secondary(\alpha)$ (line 5). Since all items are in this set, the database is unchanged. At

Table 4 The database D with items sorted by $\succ$

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

Table 5 Projected database $\{a\}$-D (when transaction merging is applied)

TID	Transaction
T_3	$(e, 1)(c, 1)$
T_2	$(e, 2)(c, 6)$
T_1	$(c, 1)$

the same time, items in transactions are sorted according to $\succ$. Then, transactions are sorted according to the total order $\succ_T$ (line 6). The result is: $T_3 \prec_T T_2 \prec_T T_4 \prec_T T_5 \prec_T T_1$. The resulting database after these transformation is listed in Table 4. Then, the database is scanned again to calculate the sub-tree utility of all items with respect to α (line 7). The result is: $su(\alpha, a) = 65$, $su(\alpha, b) = 56$, $su(\alpha, c) = 54$, $su(\alpha, d) = 25$, $su(\alpha, e) = 27$, $su(\alpha, f) = 5$ and $su(\alpha, g) = 7$. Thus, the set $Primary(\alpha) = \{b, a, c\}$ (line 8), which means that only sub-trees of b, a and c will be explored by the depth-first search. But in each sub-tree, the items f, g, d, b, a, e, c may be considered in descendant nodes (since they are in $Secondary(\alpha)$).

The procedure $Search$ is then called to perform the depth-first search (Algorithm 2). It receives the sets $Primary(\alpha)$ and $Secondary(\alpha)$ as parameters. This procedure loops on the primary items b, a and c, in that order (line 1). Here, we will assume that item a is first processed, as the process for item a is more interesting than item b. Thus, consider that $\beta = \{a\}$ is considered (line 2). Then, the database is scanned to create the projected database $\{a\}$-D (line 3). If only the database projection was performed, the resulting projected database would be as shown in (Table 5). However, as previously explained, transaction merging is performed at the same time as database projection, thanks to the efficient implementation of merging proposed in EFIM. Thus, transactions T_2 and T_3 are merged into a new transaction T_{23} during the database projection operation, and the resulting merged projected database is listed in Table 6. Moreover, during the database projection operation $u(\{a\})$ is calculated, and it is found that $u(\{a\}) = 20$. Since $\{a\}$ is not a HUI ($u(\{a\}) = 20 < minutil$), the itemset {a} is not output (line 4). Then, the projected database $\{a\}$-D is scanned to calculate the upper bounds lu and ru for items in $E(\{a\})$ using two utility-bin arrays. The result is $lu(\beta, c) = 65$, $lu(\beta, e) = 57$, $su(\beta, c) = 55$, $su(\beta, e) = 34$. The primary and secondary items are calculated as: $Primary(\beta) = \{e\}$ and $Secondary(\beta) = \{c, e\}$ (line 6 and 7). This

Table 6 Projected database $\{a\}$-D (when transaction merging is not applied)

TID	Transaction
T_{23}	$(e, 3)(c, 7)$
T_1	$(c, 1)$

means that only the extensions of $\beta \cup \{e\}$ will be explored and that in these sub-trees only items {c,e} will be considered for the depth-first search. The procedure is then recursively called to explore these sub-trees (line 9). The procedure then continues in the same way until all the high-utility itemsets have been found. The final set of high-utility itemsets output by the algorithm is listed in Table 3.

4.8 Complexity

The complexity of EFIM can be analyzed as follows. In terms of time, a $O(nw\ log(nw))$ sort is performed initially. This cost is however negligible since it is performed only once. Then, to process each primary itemset α encountered during the depth-first search, EFIM performs database projection, transaction merging and upper bound calculation. These three operations are each carried out in linear time ($O(nw)$), as discussed previously. Let l be the number of itemsets in the search space. The global time complexity of EFIM is thus $O(nw\ log(nw) + l(nw + nw + nw))$. Since the sort is only performed once, the term $nw\ log(nw)$ can be ignored and the complexity is actually closer to $O(lnw)$.

Thus, the performance of the algorithm is proportional to the number of itemsets in the search space. The number of itemsets in the search space is determined by the upper bounds which are used to prune the search space. In EFIM, two tight upper bounds are proposed. It was shown in the previous section that the proposed (revised) sub-tree utility upper bound is tighter than the remaining utility upper bound used in state-of-the-art one-phase HUI mining algorithms. But the efficiency of the proposed upper bounds will also be evaluated in the experimental evaluation section of this paper. Another aspect that increase the efficiency of EFIM is the use of a pattern-growth approach that only consider itemsets appearing in the database (unlike algorithms such as HUI-Miner, FHM and HUP-Miner, which may consider patterns not appearing in the database). Thus, this contributes to reduce the constant l.

Based on the above time complexity analysis, it can also be observed that the number of transactions n depends on how the actual size of projected databases, and in particular how many transactions can be merged. Thus, the more transactions are merged, the smaller n will be and the faster the algorithm will be. The influence of the effectiveness of transaction merging is also discussed in Experimental evaluation of this paper.

In terms of space complexity, the main cost is the space used by utility-bin arrays and the space for storing projected databases. Utility-bin arrays are created once and require $O(|I|)$ space. The database projection operation is performed for each primary itemset α and requires at most $O(nw)$ space for each projected database. In practice, this is small considering that projected databases become smaller as larger itemsets are explored and that database projections are implemented using offset pointers. Globally, the space complexity of EFIM is $O(|I| + lnw)$ because the number of projected databases is determined by the number of itemsets in the search space l.

5 Experimental results

We performed several 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 five state-of-the-art algorithms, namely UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM. Moreover, to also evaluate the influence of the design decisions in EFIM, we also compared it with two versions of EFIM named EFIM(nop) and EFIM(lu) where transaction

Table 7 Dataset characteristics

Dataset	# Transactions	# Distinct items	Avg. trans. length
Accident	340,183	468	33.8
BMS	59,601	497	4.8
Chess	3196	75	37.0
Connect	67,557	129	43.0
Foodmart	4141	1559	4.4
Mushroom	8124	119	23.0
Chainstore	1,112,949	46,086	7.2
Pumsb	49,046	2113	74
Kosarak	990,000	41,270	8.1

Table 8 Dataset types

Dataset	Type
Accident	Moderately dense, moderately long transactions
BMS	Sparse, short transactions
Chess	Dense, long transactions
Connect	Dense, long transactions
Foodmart	Sparse, short transactions
Mushroom	Dense, moderately long transactions
Chainstore	Very sparse, short transactions
Pumsb	Dense, very long transactions
Kosarak	Very sparse, moderately short transactions

merging (HTM) and search space pruning using the sub-tree utility were, respectively, deactivated. All the algorithms first read the database in main memory, then search for high-utility itemsets and write the result to disk. Since the input and output is the same for all algorithms, the cost of disk accesses has no influence on the results of the experiments.

Algorithms were implemented in Java and memory measurements were done using the standard 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*, *Mushroom* and *Chainstore*). These datasets are chosen because they have varied characteristics. Tables 7 and 8 summarize their characteristics. *Accident* is a large dataset with moderately long transactions. *BMS* is a sparse dataset with short transactions. *Chess* and *Connect* are dense datasets with long transactions and few items.*Foodmart* is a sparse dataset with short transactions. *Chainstore* is a sparse dataset with short transactions. *Mushroom* is a dense dataset with moderately long transactions. *Foodmart* and *Chainstore* are customer transaction database containing real external/internal utility values. For other datasets, external/internal utility values have been, respectively, generated in the [1, 1000] and [1, 5] intervals using a log-normal distribution, as done in previous state-of-the-art HUIM studies [2,6,16,26]. The datasets and the source code of the compared algorithms can be downloaded as part of the SPMF open-source data mining library [4] at http://www.philippe-fournier-viger.com/spmf/.

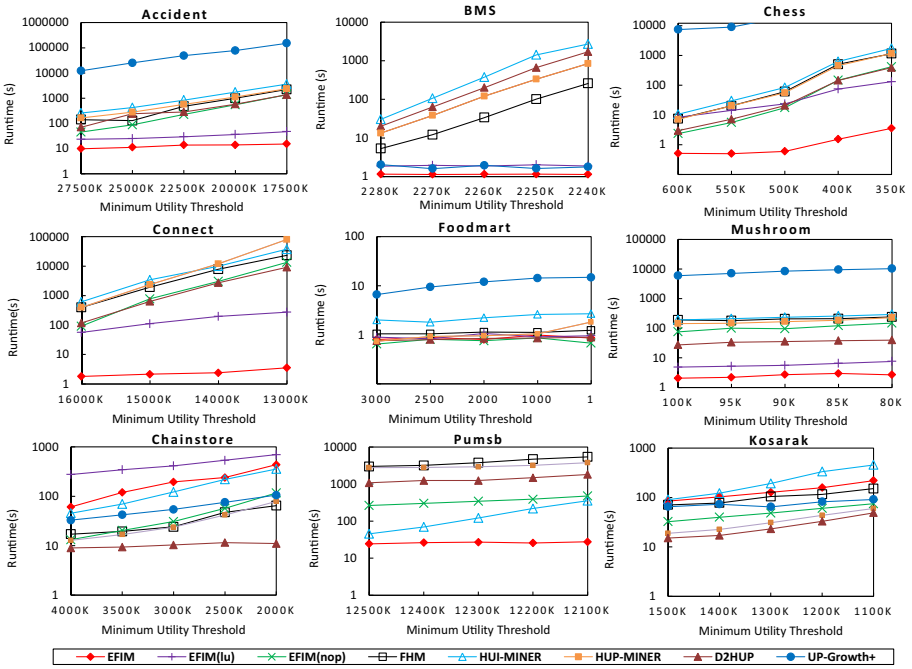

Fig. 4 Execution times on different datasets

5.1 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. As it Execution times are shown in Fig. 4. Note that for UP-Growth+, no result is shown for the *Connect* and *Pumsb* datasets and that some results are missing for the *Chess* dataset because UP-Growth+ exceeded the 16 GB memory limit. Results are also not shown for the EFIM(lu) algorithm on *Pumsb* and *Kosarak* because it took too long to terminate.

A first observation is that EFIM outperforms UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM on seven out of the nine datasets (all datasets except *Chainstore* and *Kosarak*). EFIM performs especially well for dense or moderately dense datasets. EFIM is in general about two to three orders of magnitude faster than the other algorithms. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart*, *Mushroom* and *Pumsb*, EFIM is, respectively, up to 15,334, 2, 33,028, –, 17, 3855 and – times faster than UP-Growth+, 154, 741, 323, 22,636, 2, 85 and 136 times faster than HUP-Miner, 89, 1490, 109, 2587, 1, 15 times, and 65 faster than d^{2}HUP, 236, 2370, 482, 10,586, 3, 110, 17 times faster than HUI-Miner and 145, 227, 321, 6606, 1, 90 and 197 times faster than FHM.

On the *Chainstore* dataset, EFIM(nop) is about 10 times slower than d^{2}HUP, but it is faster than UPGrowth+ and HUI-Miner and has comparable speed to FHM. On the *Kosarak* dataset, EFIM(nop) is faster than all algorithms except HUP-Miner and d^{2}HUP, and the difference with these two algorithms is small (1 time or less).

The reasons why EFIM performs so well in terms of execution time are fourfold and will be analyzed in more detail in the following experiments.

– First, the proposed sub-tree utility and local utility upper bounds allow EFIM to prune in general, a larger part of the search space compared to other algorithms, mainly because the sub-tree utility is a tighter upper bound than the remaining utility upper bound used by previous one-phase algorithms. Thus, less itemsets need to be considered to find the high-utility itemsets.
– Second, contrarily to algorithms such as HUI-Miner, FHM and HUP-Miner, EFIM only considers itemsets existing in the database (as it is a pattern-growth algorithm), thus avoiding spending time to consider itemsets that do not occur in the database.
– Third, the proposed HTM transaction merging technique allows replacing several transactions by single transactions, which greatly reduces the cost of database scans. Transaction merging works especially well for datasets where many transactions are similar, such as dense datasets. For these datasets, EFIM has the best performance in terms of execution time.
– Fourth, the efficient calculation of the proposed upper bounds in linear time using utility-bin arrays also contribute to the time efficiency of EFIM.

5.2 Influence of the *minutil* threshold on memory usage

In the previous experiment, the memory usage of EFIM was also recorded and compared with the same algorithms using the same parameters. Results are listed in Tables 9 and 10.

In terms of memory usage, EFIM clearly outperforms the other algorithms on all datasets. For *Accident*, *BMS*, *Chess*, *Connect*, *Foodmart*, *Mushroom*, *Chainstore*, *Pumsb* and *Kosarak*, EFIM uses 1.8, 4.4 14.9, 4.5, 1.3, 6.5, 1.9, 2.1 and 2.2 times less memory than the second fastest algorithm (d^{2}HUP). Moreover, EFIM uses 1.6, 9.2, 4.6, 8.1, 3.2, 3.1, 2.8, 1.47 and 2.45 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 nine datasets, and never more than 1 GB, while other algorithms often exceed 1 GB. UP-Growth+ generally performs the worse, as it even exceeded the 16 GB memory limit on the *Chess* dataset.

It is also interesting that EFIM is the most memory efficient, even for the *Chainstore* and *Kosarak* datasets, where EFIM is slower than d^{2}HUP. For these datasets, the trade-off that EFIM provides is useful for application where memory is limited.

A reason why EFIM 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). Other algorithms rely on more complex structures such as tree structures (e.g., UP-Growth+) and list structures (e.g., HUP-Miner, HUI-Miner and FHM), hyperlink structures (e.g., d^{2}HUP), which requires additional memory for pointers between nodes and for maintaining additional information. For example, it can be easily seen that for any itemset α the size of the α projected database of EFIM is smaller than the size of the α utility-list of HUP-Miner, HUI-Miner and FHM. Both structures contain entries representing transactions where α occurs. However, EFIM stores two fields per entry (transaction id and pointer), while the utility-list stores three (transaction id, *rutil* and *iutil* values). Moreover, a projected database can contain much less entries than the corresponding utility-list because of transaction merging. Thus, by applying transaction merging, several entries are replaced by a single one. For this reason, the gap in terms of memory between EFIM and the other algorithms tends to increase for dense datasets or datasets where transaction merging is more effective (the effect of transaction merging will be studied in the next subsection). Another reason for the high memory efficiency of EFIM compared to utility-list-based algorithms is that the number of projected databases created by EFIM is less than the number of utility-lists, because EFIM visits less nodes of the search-enumeration tree using the proposed sub-tree utility upper

Table 9 Comparison of maximum memory usage (MB) of HUI-Miner, FHM, EFIM and UP-Growth+

Dataset	HUI-MINER	FHM	EFIM	UP-Growth+
Accident	1656	1480	**895**	765
BMS	210	590	**64**	64
Chess	405	305	**65**	–
Connect	2565	3141	**385**	–
Foodmart	808	211	**64**	819
Mushroom	194	224	**71**	1507
Chainstore	1164	1270	**460**	1058
Pumsb	1221	1436	**986**	–
Kosarak	1163	1409	**576**	1207

Table 10 Comparison of maximum memory usage (MB) of HUP-Miner and d^{2}HUP

Dataset	HUP-Miner	d^{2}HUP
Accident	1787	1691
BMS	758	282
Chess	406	970
Connect	1204	1734
Foodmart	68	84
Mushroom	196	468
Chainstore	1034	878
Pumsb	1021	2046
Kosarak	712	1260

bound and by considering only itemsets appearing in the database (as it will be shown later). EFIM is also more efficient than two-phase algorithms such as UP-Growth+ since it is a one-phase algorithm (it does not need to maintain a large number of candidates in memory).

Lastly, another important characteristic of EFIM in terms of memory efficiency is that it reuses some of its data structures. As explained in Sect. 4.5, EFIM uses a very efficient mechanism called Fast Array Counting for calculating upper bounds. FAC only requires to create three utility-bin arrays that are then reused during the depth-first search to calculate the upper bounds of each itemset.

5.3 Influence of transaction merging on execution time

In terms of optimizations, the proposed transaction merging technique used in EFIM sometimes greatly increases its performance in terms of execution time. To assess how effective transaction merging is on the various datasets, Table 11 shows the average number of transactions in projected databases when EFIM is run with transaction merging activated (EFIM) and when it is run with transaction merging deactivated (EFIM(nop)). It can be observed that on the *Accident*, *Chess*, *Connect*, *Mushroom* and *Pumsb* datasets, transaction merging allows to reduce the size of projected database by more than 90 % in terms of number of transactions.

This is one of the reason why EFIM perform very well on dense or moderately dense datasets (*Chess*, *Connect*, *Mushroom* and *Pumsb*). For example, for *Connect* and $minutil =$ 13M, EFIM terminates in 3 seconds while HUP-Miner, d^{2}HUP, HUI-Miner and FHM, respec-

Table 11 Average projected database size (number of transactions)

Dataset	EFIM	EFIM(nop)	Size reduction (%)
Accident	784	113,304	99.3
BMS	112.6	204.1	44.8
Chess	2.6	1363.9	99.8
Connect	1.4	43,687	99.9
Foodmart	1.12	1.21	7.1
Mushroom	1.3	573	99.7
Chainstore	1085	1326	18.1
Pumsb	1075	22,326	95.2
Kosarak	1727	3653	53

Table 12 Comparison of visited node count for HUI-Miner, FHM and EFIM

Dataset	HUI-MINER	FHM	EFIM
Accident	131,300	128,135	**51,883**
BMS	2,205,782,168	212,800,883	**323**
Chess	6,311,753	6,271,900	**2,875,166**
Connect	3,444,785	3,420,253	**1,366,893**
Foodmart	55,172,950	1,880,740	**233,231**
Mushroom	3,329,191	3,089,819	**2,453,683**
Chainstore	4,422,322	8,285	**3005**
Pumsb	74,050	68,050	**56,267**
Kosarak	4,794,819	135,874	**2073**

tively, run for 22, 2, 10 and 6 hours. On dense datasets, transaction merging is very effective as projected transactions are more likely to be identical. This can be seen by comparing the runtime of EFIM and EFIM(nop). On *Chess*, *Connect* and *Mushroom*, EFIM is up to 116, 3790 and 55 times faster than EFIM(nop). For the *Accidents*, *BMS* and *Foodmart* datasets, transaction merging also reduces execution times but by a lesser amount (EFIM is up to 90, 2 and 2 times than EFIM(nop) on *Accident*, *BMS* and *Foodmart*). It is also interesting to note that the proposed transaction merging mechanism 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.

Lastly, another interesting observation related to transaction merging is that on the *Chainstore* and *Kosarak* datasets, which are quite sparse datasets, transaction merging is not very effective. For these datasets, EFIM(nop) is faster than EFIM. Thus, this shows that for some datasets the cost of transaction merging exceeds its benefits, and it should be deactivated.

5.4 Comparison of the number of visited nodes

We also performed an experiment to compare the ability at pruning the search space of EFIM with other algorithms. Tables 12 and 13 show the number of nodes of the search-enumeration tree (itemsets) visited by EFIM, UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM

Table 13 Comparison of visited node count for UP-Growth+, HUP-Miner and d^{2}HUP

Dataset	UP-Growth+	HUP-Miner	d^2 HUP
Accident	3,234,611	113,608	119,427
BMS	91,195	205,556,936	220,323,377
Chess	–	3,385,134	3,051,789
Connect			
Foodmart	233,231	1,258,820	233,231
Mushroom	13,779,114	3,054,253	2,919,842
Chainstore	987	3007	
Pumsb	–	1,029,702	62,361
Kosarak	2292	57,706	9257

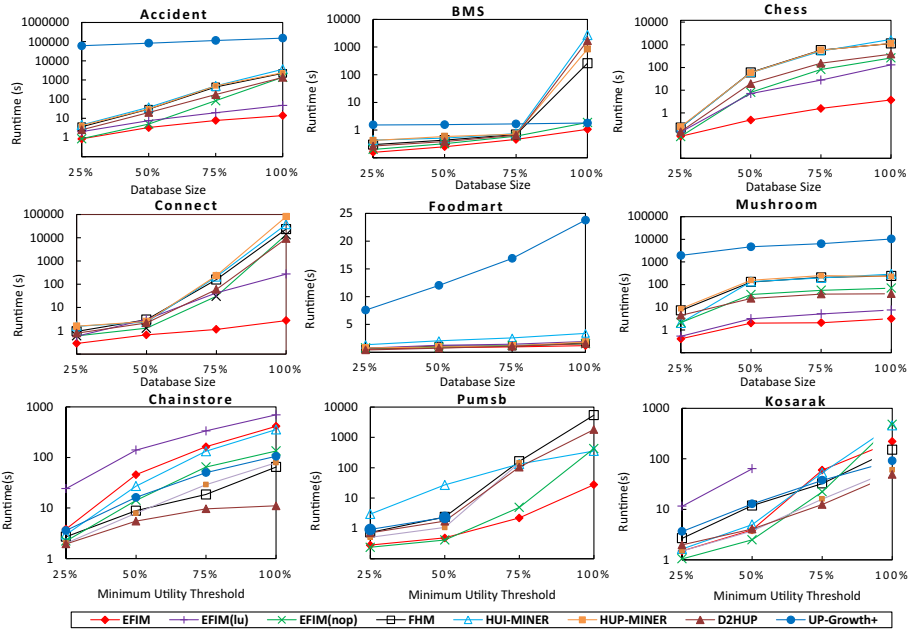

Fig. 5 Scalability on different datasets

for the lowest *minutil* values on the same datasets. It can be observed that EFIM is generally more effective at pruning the search space than the other algorithms, thanks to its proposed sub-tree utility and local utility upper bounds. For example, it can be observed that EFIM visits, respectively, up to 282, 636,000, 682,000, 6,000,800 and 658,000 times less nodes than UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM.

5.5 Influence of the number of transactions on execution time

Lastly, we also compared the execution time of EFIM with UP-Growth+, HUP-Miner, d^{2}HUP, HUI-Miner and FHM while varying the number of transactions in each dataset to assess the scalability of the algorithms. For this experiment, algorithms were run on the same datasets

using the lowest *minutil* values used in previous experiments, while varying the number of transactions from 25 to 100 %. Results are shown in Fig. 5. It can be observed that EFIM's runtime linearly increases w.r.t to the number of transactions for the six first datasets, while runtimes of UP-Growth+, HUP-Miner, d^2HUP, HUI-Miner and FHM almost exponentially increase for some datasets such as *Chess* and *BMS*. Thus, it can be concluded that EFIM hasthe overall best scalability on these datasets. The reason for this excellent scalability is that most operations performed by EFIM for an itemset α are performed in linear time. Thus, the complexity of EFIM is mostly linear for each node visited in the search space. Moreover, thanks to the proposed sub-tree utility and local utility upper bounds, EFIM can prune a large part of the search space.

For the three last datasets (*Chainstore*, *Pumsb* and *Kosarak*), the scalability of basically all algorithms exponentially increase, and EFIM has comparable scalability to other algorithms on those datasets. The reason why the execution times of algorithms increase exponentially on these three datasets is that the *minutil* threshold is fixed, and thus, for a larger amount of transactions, more itemsets cannot be pruned by the *minutil* threshold. Because the search space becomes exponentially larger, even if EFIM perform linear time operations for each itemset in its search space, its execution time exponentially increases (as other algorithms for this dataset).

6 Conclusion

High-utility itemset mining is an important data mining task with numerous applications. However, it remains very time-consuming [6,16,26]. To improve the efficiency of HUIM in terms of memory and execution time, 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* to prune the search space. 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 two techniques for database projection and transaction merging named *High-utility Database Projection* (HDP) and *High-utility Transaction Mergin* (HTM), 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. For very sparse datasets, EFIM is sometimes slower than some other algorithms. However, in terms of memory usage, it always outperforms the other algorithms. Moreover, results show that EFIM has excellent scalability on both sparse and dense datasets. The source code of all algorithms and datasets used in the experiments can be downloaded as part of the SPMF open-source data mining library [4] at http://www.philippe-fournier-viger.com/spmf/.

The work presented in this paper introduces several novel ideas that can be reused in the field of high-utility pattern mining to develop extensions of the problem of high-utility itemset mining. In particular, some research possibilities are to extend EFIM for popular variations of the HUIM problem such as mining concise representations of high-utility itemsets [5], high-utility association rules [23], top-k high-utility itemsets [19,21,27], incremental high-utility itemsets [2,13], periodic high-utility itemsets [7] and on-shelf high-utility itemsets [8]. Moreover, additional optimizations may also be developed to further improve the performance of EFIM. For example, in this paper, the EFIM algorithm was applied to database that fits in

main memory. An interesting possibility is to redesign EFIM as a distributed algorithm to be able to mine very large databases, similar to the algorithm proposed in [14].

Acknowledgments This project was supported by a NSERC Discovery grant from the Government of Canada and an initiating fund provided to the second author by Harbin Institute of Technology (Shenzhen).

References

1. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th international conference on very large databases, Morgan Kaufmann, Santiago de Chile, Chile, September 1994, pp 487–499
2. Ahmed CF, Tanbeer SK, Jeong BS, Lee YK (2009) Efficient tree structures for high-utility pattern mining in incremental databases. IEEE Trans Knowl Data Eng 21(12):1708–1721
3. Ahmed CF, Tanbeer SK, Jeong B (2010) Mining high utility web access sequences in dynamic web log data. In: Proceedings of the international conference on software engineering artificial intelligence networking and parallel/distributed computing, IEEE, London, UK, June 2010, pp 76–81
4. Fournier-Viger P, Gomariz A, Gueniche T, Soltani A, Wu CW, Tseng VS (2014) SPMF: a Java open-source pattern mining library. J Mach Learn Res 15:3389–3393
5. Fournier-Viger P, Wu CW, Tseng VS (2014) Novel concise representations of high utility itemsets using generator patterns. In: Proceedings of the 10th international conference on advanced data mining and applications, Guilin, China, December 2014. Lecture Notes in Artificial Intelligence, vol 8933. Springer, Berlin, pp 30–43
6. Fournier-Viger P, Wu CW, Zida S, Tseng VS (2014) FHM: faster high-utility itemset mining using estimated utility co-occurrence pruning. in: Proceedings of the 21st international symposium on methodologies for intelligent systems, Roskilde, Denmark, June 2014. Lecture Notes in Artificial Intelligence, vol 9384. Springer, Berlin, pp 83–92
7. Fournier-Viger P, Lin JCW, Duong QH, Dam TL (2016) PHM: mining periodic high-utility itemsets. In: Proceedings of the 16th industrial conference on data mining, New York, USA, July 2016. Lecture Notes in Artificial Intelligence, vol 9728, Springer, Berlin, pp 64–79
8. Fournier-Viger P, Zida S (2015) FOSHU: faster on-shelf high utility itemset mining with or without negative unit profit. In: Proceedings of the 30th symposium on applied computing, ACM, Salamanca, Spain, April 2015, pp 857–864
9. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min Knowl Discov 8(1):53–87
10. Pei J, Han J, Lu H, Nishio S, Tang S, Yang D (2001) H-Mine: hyper-structure mining of frequent patterns in large databases. In: Proceedings of the 2001 IEEE international conference on data mining, IEEE, San Jose, CA, November 2001, pp 441–448
11. Krishnamoorthy S (2015) Pruning strategies for mining high utility itemsets. Expert Syst Appl 42(5):2371–2381
12. Lan GC, Hong TP, Tseng VS (2014) An efficient projection-based indexing approach for mining high utility itemsets. Knowl Inf Syst 38(1):85–107
13. Lin JCW, Hong TP, Lan GC, Wong JW, Lin WY (2015) Efficient updating of discovered high-utility itemsets for transaction deletion in dynamic databases. Adv Eng Inform 29(1):16–27
14. Lin YC, Wu CW, Tseng VS (2015) Mining high utility itemsets in big data. In: Proceedings of the 9th Pacific-Asia conference on knowledge discovery and data mining, Ho Chi Minh City, Vietnam, May 2015, Lecture Notes in Artificial Intelligence, vol 9077. Springer, Berlin, pp 649–661
15. Liu J, Wang K, Fung B (2012) Direct discovery of high utility itemsets without candidate generation. In: Proceedings of the 12th IEEE international conference on data mining, IEEE, Brussels, Belgium, December 2012, pp 984–989
16. Liu M, Qu J (2012) Mining high utility itemsets without candidate generation. In: Proceedings of the 22nd ACM international conference on information and knowledge management, ACM, Maui, HI, October 2012, pp 55–64
17. Liu Y, Cheng C, Tseng VS (2013) Mining differential top-k co-expression patterns from time course comparative gene expression datasets. BMC Bioinform 14(230)
18. Liu Y, Liao W, Choudhary A (2005) A two-phase algorithm for fast discovery of high utility itemsets. In: Proceedings of the 9th Pacific-Asia conference on knowledge discovery and data mining, Hanoi, Vietnam, May 2005, Lecture Notes in Artificial Intelligence, vol 3518. Springer, Berlin, pp 689–695

19. Lu T, Liu Y, Wang L (2014) An algorithm of top-k high utility itemsets mining over data stream. J Softw 9(9):2342–2347
20. Pei J, Han J, Mortazavi-Asl B, Wang J, Pinto H, Chen Q, Dayal U, Hsu M (2004) Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans Knowl Data Eng 16(11):1424–1440
21. Ryang H, Yun U (2015) Top-k high utility pattern mining with effective threshold raising strategies. Knowl Based Syst 76:109–126
22. Rymon R (1992) Search through systematic set enumeration. In: Proceedings of the third international conference on principles of knowledge representation and reasoning, Morgan Kaufmann, Cambridge, MA, October 1992, pp 539–50
23. Sahoo J, Das AK, Goswami A (2015) An efficient approach for mining association rules from high utility itemsets. Expert Syst Appl 42(13):5754–5778
24. Song W, Liu Y, Li J (2014) BAHUI: fast and memory efficient mining of high utility itemsets based on bitmap. Proc Int J Data Wareh Min 10(1):1–15
25. Thilagu M, Nadarajan R (2012) Efficiently mining of effective web traversal patterns with average utility. In: Proceedings of the international conference on communication, computing, and security. CRC Press, Gurgaon, India, September 2016, pp 444–451
26. Tseng VS, Shie BE, Wu CW, Yu PS (2013) Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans Knowl Data Eng 25(8):1772–1786
27. Tseng VS, Wu CW, Fournier-Viger P, Yu P (2016) Efficient algorithms for mining top-k high utility itemsets. IEEE Trans Knowl Data Eng 28(1):54–67
28. Uno T, Kiyomi M, Arimura H (2004) LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets. In: Proceedings of the ICDM'04 Workshop on Frequent Itemset Mining Implementations, CEUR, Brighton, UK, November 2014
29. Yao H, Hamilton HJ, Butz CJ (2004) A foundational approach to mining itemset utilities from databases. In: Proceedings of the 3rd SIAM international conference on data mining, SIAM, Lake Buena Vista, FL, USA, April 2004, pp 482–486
30. Yin J, Zheng Z, Cao L, Song Y, Wei, W (2012) An efficient algorithm for mining high utility sequential patterns.In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, ACM, Beijing, China, August 2012, pp 660–668
31. Yin J, Zheng Z, Cao L, Song Y, Wei W (2013) Efficiently mining top-k high utility sequential patterns. In: Proceedings of the 13th international conference on data mining, IEEE, Dallas, TX, USA, December 2013, pp 1259–1264
32. Yun U, Ryang H, Ryu KH (2014) High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates. Expert Syst Appl 41(8):3861–3878
33. Zaki MJ (2000) Scalable algorithms for association mining. IEEE Trans Knowl Data Eng 12(3):372–390
34. Zida S, Fournier-Viger P, Wu CW, Lin JCW, Tseng VS (2015) Efficient mining of high utility sequential rules. In: Proceedings of the 11th international conference on machine learning and data mining, Hamburg, Germany, July 2015, Lecture Notes in Artificial Intelligence vol 9166. Springer, Berlin, pp 1–15
35. Zida S, Fournier-Viger P, Lin JCW, Wu CW, Tseng VS (2015) EFIM: a highly efficient algorithm for high-utility itemset mining. In: Proceedings of the 14th Mexican international conference on artificial intelligence, Cuernavaca, Mexico, October 2015. Lecture Notes in Artificial Intelligence, vol 9413. Springer, Berlin, pp 530–546

Souleymane Zida has obtained a M.Sc. student in Computer Science at University of Moncton, Canada. His research interests are high-utility mining, frequent pattern mining, sequential rule mining and algorithmic.

Philippe Fournier-Viger received the Ph.D. degree in Computer Science from the University of Quebec in Montreal (2010). He is currently a full professor and Youth 1000 scholar at the Harbin Institute of Technology Shenzhen Grad. School, China. He has published more than 130 research papers in refereed international conferences and journals. His research interests include data mining, pattern mining, sequence analysis and prediction, text mining, e-learning and social network mining. He is the founder of the popular SPMF open-source data mining library, which has been cited in more than 330 research papers since 2010.

Jerry Chun-Wei Lin received the Ph.D. degree in 2010 from the National Cheng Kung University, Tainan, Taiwan. He is currently working as an assistant professor at School of Computer Science and Technology, Harbin Institute of Technology Shenzhen Graduate School, China. He has published around 180 research papers in referred journals and international conferences. His interests include data mining, soft computing, privacy preserving data mining and security, social network and cloud computing.

Cheng-Wei Wu received the Ph.D. degree from the Department of Computer Science and Information Engineering at National Cheng Kung University, Taiwan, in 2015. Currently, he is hired as a postdoctoral researcher in College of Computer Science, National Chiao Tung University, Taiwan. His research interests include big data mining, machine learning, Internet of Things and sensor data analysis.

Vincent S. Tseng is a Professor at Department of Computer Science in National Chiao Tung University. Currently he also serves as the chair for IEEE Computational Intelligence Society Tainan Chapter. Dr. Tseng has a wide variety of research interests covering data mining, big data, biomedical informatics, multimedia databases, mobile and Web technologies. He has published more than 300 research papers in referred journals and international conferences as well as 15 patents held. He has been on the editorial board of a number of journals including IEEE Transactions on Knowledge and Data Engineering, IEEE Journal on Biomedical and Health Informatics, ACM Transactions on Knowledge Discovery from Data, etc. He has also served as chairs/program committee members for premier international conferences related including KDD, ICDM, SDM, PAKDD, ICDE, CIKM, IJCAI. He is also the recipient of 2014 K. T. Li Breakthrough Award.