Mining Minimal High-Utility Itemsets

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

[1] School of Natural Sciences and Humanities,
[2] School of Computer Science and Technology, Harbin Institute of Technology
Shenzhen Graduate School, Shenzhen, China
[3] Dept. of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
[4] Dept. of Computer Science and Mathematics, University of Indianapolis,
Indianapolis, United states
philfv@hitsz.edu.cn, jerrylin@ieee.org, silvemoonfox@gmail.com,
vtseng@cs.nctu.edu.tw, faghihiu@indy.edu

Abstract. Mining high-utility itemsets (HUIs) is a key data mining task. It consists of discovering groups of items that yield a high profit in transaction databases. A major drawback of traditional high-utility itemset mining algorithms is that they can return a large number of HUIs. Analyzing a large result set can be very time-consuming for users. To address this issue, concise representations of high-utility itemsets have been proposed such as closed HUIs, maximal HUIs and generators of HUIs. In this paper, we explore a novel representation called the *minimal high utility itemsets* (MinHUIs), defined as the smallest sets of items that generate a high profit, study its properties, and design an efficient algorithm named MinFHM to discover it. An extensive experimental study with real-life datasets shows that mining MinHUIs can be much faster than mining other concise representations or all HUIs, and that it can greatly reduce the size of the result set presented to the user.

Keywords: utility mining, high-utility itemsets, minimal itemsets

1 Introduction

High-utility itemset mining (HUIM) is an emerging data mining task, which consists of discovering sets of items that have a high utility (yield a high profit) in customer transaction databases [2, 5, 8–13, 15]. HUIM can be viewed as a generalization of *Frequent Itemset Mining* (FIM) [1, 3, 4, 17], where weights are assigned to each item to represent their importance (e.g. unit profit), and purchase quantities of items in transactions are not restricted to binary values. HUIM has applications in many domains such as customer purchase behavior analysis, website click stream analysis, and biomedicine [2, 12, 15]. HUIM is widely considered as more difficult than FIM because the utility measure used in HUIM is neither anti-monotonic nor monotonic, unlike the support measure used in FIM [1]. In other words, the utility of an itemset can be lower, equal or higher than the utility of any of its supersets. Hence, techniques for pruning the search space in FIM

cannot be directly applied in HUIM. Although HUIM has attracted lots of attention in recent years, an important limitation of traditional high-utility itemset mining algorithms [2, 5, 8–13, 15] is that they can generate a very large amount of HUIs. This can make HUI mining algorithms run out of storage space or even fail to terminate. Moreover, it is very time-consuming for a user to analyze a very large set of HUIs [6, 16]. To address this issue, it was proposed to mine concise representations of all HUIs rather than the whole set of HUIs. Three main representations have been proposed in previous work. *Maximal HUIs* are the HUIs that are not included in other HUIs. For a retailer, it answers the question of finding the largest sets of items that yield a high profit. *Closed HUIs* [16] are the HUIs that are not included in another HUIs having the same support. For a retailer, it answers the question of finding the largest groups of items yielding a high profit, which are common to groups of customers. Finally, *Generators of HUIs* [6] answer the question of finding the smallest sets of items common to groups of customers having bought a same high-utility itemset.

In this paper, we investigate a novel representation of HUIs named the *minimal high-utility itemsets* (MinHUIs), defined as the smallest HUIs (HUIs that are not included in another HUI). This representation addresses the problem that HUIM algorithms often find very long HUIs containing many items. But these HUIs often represent rare cases, as in real-life, few customers exactly buy the same large set of items. For marketing purpose, a retailer may be more interested in finding the smallest sets of items that generate a high profit, since it is easier to co-promote a small set of items targeted at many customers rather than a large set of items targeted at few customers. The proposed representation is the opposite of maximal HUIs. It aims at discovering the smallest sets of items that generate a high profit in a database rather than the largest ones. Because this representation has been unexplored, it remains an important challenge to explore the properties of this representation and define an efficient algorithm for mining this representation. In this paper, we address this challenge. We propose a novel algorithm named *MinFHM* to discover this representation efficiently. MinFHM extends FHM, a state-of-the-art algorithm for HUI mining by using a novel pruning property, and several optimizations to mine MinHUIs efficiently. We compare the performance of MinFHM with FHM on several real-life datasets. Results show that mining minimal HUIs is almost two orders of magnitude faster than mining all HUIs, or other concise representations of HUIs and that it can greatly reduce the result set presented to the user. The rest of this paper is organized as follows. Section 2, 3, 4, 5 and 6 respectively present related work, minimal high-utility itemsets, the MinFHM algorithm, the experimental evaluation and the conclusion.

2 Related Work

The problem of HUIM is defined as follows [5, 12, 13, 15]. Consider a set of items (symbols) denoted as I. A *transaction database* is a set of transactions $D = \{T_0, T_1, ..., T_n\}$ such that for each transaction T_c, $T_c \subseteq I$ and T_c has a unique

identifier c called its Tid. Each item $i \in I$ is associated with a positive number $p(i)$, called its external utility, representing its importance (e.g. unit profit). Moreover, for each transaction T_c such that $i \in T_c$, a positive number $q(i, T_c)$ is called the internal utility of i, which represents the purchase quantity of i in transaction T_c. For example, Table 1 shows a transaction database containing five transactions $(T_0, T_1 ... T_4)$, which will be used as running example. Transaction T_3 indicates that items a, c, and e appear in this transaction with an internal utility of respectively 2, 6, and 2. Table 2 indicates that the external utilities of these items are respectively 5, 1, and 3.

Table 1. A transaction database

TID	Transaction
T_0	$(a,1),(b,5),(c,1),(d,3),(e,1)$
T_1	$(b,4),(c,3),(d,3),(e,1)$
T_2	$(a,1),(c,1),(d,1)$
T_3	$(a,2),(c,6),(e,2)$
T_4	$(b,2),(c,2),(e,1)$

Table 2. External utility values

Item	a	b	c	d	e
Unit profit	5	2	1	2	3

The utility of an item i in a transaction T_c is denoted as $u(i, T_c)$ and defined as $p(i) \times q(i, T_c)$. The utility of an itemset X (a group of items $X \subseteq I$) in a transaction T_c is denoted as $u(X, T_c)$ and defined as $u(X, T_c) = \sum_{i \in X} u(i, T_c)$. The utility of an itemset X (in all transactions of a transaction database) is denoted as $u(X)$ and defined as $u(X) = \sum_{T_c \in g(X)} u(X, T_c)$, where $g(X)$ is the set of transactions containing X. The *problem of high-utility itemset mining* is to discover all high-utility itemsets. 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. For instance, the utility of the itemset $\{a, c\}$ is $u(\{a, c\}) = u(a) + u(c) = u(a, T_0) + u(a, T_2) + u(a, T_3) + u(c, T_0) + u(c, T_2) + u(c, T_3) = 5 + 5 + 10 + 1 + 1 + 6 = 28$. If $minutil = 25$, the set of HUIs is $\{a, c\} : 28$, $\{a, c, e\} : 31$, $\{a, b, c, d, e\} : 25$, $\{b, c\} : 28$, $\{b, c, d\} : 34$, $\{b, c, d, e\} : 40$, $\{b, c, e\} : 37$, $\{b, d\} : 30$, $\{b, d, e\} : 36$, $\{b, e\} : 31$ and $\{c, e\} : 27$, where each HUI is annotated with its utility. A major challenge in HUIM is that the utility measure is not monotonic or anti-monotonic, and thus that pruning techniques developed in FIM cannot be directly used in FIM to prune the search space. Many HUIM algorithms such as Two-Phase [13], IHUP [2], BAHUI [11], PB [8], and UPGrowth+ [15] overcome this challenge by using a measure called the *Transaction-Weighted Utilization (TWU)* measure, which provides an upper-bound on the utility of itemsets and is anti-monotonic [2, 13, 15]. The aforementioned algorithms first identify candidate high utility itemsets by calculating their TWUs. Then, in a second phase, they scan the database to calculate the exact utility of all candidates found in the first phase to eliminate low utility itemsets. The TWU measure is defined as follows. The *transaction utility* (TU) of a transaction T_c is the sum of the utilities of all the items in T_c. i.e. $TU(T_c) = \sum_{x \in T_c} u(x, T_c)$. The *transaction-weighted utilization* (TWU) of an itemset X is defined as the sum of the transaction

utilities of transactions containing X, i.e. $TWU(X) = \sum_{T_c \in g(X)} TU(T_c)$. For instance, the TUs of T_0, T_1, T_2, T_3 and T_4 are respectively 25, 20, 8, 22 and 9. The TWU of single items a, b, c, d, e are respectively 55, 54, 84, 53 and 76. $TWU(\{c,d\}) = TU(T_0) + TU(T_1) + TU(T_2) = 25 + 20 + 8 = 53$. The TWU has the following useful property for pruning the search space [13].

Property 1 (Pruning search space using the TWU). Let X be an itemset, if $TWU(X) < minutil$, then X and its supersets are low utility. [13]

Recently, algorithms were proposed to mine HUIs directly using a single phase [5, 9, 12], and were shown to outperform previous algorithms. FHM is to our knowledge the fastest algorithm for mining HUIs [5]. It performs a depth-first search to explore the search space of HUIs, and introduces an additional optimization named EUCP [5] to prune the search space using information about co-occurrences. FHM assign a structure named *utility-list* [5, 9, 12] to each itemset. Utility-lists allow calculating the utility of an itemset quickly by making join operations with utility-lists of shorter patterns. Utility-lists are defined as follows. Let $\succ$ be any total order on items from I. The *utility-list* of an itemset X in a database D is a set of tuples such that there is a tuple $(tid, iutil, rutil)$ for each transaction T_{tid} containing X. The *iutil* element of a tuple is the utility of X in T_{tid}. i.e., $u(X, T_{tid})$. The *rutil* element of a tuple is defined as $\sum_{i \in T_{tid} \wedge i \succ x \forall x \in X} u(i, T_{tid})$. For instance, assume that $\succ$ is the alphabetical order. The utility-list of $\{a\}$ is $\{(T_0, 5, 20), (T_2, 5, 3), (T_3, 10, 12)\}$. The utility-list of $\{d\}$ is $\{(T_0, 6, 3), (T_1, 6, 3), (T_2, 2, 0)\}$. The utility-list of $\{a, d\}$ is $\{(T_0, 11, 3), (T_2, 7, 0)\}$. To discover HUIs, FHM performs a single database scan to create utility-lists of patterns containing single items. Then, longer patterns are obtained by performing the join operation of utility-lists of shorter patterns (see [5, 12] for details). Calculating the utility of an itemset using its utility-list and pruning the search space is done as follows.

Property 2 (Calculating utility of an itemset using its utility-list). The utility of an itemset is the sum of *iutil* values in its utility-list.

Property 3 (Pruning search space using utility-lists). Let X be an itemset. Let the *extensions* of X be the itemsets that can be obtained by appending an item y to X such that $y \succ i$, $\forall i \in X$. If the sum of *iutil* and *rutil* values in $ul(X)$ is less than $minutil$, X and its extensions are low utility.

FHM is very efficient. However, it can generate a huge amount of HUIs. This can make the algorithm run out of storage space, and fail to terminate. Furthermore, it is very inconvenient for a user to analyze a large set of HUIs. To discover small and representative subsets of all HUIs, concise representations of HUIs have been proposed such as *closed HUIs* [16], *maximal HUIs* [14], and *generators of HUIs* [6], defined as follows. The *support* of an itemset X in a database D is denoted as $sup(X)$ and defined as $|g(X)|$, the number of transactions containing X. A HUI X is a *closed HUI* (*CHUI*) [16] iff there exists no HUI Y such that $X \subset Y$ and $sup(X) = sup(Y)$. A HUI X is a *maximal HUI* (*MaxHUI*) [14] iff there exists no HUI Y, such that $X \subset Y$. An itemset X is a *generator of high-utility itemsets*

($GHUI$) iff (1) there exists no itemset $Y \subset X$, such that $sup(X) = sup(Y)$, and (2) there exists an itemset Z such that $X \subseteq Z$ and $u(Z) \geq minutil$ [6].

3 The Minimal High Utility Itemsets

CHUIs, MaxHUIs and GHUIs are designed to provide answers to different questions that retailers may have about customer transactions, as outlined in the introduction. A drawback of the representations of CHUIs and MaxHUIs is that they tend to find very long HUIs, containing many items. A problem with these representations is thus that these HUIs often represent rare cases, as generally few customers exactly buy a same large set of items. For marketing purpose, a retailer may be more interested in finding the smallest sets of items generating a high profit, since it is easier to co-promote a small set of items targeted at a many customers rather than a large set of items targeted at few customers. The representation of GHUIs [6] partially addresses this issue by finding the smallest itemsets common to groups of customers having bought a set of items generating a high profit. However, no research has yet considered mining only the smallest HUIs. To address this research gap, we thereafter propose the novel concise representation of *minimal high-utility itemsets* (MinHUIs).

Definition 1 (Minimal HUIs). An itemset X is a *minimal high-utility itemset* ($MinHUI$) iff $u(X) \geq minutil$ and there does not exist an itemset $Y \subset X$ such that $u(Y) \geq minutil$.

This proposed representation is the opposite of maximal HUIs, i.e. it consists of the smallest sets of items that generate a high profit rather than the largest. To better show the relationship between the proposed MinHUIs, and the previously proposed CHUIs, MinHUIs and GHUIs, Fig. 1 presents an illustration of these various types of patterns, for the running example. In this figure, all equivalence classes containing at least a HUI are represented. An *equivalence class* is a set of itemsets supported by the same set of transactions, ordered by the subset relation. For example, $\{\{a, e\}, \{a, c, e\}\}$ is the equivalence class of itemsets appearing in transactions T_0 and T_2. An alternative and equivalent definition of GHUIs and CHUIs is the following. For each equivalence class containing a HUI, the CHUI is the largest itemset (the one having no superset in that equivalence class), while GHUI(s) are the smallest itemsets (those having no subset in that same equivalence class). Note that in the illustration equivalence classes are represented as Hasse diagrams and that low-utility itemsets that are not GHUIs in each equivalence class are not shown. As it can be seen in this example, MaxHUIs can be very long and thus offer few useful information to the user. For example, the only MaxHUI found in the running example is $\{a, b, c, d, e\}$, and it represents the very specific case of a single customer (T_0). CHUIs are interesting but they also tend to contain very large itemsets. For example, CHUIs include $\{a, b, c, d, e\}$ in the example. GHUIs find the smallest itemsets common to a set of customers. However, a drawback of GHUIs is that some of these itemsets are low-utility such as $\{e\}$ in the example. To address these issues, the proposed MinHUIs are defined

as the smallest high-utility itemsets. These itemsets are interesting as they tend to have a high support (represent many customers) as shown in this example, and are all HUIs. MinHUIs in this example are: $\{b, c\}$, $\{b, d\}$, $\{b, e\}$, $\{a, c\}$ and $\{c, e\}$. Formally, the relationship between these various sets of HUIs are the following: $MinHUIs \subseteq HUIs \subseteq 2^I$, $MaxHUIs \subseteq CHUIs \subseteq HUIs \subseteq 2^I$, and $GHUIs \subseteq 2^I$.

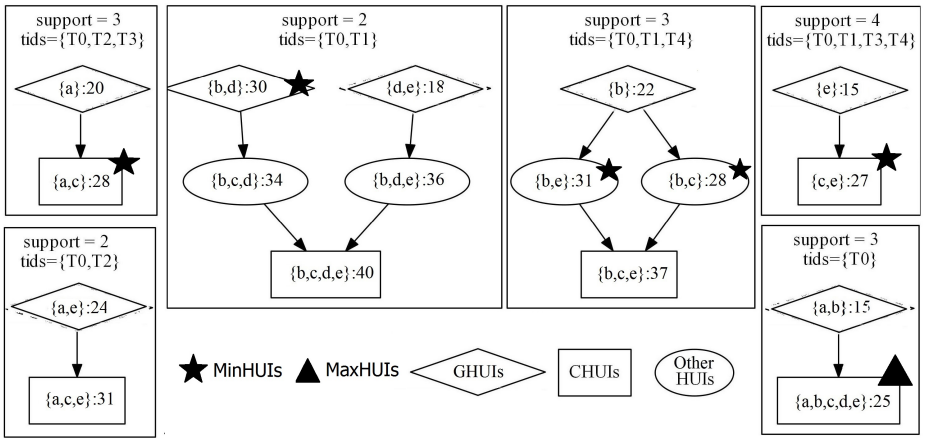

Fig. 1. HUIs and their equivalence classes (represented using Hasse diagrams)

A problem with previous representations is that the number of discovered patterns can still be very large since the number of HUIs, CHUIs, GHUIs and MaxHUIs increases when the *minutil* threshold is decreased. It is interesting to note that this is not necessarily the case for MinHUIs (Property 4).

Property 4 (Influence of minutil on MinHUI count). If *minutil* is lowered, the number of MinHUIs may increase, decrease or stay the same. Moreover, if $minutil = 1$, the set of MinHUIs is equal to I.

The above property is demonstrated using the running example. For $minutil = 20$, there is 3 MinHUIs: $\{a\}$, $\{b\}$, and $\{c, e\}$. For $minutil = 25$, there are 5 MinHUIs: $\{b, c\}$, $\{b, d\}$, $\{b, e\}$, $\{a, c\}$, and $\{c, e\}$. For $minutil = 30$, there are 3 MinHUIs: $\{b, d\}$, $\{b, e\}$, and $\{a, c, e\}$. Another interesting property of MinHUIs is used for pruning the search space in the proposed MinFHM algorithm

Property 5 (pruning property of minimal high-utility itemsets). If an itemset X is a MinHUI, then supersets of X are not MinHUIs.

4 The MinFHM algorithm

This section presents the proposed MinFHM algorithm. It first describes the main procedure, which is inspired by the FHM [5] algorithm. This procedure

is designed to mine all HUIs. Then, it explains how that procedure is adapted to find only MinHUIs. The resulting algorithm is called MinFHM. The main procedure of MinFHM (Algorithm 1) takes as input a transaction database with utility values and the *minutil* threshold. The algorithm first scans the database to calculate the TWU of each item. Then, the algorithm identifies the set I^* of all items having a TWU no less than *minutil* (other items are ignored since they cannot be part of a high-utility itemsets by Property 3). The TWU values of items are then used to establish a total order $\succ$ on items, which is the order of ascending TWU values (as suggested in [12]). A second database scan is then performed. During this database scan, items in transactions are reordered according to the total order $\succ$, the utility-list of each item $i \in I^*$ is built and a structure named EUCS (Estimated Utility Co-Occurrence Structure) is built [5]. This latter structure is defined as a set of triples of the form $(a, b, c) \in I^* \times I^* \times \mathbb{R}$. A triple (a,b,c) indicates that TWU($\{a, b\}$) = c. The EUCS can be implemented as a triangular matrix or as a hash map of hash maps where only tuples of the form (a, b, c) such that $c \neq 0$ are kept. In our implementation, we have used this latter representation as it is more memory efficient. Building the EUCS is very fast (it is performed with a single database scan) and occupies a small amount of memory, bounded by $|I^*| \times |I^*|$, although in practice the size is much smaller because a limited number of pairs of items co-occurs in transactions (cf. section 5). After the construction of the EUCS, the depth-first search exploration of itemsets starts by calling the recursive procedure *Search* with the empty itemset $\emptyset$, the set of single items I^*, *minutil* and the EUCS structure. The

Algorithm 1: The MinFHM algorithm

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

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

Search procedure (Algorithm 2) takes as input (1) an itemset P, (2) extensions of P having the form Pz meaning that Pz was previously obtained by appending an item z to P, (3) *minutil* and (4) the EUCS. The search procedure operates as follows. For each extension Px of P, if the sum of the *iutil* values of the utility-list of Px is no less than *minutil*, then Px is a high-utility itemset and it is output (cf. Property 4). Then, if the sum of *iutil* and *rutil* values in the utility-list of Px are no less than *minutil*, it means that extensions of Px should be explored. This is performed by merging Px with all extensions Py of P such that $y \succ x$ to form extensions of the form Pxy containing $|Px| + 1$ items. The

utility-list of Pxy is then constructed as in HUI-Miner by calling the *Construct* procedure (cf. Algorithm 3) to join the utility-lists of P, Px and Py. This latter procedure is the same as in HUI-Miner [12] and is thus not detailed here. Then, a recursive call to the *Search* procedure with Pxy is done to calculate its utility and explore its extension(s). Since the *Search* procedure starts from single items, it recursively explores the search space of itemsets by appending single items and it only prunes the search space based on Property 5. It can be easily seen based on Property 1, 2 and 3 that this procedure is correct and complete to discover all high-utility itemsets.

Algorithm 2: The *Search* procedure

input : P: an itemset, *ExtensionsOfP*: a set of extensions of P, , *minutil*: a user-specified threshold, $EUCS$: the $EUCS$
output: the set of high-utility itemsets

```
1  foreach itemset Px ∈ ExtensionsOfP do
2      if SUM(Px.utilitylist.iutils)+SUM(Px.utilitylist.rutils) ≥ minutil then
3          ExtensionsOfPx ← ∅;
4          foreach itemset Py ∈ ExtensionsOfP such that y ≻ x do
5              if ∃(x, y, c) ∈ EUCS such that c ≥ minutil) then
6                  Pxy ← Px ∪ Py;
7                  Pxy.utilitylist ← Construct (P, Px, Py);
8                  ExtensionsOfPx ← ExtensionsOfPx ∪ {Pxy};
9                  if SUM(Pxy.utilitylist.iutils) ≥ minutil then output Px;
10             end
11         end
12         Search (Px, ExtensionsOfPx, minutil);
13     end
14 end
```

We now explain how the search procedure is modified to mine only MinHUIs, rather than all HUIs. The first modification is to the main MinFHM procedure (Algorithm 1). During the first database scan, the utility of each single item is now calculated. Then, each item x that is a high-utility itemset is directly output. The reason is that each such item x is a MinHUI , since no smaller itemset can be a HUI. Thereafter, each such item x is removed from the set I (and thus will not be inserted in I^*). Thus, no superset of x will be explored by the *Search* procedure, and item x will be ignored in TWU and remaining utility calculations, afterward. The reason for removing item x from I is that if x is a HUI, then all supersets of x are not MinHUIs according to Property 5. By applying the previous modification, the algorithm will correctly output MinHUIs that are single items. To find MinHUIs having more than one item, modifications are made to the *Search* procedure (Algorithm 2) as follows. A new structure called the *MinHUI-store* is introduced. At any time, this structure stores the itemsets, which are currently considered to be MinHUIs. When a new HUI Pxy is found,

Algorithm 3: The Construct procedure

input : P: an itemset, Px: the extension of P with an item x, Py: the extension of P with an item y
output: the utility-list of Pxy

```
1  UtilityListOfPxy ← ∅;
2  foreach tuple ex ∈ Px.utilitylist do
3      if ∃ey ∈ Py.utilitylist and ex.tid = exy.tid then
4          if P.utilitylist ≠ ∅ then
5              Search element e ∈ P.utilitylist such that e.tid = ex.tid.;
6              exy ← (ex.tid, ex.iutil + ey.iutil − e.iutil, ey.rutil);
7          end
8          else
9              exy ← (ex.tid, ex.iutil + ey.iutil, ey.rutil);
10         end
11         UtilityListOfPxy ← UtilityListOfPxy ∪ {exy};
12     end
13 end
14 return UtilityListPxy;
```

the modified algorithm checks if there exists an itemset Y in the *MinHUI-store* such that $Y \subset Pxy$. If there exists such an itemset Y, then Pxy is not a MinHUI. Thus, Pxy is not output. Moreover, by Property 5, supersets of Pxy are also not MinHUIs. Thus, Pxy is not added to the set *ExtensionsOfPx*, to ensure that extension of Pxy will not be considered by the search procedure. If there does not exist an itemset Y such that $Y \subset Pxy$, then Pxy is assumed to be a MinHUI. The itemset Pxy is thus inserted into the *MinHUI-store*. Then, the modified algorithm removes each itemset Z in the *MinHUI-store* such that $Pxy \subset Z$, because each such itemset Z is no longer a MinHUI, after the discovery of Pxy. When the algorithm terminates, all MinHUIs in the left-store are output. The union of these itemsets with the single items that are MinHUIs (which have been previously output), are the full set of MinHUIs. By the definition and properties presented in this paper, it can easily be seen that this algorithm is correct and complete for mining MinHUIs.

To further optimize the MinFHM algorithm, it is important to implement the *MinHUI-store* structure efficiently. In our implementation, it is implemented as a list of lists of itemsets. More specifically, the *MinHUI-store* structure stores itemsets having the same size in the same list of itemsets. This allows to efficiently check if an itemset Pxy has proper supersets (subsets) in the *MinHUI-store*, by only comparing Pxy with larger (smaller) itemsets. Furthermore, to be able to quickly compare two itemsets, items in itemsets are lexicographically ordered. Another optimization is that it is not necessary to check if a HUI containing two items has a subset in the *MinHUI-Store*, since MinHUIs of size 1 are not used to generate larger itemsets. Thus, HUI of two items found by the search procedure can be directly assumed to be MinHUIs. Finally, the LA-Prune optimization [9]

is also incorporated. Moreover, for each MinHUI $\{x, y\}$ of size 2 that is found, the corresponding tuple in the EUCS can be replaced by $(x, y, 0)$ to help prune the search space.

5 Experimental Study

We assessed the performance of MinFHM on a computer with a third generation 64 bit Core i5 processor running Windows 7 and 5 GB of free RAM. We compared the performance of the proposed MinFHM algorithm with FHM [5], CHUD [16], and GHUI-Miner [6], which are respectively the state-of-the-art algorithms for mining HUIs, CHUIs and GHUIs. All memory measurements were done using the Java API. The experiment was carried on four real-life datasets commonly used in the HUIM literature: *mushroom*, *retail*, *kosarak* and *foodmart*. These datasets have varied characteristics and represent the main types of data typically encountered in real-life scenarios (dense, sparse and long transactions). Let $|I|$, $|D|$ and A represents the number of transactions, distinct items and average transaction length. *mushroom* is a dense dataset ($|I|$ = 16,470, $|D|$ = 88,162, $A = 23$). *kosarak* is a dataset that contains many long transactions ($|I|$ = 41,270, $|D|$ = 990,000, $A = 8.09$). *retail* is a sparse dataset with many different items ($|I|$ = 16,470, $|D|$ = 88,162, A = 10,30). *foodmart* is a sparse dataset ($|I|$ = 1,559, $|D|$ = 4,141, $A = 4.4$). *foodmart* contains real external and internal utility values. For the other datasets, external utilities for items are generated between 1 and 1,000 by using a log-normal distribution and quantities of items are generated randomly between 1 and 5, as the settings of [2, 12, 15]. The source code of all algorithms and datasets can be downloaded as part of the SPMF open-source data mining library [7] at `http://www.philippe-fournier-viger.com/spmf/`. Algorithms were run on each dataset, while decreasing the *minutil* threshold until they became too long to execute, ran out of memory or a clear trend was observed. Fig. 2 shows the execution times of MinFHM, FHM, CHUD, and GHUI-Miner. Fig. 3 compares the number of MinHUIs, HUIs, CHUIs and GHUIs, respectively generated by these algorithms.

It can first be observed that mining MinHUIs using MinFHM is faster than mining HUIs, CHUIs and GHUIs, using FHM, CHUD and GHUI-Miner. On *mushroom*, MinFHM is up to 824, 44, and 71 times, faster than FHM, CHUD and GHUI-Miner. On *foodmart*, MinFHM is up to 80, 52, and 75 times faster than FHM, CHUD and GHUI-Miner. On *retail*, MinFHM is up to 6, 62, and 63 times faster than FHM, CHUD and GHUI-Miner. On *kosarak*, MinFHM is up to 1.8, 15, and 16 times faster than FHM, CHUD and GHUI-Miner. The reason for the excellent performance of MinFHM is that it prunes a large part of the search space by not exploring the transitive extensions[5] of MinHUIs.

A second observation is that MinFHM scales well when *minutil* is decreased. For example, on *mushroom*, the runtime of MinFHM does not vary much and remains less than 1 second, while the runtime of FHM increases rapidly as *minutil*

[5] Recall that for an itemset X, the *extensions*of X are the itemsets that can be obtained by appending an item y to X such that $y \succ i$, $\forall i \in X$.

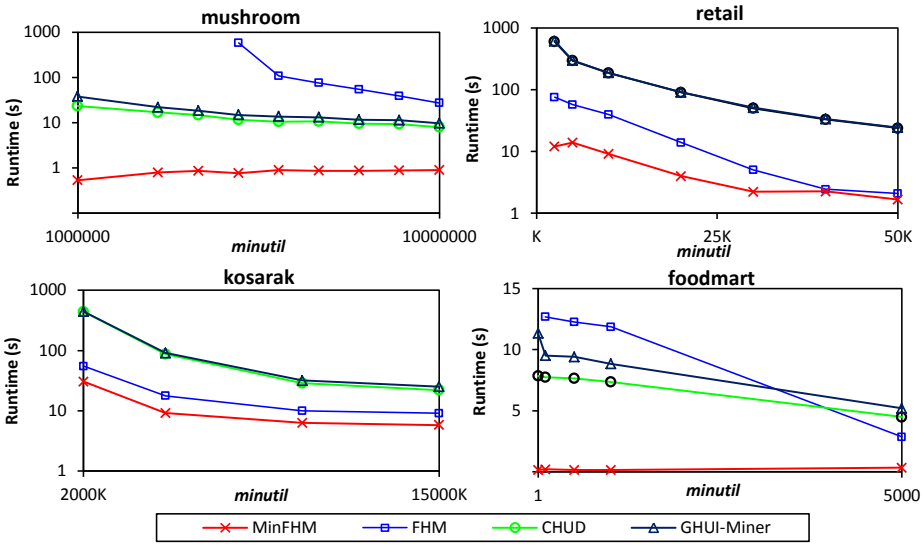

Fig. 2. Execution times

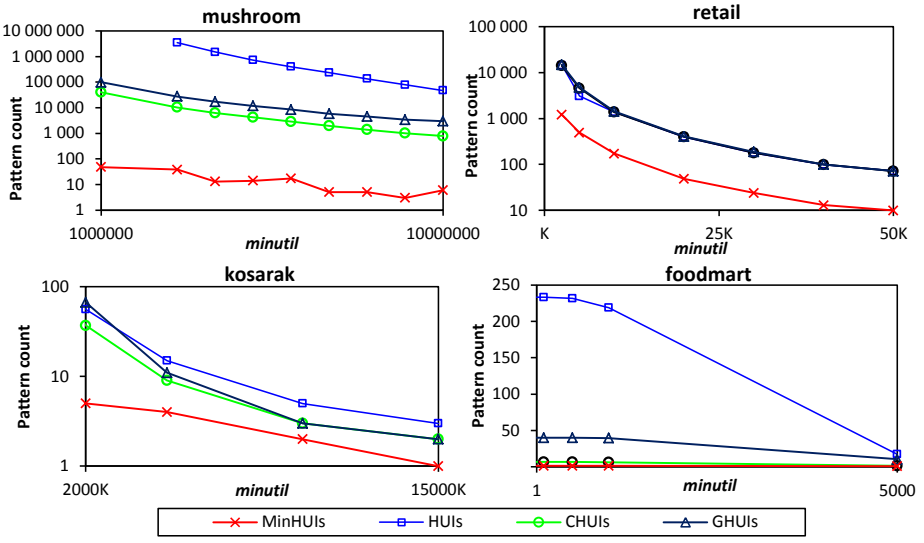

Fig. 3. Number of patterns found

is decreased, taking more than 10 minutes to terminate. MinFHM shows a similar behavior on *foodmart* dataset, where the runtime of MinFHM is very stable while the runtimes of other algorithms increase considerably when *minutil* is decreased. On the *retail*, the runtime of MinFHM increases by a lesser amount compared to the other algorithms when *minutil* is decreased. Finally, on the

Kosarak, the increase is comparable to the other algorithms. The reason why the runtime of MinFHM is generally very stable is that when *minutil* is decreased, the number of MinHUIs generally increases less rapidly than the number of HUIs, CHUIs and GHUIs (see Fig. 3). As mentioned in property 4, MinHUIs have the nice property that their number may increase or decrease, as *minutil* is decreased, while the numbers of HUIs, CHUIs and GHUIs cannot decrease, and generally increase very quickly.

It is also interesting to observe that the number of MinHUIs never exceeded 1,300 patterns, while other algorithms generated up to millions of patterns. For example, on the dense *mushroom* dataset and $minutil = 3,000,000,38$ MinHUIs, 3,538,181 HUIs, 10,311 CHUIs, and 27,640 GHUIs, are found. The number of MinHUIs is thus respectively, 931,000, 271, and 727 times less than the number of HUIs, CHUIs and GHUIs. It can thus be concluded that HUIs, CHUIs and GHUIs, generally depend on a very small set of MinHUIs, and that finding these MinHUIs provides a very compact and informative set of results to the user.

6 Conclusion

This paper has studied a novel representation of high-utility itemsets named Minimal High-Utility Itemsets (MinHUIs), its properties, and presented an efficient algorithm named MinFHM to discover MinHUIs. MinFHM includes numerous optimizations to discover MinHUIs efficiently. An extensive experimental study on real-life datasets shows that mining minimal HUIs is almost two orders of magnitude faster than mining HUIs, CHUIs or GHUIs and that it can greatly reduce the result set presented to the user. The source code of all algorithms and datasets can be downloaded as part of the SPMF open-source data mining library [7] at `http://www.philippe-fournier-viger.com/spmf/`.

For future work, an interesting possibility is to use MinHUIs as a negative border in HUI stream mining and incremental HUI mining, and also to explore the properties of MinHUIs for associative classifiers [18], and the discovery of minimal high-utility sequential patterns [19, 20]. Lastly, another possibility is to design a faster algorithm for mining MinHUIs based on EFIM [21], a recently-proposed algorithm that was shown to outperform FHM for the traditional problem of HUI mining.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. Int. Conf. Very Large Databases, pp. 487–499, (1994)
2. Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., Lee, Y.-K.: Efficient Tree Structures for High-utility Pattern Mining in Incremental Databases. In: IEEE Trans. Knowl. Data Eng. 21(12), pp. 1708–1721 (2009)
3. Deng, Z.: DiffNodesets: An efficient structure for fast mining frequent itemsets. Appl. Soft Computing 41, pp. 214–223 (2016)

4. Deng, Z., Lv, S.-H.: PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via Children-Parent Equivalence pruning. Expert Syst. Appl. 42(13), pp. 5424–5432 (2015)
5. Fournier-Viger, P., Wu, C.-W., Zida, S., Tseng, V. S.: FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In: Proc. 21st Int. Symp. on Methodologies for Intell. Syst., pp. 83–92 (2014)
6. Fournier-Viger, P., Wu, C.-W., Tseng, V. S. : Novel Concise Representations of High Utility Itemsets using Generator Patterns. Proc. of 10th Int. Conf. on Advanced Data Mining and Applications, pp. 30–43 (2014)
7. Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu., C., Tseng, V. S.: SPMF: a Java Open-Source Pattern Mining Library. Journal of Machine Learning Research (JMLR), 15, pp. 3389-3393 (2014)
8. Lan, G. C., Hong, T. P., Tseng, V. S.: An efficient projection-based indexing approach for mining high utility itemsets. Knowl. Inform. Syst. 38(1), 85–107 (2014)
9. Krishnamoorthy, S.: Pruning strategies for mining high utility itemsets. Expert Systems with Applications, 42(5), 2371-2381 (2015)
10. Li, Y.-C., Yeh, J.-S., Chang, C.-C.: Isolated items discarding strategy for discovering high utility itemsets. In: Data & Knowledge Engineering. 64(1), pp. 198–217 (2008)
11. Song, W., Liu, Y., Li, J.: BAHUI: Fast and memory efficient mining of high utility itemsets based on bitmap. Int.. J. Data Warehous. 10(1), 1–15 (2014)
12. Liu, M., Qu, J.: Mining high utility itemsets without candidate generation. In: Proc. 22nd ACM Int. Conf. Info. and Know. Management, pp. 55–64 (2012)
13. Liu, Y., Liao, W., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Proc. 9th Pacific-Asia Conf. on Knowl. Discovery and Data Mining, pp. 689–695 (2005)
14. Shie, B.-E., Yu, P.S., Tseng, V.S.: Efficient algorithms for mining maximal high utility itemsets from data streams with different models. Expert Syst. Appl. 39(17), pp. 12947-12960 (2012)
15. Tseng, V. S., Shie, B.-E., Wu, C.-W., Yu., P. S.: Efficient algorithms for mining high utility itemsets from transactional databases. IEEE Trans. Knowl. Data Eng. 25(8), 1772–1786 (2013)
16. Tseng, V., Wu, C., Fournier-Viger, P., Yu, P.: Efficient algorithms for mining the concise and lossless representation of closed+ high utility itemsets. IEEE Trans. Knowl. Data Eng. 27(3), 726–739 (2015)
17. Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets, Proc. ICDM'04 Workshop on Frequent Itemset Mining Implementations, CEUR (2004).
18. Nguyen, D., Vo, B., Le, B.: CCAR: An efficient method for mining class association rules with itemset constraints. Engineering Applications of Artificial Intelligence, 37, 115–124 (2015)
19. Yin, J., Zheng, Z., Cao, L.: USpan: An Efficient Algorithm for Mining High Utility Sequential Patterns. In: Proc. 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 660–668 (2012)
20. Zida, S., Fournier-Viger, P., Wu, C.-W., Lin, J. C. W., Tseng, V.S.: Efficient mining of high utility sequential rules. In: Proc. 11th Intern. Conf. Machine Learning and Data Mining, pp. 1–15 (2015)
21. Zida, S., Fournier-Viger, P., Lin, J. C.-W., Wu, C.-W., Tseng, V.S.: EFIM: A Highly Efficient Algorithm for High-Utility Itemset Mining. Proc. 14th Mexican Intern. Conf. on Artificial Intelligence, pp. 530–546.