Novel Concise Representations of High Utility Itemsets Using Generator Patterns

Philippe Fournier-Viger[1], Cheng-Wei Wu[2], and Vincent S. Tseng[2]

[1] Dept. of Computer Science, University of Moncton, Canada
[2] Dept. of Comp. Sci. and Info. Eng., National Cheng Kung University, Taiwan
philippe.fournier-viger@umoncton.ca, silvemoonfox@hotmail.com, tseng@mail.ncku.edu.tw

Abstract. Mining *High Utility Itemsets* (*HUIs*) is an important task with many applications. However, the set of HUIs can be very large, which makes HUI mining algorithms suffer from long execution times and huge memory consumption. To address this issue, concise representations of HUIs have been proposed. However, no concise representation of HUIs has been proposed based on the concept of *generator* despite that it provides several benefits in many applications. In this paper, we incorporate the concept of generator into HUI mining and devise two new concise representations of HUIs, called *High Utility Generators* (*HUGs*) and *Generator of High Utility Itemsets* (*GHUIs*). Two efficient algorithms named *HUG-Miner* and *GHUI-Miner* are proposed to respectively mine these representations. Experiments on both real and synthetic datasets show that proposed algorithms are very efficient and that these representations are up to 36 times smaller than the set of all HUIs.

Keywords: pattern mining, high utility itemset mining, concise representation, high utility generator, generator of high utility itemsets.

1 Introduction

High Utility Itemset Mining (*HUIM*) [2,4,6,13] is an important research topic in data mining. As opposed to *Frequent itemset Mining* (*FIM*) [1], HUIM considers the importance of items (e.g. unit profit) and their quantities in transactions. Therefore, it can be used to discover itemsets having a high utility (e.g. high profit), that is *High Utility Itemsets* (*HUIs*). HUIM has a wide range of applications such as cross-marketing and click stream analysis and biomedical applications [2,4,6,13]. The problem of HUIM is widely recognized as more difficult than that of FIM. In FIM, the *downward-closure property* states that the support (frequency) of an itemset is *anti-monotonic* [1], that is the supersets of an infrequent itemset are infrequent. This property is very powerful to prune the search space. In HUIM, the utility of an itemset is neither monotonic or anti-monotonic. That is, a HUI may have a superset or subset with lower, equal or higher utility [2,4,6,13]. Thus techniques to prune the search space developed in FIM cannot be directly applied to HUIM.

X. Luo, J.X. Yu, and Z. Li (Eds.): ADMA 2014, LNAI 8933, pp. 30–43, 2014.

Many studies have been carried to develop efficient HUIM algorithms [2,4,6,7,13]. However, a crucial problem of HUIM is that the set of HUIs generated by these algorithms can be very large. This makes HUIM algorithms suffer from long execution times and even fail to run due to huge memory consumption or lack of storage space. Moreover, it is very inconvenient for a user to analyze a very large set of HUIs. To address this issue, it was proposed to mine concise representations of HUIs rather than all HUIs. *GUIDE* [9] is an approximate algorithm that integrates the concept of *maximal pattern* from FIM to mine *maximal HUIs* (HUIs having no proper supersets that are high utility). Another work is *CHUD* [14], which adapts the concept of *closed pattern* from FIM to discover *closed HUIs* (HUIs having no proper supersets that are HUIs and have the same support). Although these representations are useful, no work has been done yet on integrating the concept of *generator pattern* (or simply called *generators*) in HUIM despite that generators have shown to provides several benefits over all/closed/maximal patterns in many applications [5,8,10,11,12].

A generator is an itemset that has no proper subset having the same support. Generators provide the following benefits. First, when generators are combined with closed patterns, they give additional information that closed patterns alone cannot provide, for example to generate minimal rules between patterns with a minimal antecedent (generator) and a maximal consequent (closed pattern) [8]. Second, generators can provide higher classification accuracy and are more useful for model selection than using all or only closed patterns [8]. Third, generators are preferable according to the *Minimum Description Length* principle to closed/maximal patterns, since generators are the minimal members of equivalence classes rather than the maximal ones [5,8,11,12]. Lastly, mining generators is generally more efficient than discovering all patterns because it is generally a very small subset of all patterns [5].

Therefore, several interesting questions are raised: How to integrate the concept of generator in HUIM to define meaningful concise representations of HUIs? How much reduction can be achieved by these representations? Can efficient algorithms be developed to mine these representations? What are the pros and cons of these representations? To answer these questions, we investigate the properties of generators in the context of HUIM and devise two alternative concise representations of HUIs using generators, respectively called *High Utility Generators* (HUGs) and *Generator of High Utility Itemsets* (*GHUIs*). We propose two efficient algorithms named *HUG-Miner* and *GHUI-Miner* to respectively mine these representations, and we analyze their respective advantages in terms of speed and number of patterns found. Experimental results on both real and synthetic datasets show that the proposed algorithms are very efficient and that the mined representations are up to 36 times smaller than the set of HUIs.

The rest of this paper is organized as follows. Section 2, 3, 4 and 5 respectively presents the problem definition and related work, the proposed algorithms, the experimental evaluation and the conclusion.

TID	Transactions
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)

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

Fig. 1. A transaction database (left) and external utility values (right)

2 Problem Definition and Related Work

2.1 High Utility Itemset Mining

Let I be a set of items. A *transaction database* is a set of transactions $D = \{T_0, T_1, ..., T_n\}$ such that for each transaction T_c, $T_c \in I$ 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* (e.g. unit profit). 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 (e.g. purchase quantity). For example, Fig. 1 show a transaction database containing five transactions ($T_0, T_1 ... T_4$). External utilities of items a, c, and e in T_2 are $q(a, T_2) = 2$, $q(c, T_2) = 6$ and $q(e, T_2) = 2$. Fig. 1 (right) indicates that the external utility of a, b, e are respectively $p(a) = 5$, $p(c) = 1$ and $p(e) = 3$.

Definition 1 (Utility of an itemset). The *utility of an item i* in a transaction T_c is denoted as $u(i, T_c)$ and defined as $p(i) \times q(i, T_c)$. An *itemset* is a set of items. The *utility of an itemset* X in a transaction T_c is defined as $u(X, T_c) = \sum_{i \in X} u(i, T_c)$. The set of transactions containing X is denoted as $g(X)$. The *utility of X in a database* is defined as $u(X) = \sum_{T_c \in g(X)} u(X, T_c)$.

Example 1. The utility of the itemset $\{a, e\}$ in the transaction T_0 is $u(\{a, e\}, T_0)$ $= u(\{a\}, T_0) + u(\{e\}, T_0) = 1 \times 5 + 1 \times 3 = 8$. The utility of $\{a, e\}$ is $u(\{a, e\}, T_0)$ $+ u(\{a, e\}, T_3) = 8 + 16 = 24$.

Definition 2 (Problem of HUI mining). An itemset X is a *high utility itemset* if its utility is no less than a user-specified *minimum utility threshold minutil* given by the user. Otherwise, X is a *low utility itemset*. The *problem of high utility itemset mining* is to discover all high utility itemsets in the database.

Example 2. If $minutil = 25$, the complete 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.

HUIM is harder than FIM because the utility measure is not monotonic or anti-monotonic [2,7,13], i.e., an itemset may have a utility 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 support cannot be directly transfered to HUIM. Several HUIM algorithms circumvent this problem by overestimating the utility of itemsets using the *Transaction-Weighted Utilization (TWU)* measure [2,7,13], which is anti-monotonic, and defined as follows.

Definition 3 (Transaction weighted utilization). The *transaction utility* (TU) of a transaction T_c is the sum of the utility 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 utility of transactions containing X, i.e. $TWU(X) = \sum_{T_c \in g(X)} TU(T_c)$.

Example 3. 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. $TWU(\{c, d\}) = TU(T_0) + TU(T_1) + TU(T_2) = 25 + 20 + 8 = 53$.

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. [7]

Algorithms such as *Two-Phase* [7], *IHUP* [2] and *UP-Growth* [13] utilizes the above 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. In the second phase, they scan the database to calculate the exact utility of all candidates found in the first phase to eliminate low utility itemsets. Recently, an alternative approach called *HUI-Miner*[6] was proposed to mine HUIs directly using a single phase. A faster algorithm named FHM was then proposed. FHM is to our knowledge the fastest algorithm for mining HUIs[4]. It utilizes the depth-first search procedure of HUI-Miner [6] to explore the search space of HUIs but introduces an additional optimization named EUCP [4] that makes FHM up to 6 tims faster than HUI-Miner. In FHM, each itemset is associated with a structure named *utility-list* [4,6]. Utility-lists allow calculating the utility of an itemset quickly by making join operations with utility-lists of shorter patterns. utility-lists a re defined as follows.

Definition 4 (Utility-list). 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 $\sum_{i \in T_{tid} \wedge i \succ x \forall x \in X} u(i, T_{tid})$.

Example 4. 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. The join operation for single items is performed as follows. Consider two items x, y such that $x \succ y$, and their utility-lists $ul(\{x\})$ and $ul(\{y\})$. The utility-list of $\{x, y\}$

is obtained by creating a tuple $(ex.tid, ex.iutil + ey.iutil, ey.rutil)$ for each pairs of tuples $ex \in ul(\{x\})$ and $ey \in ul(\{y\})$ such that $ex.tid = ey.tid$. The join operation for two itemsets $P \cup \{x\}$ and $P \cup \{y\}$ such that $x \succ y$ is performed as follows. Let $ul(P)$, $ul(\{x\})$ and $ul(\{y\})$ be the utility-lists of P, $\{x\}$ and $\{y\}$. The utility-list of $P \cup \{x, y\}$ is obtained by creating a tuple $(ex.tid, ex.iutil + ey.iutil - ep.iutil, ey.rutil)$ for each set of tuples $ex \in ul(\{x\})$, $ey \in ul(\{y\})$, $ep \in ul(P)$ such that $ex.tid = ey.tid = ep.tid$. 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 [6].

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 [6].

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.

2.2 Concise Representations of High Utility Itemsets

To discover small and representative subsets of all HUIs, concise representations of HUIs such as *closed HUIs*[14] and *maximal HUIs*[9] have been proposed, which are defined as follows.

Definition 5 (Closed HUIs and maximal HUIs). The *support* of an itemset X in a database D is denoted as $sup(X)$ and defined as $|g(X)|$. A HUI X is a *closed HUI* (*CHUI*) [14] iff there exists no HUI Y such that $X \subset Y$ and $sup(X) = sup(Y)$. A HUI X is a *maximal HUI* (*MHUI*) [9] iff there exists no HUI Y, such that $X \subset Y$.

Although these representations are useful in some applications, to our knowledge, no work has been done on integrating the concept of generator pattern from FIM in HUIM, despite that generators provides several benefits over closed and maximal patterns [5,8,11,12]. In FIM, an itemset X is a *generator* (a.k.a *key pattern* or *minimal pattern*) iff there is no itemset Y such that $Y \subset X$ and $sup(X) = sup(Y)$ [5,10,11]. A generator pattern X is the generator of an itemset Y if $X \subseteq Y$ and $sup(X) = sup(Y)$. The concept of generator pattern is directly related to the concept of closed pattern [10,11]. An alternative and equivalent definition of generator patterns and closed patterns is the following. Let an *equivalence class* be the set of all itemsets supported by the same set of transactions. Generator patterns are the minimal members of each equivalence class, while closed patterns are the maximal members of each equivalence class. For example, consider the equivalence class $\{\{a, e\}, \{a, c, e\}\}$ of itemsets appearing in T_0 and T_2. {a,e} is a generator and {a,c,e} is the closed pattern. Each

equivalence class contains one or more generators and a single closed pattern [11] called their *closure*. Thus, the *closure of an itemset* X is the unique closed pattern Y such that $sup(X) = sup(Y) \ \wedge X \subseteq Y$. Algorithms for mining generator patterns uses the following property to prune the search space [10,11].

Property 4 (Downward closure for generator patterns). An itemset X is not a generator pattern if there exists a strict subset of X that is not a generator [11].

2.3 Integrating the Concept of Generator in HUIM

To understand how the concept of generator can be applied to HUIM, consider the equivalence classes shown in Fig. 2 for the running example. Each equivalence class is represented as a Hasse diagram inside a rectangle and is labelled with the supporting transactions and support of its itemsets. For example, the equivalence class of T_0, T_1 and T_4 contains itemsets $\{b\}$, $\{b, c\}$, $\{b, e\}$ and $\{b, c, e\}$, which have a support of 3. In this figure, HUIs are represented as shapes with a solid line. The first important observation is that some equivalence classes (not shown in the figure) do not contain any HUIs. We name generators appearing in those equivalence classes *l-generators* (*LGs*). It is clear that LGs should not be mined. For example, $\{d\}$ should not be mined since its equivalence class do not contain any HUIs. The second important observation is that some equivalence classes contain at least one HUI. We name generators appearing in these equivalence classes *Generators of High Utility Itemsets* (*GHUIs*). It can be observed that some GHUIs are HUIs and other are not HUIs. We use the term *High Utility Generator* (*HUGs*) to refer to those that are HUIs and the term *Low Utility Generator* (*LUGs*) to refer to those that are not HUIs. LUGs, HUGs, GHUIs and CHUIs for the running example are illustrated in Fig. 2.

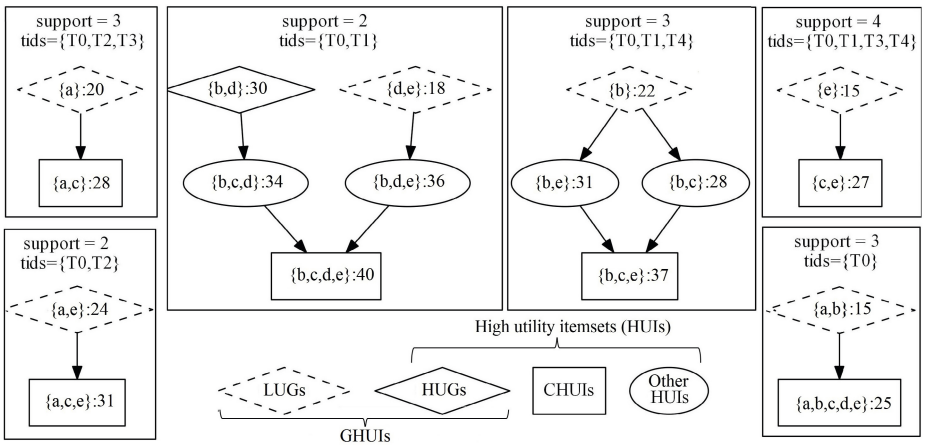

Fig. 2. HUIs and their equivalence classes (represented as Hasse diagrams)

Based on these definitions, it can be clearly seen that the set of all LUGs and the set of all HUGs are disjoint, and are subsets of the set of all GHUIs. Moreover, the set of all GHUIs is disjoint from the set of LGs. Furthermore, three important properties are:

Property 5 (Utility of itemsets in an equivalence class). Consider an equivalence class Q. Given that all itemsets in Q appear in the same set of transactions, it can be demonstrated that if $X \subset Y$ then $u(X) < u(Y)$, $\forall X, Y \in Q$.

Property 6 (utility of supersets of a HUG). In each equivalence class, supersets of a HUG are HUIs.

Property 7 (Highest utility in an equivalence class). In each equivalence class, the closure has the highest utility among all itemsets [14].

In the following, we are interested in mining the concise representations of all GHUIs and that of all HUGs. The set of GHUIs is meaningful following the *Minimum Description Length* principle since GHUIs describe each set of transactions (equivalence class) containing HUIs using minimal sets of items. However, as we will show mining LUGs is more expensive than mining HUGs because LUGs may have a very low utility. For this reason and because HUGs provides the interesting property that all supersets of a HUG are high utility, we also consider mining only HUGs.

3 The GHUI-Miner and HUG-Miner Algorithms

In this section, we first propose an algorithm named GHUI-Miner for mining all GHUIs. Then we explain how a variation named HUG-Miner is devised to mine only HUGs.

Mining GHUIs is a very difficult problem. For the reader, it may seems that GHUIs could be obtained by simply adding a generator checking mechanism used in FIM to a HUIM algorithm. However, doing so would result in an incomplete algorithm. The reason is that HUIM algorithms attempts to only explore HUIs. Thus, only HUGs would be found. To mine all GHUIs, it is thus necessary to design an algorithm that considers a larger search space, that avoids pruning LUGs. Moreover, another important challenge is that when a low utility generator is considered, checking if it is a LUGs requires to verify if there is a HUI in its equivalence class. However, state-of-the-art algorithms for generator mining [10,11] uses a depth-first search and do not keep equivalence classes into memory (maintaining them would be inefficient). Thus, it is not possible to compare an itemset with all the other members of its equivalence class efficiently. An idea that we propose to circumvent this challenge is to only compare a generator with its closure. Since the closure has the highest utility in each equivalence class (Property 7), it is sufficient to only consider the closure to determine if a low utility generator is a LUG. However, even with this solution, a problem is how to obtain the closure of a generator. In FIM, no depth-first search algorithm compute closure and generators at the same time and it cannot be done easily since

closed patterns mining algorithms do not visit all generators. In fact, the state-of-the-art algorithm for mining generators and their closures [12] is a compound algorithm that mines closed pattern and generators separately by applying two algorithms and then links generators to closed patterns by post-processing. This approach is not a viable solution for GHUI mining since it would require keeping too many candidate generators in memory. To address this issue, we initially attempted to compute the closure of each found low utility generator on-the-fly. This can be done by intersecting transactions containing the generator. However, even on moderately large datasets, this approach cannot terminate in reasonable time due to the high cost of closure calculation. A better solution that we use in GHUI-Miner is to first mine CHUIs by using a CHUI mining algorithm and then to mine generators using a modified FHM [4] search procedure. The procedure is modified to only explore generator patterns by adding an efficient generator checking mechanism. Furthermore, the procedure is modified to only explores itemsets that are subsets of CHUIs, thus greatly pruning the search space. When the algorithm produces a generator X, if X is high utility, it is output. If X is low utility, its corresponding CHUI is retrieved to quickly determine if X is a LUG that should be output. Furthermore, several pruning conditions are added to further prune the search space.

Algorithm 1. The GHUI-Miner algorithm

input : D: a transaction database, *minutil*: a user-specified threshold, $CHUIs$: the set of CHUIs
output: the set of GHUIs

1 $I_{closed} \leftarrow$ items appearing in CHUIs.;
2 Scan D to calculate the TWU of items in I_{closed};
3 Let $\succ$ be the total order of TWU ascending values on I_{closed};
4 Scan D to build the utility-list of each item $i \in I_{closed}$ and the E structure;
5 **if** $\exists C \in CHUIs$ *such that* $sup(C) = |D|$ **then** `Output` $(\emptyset)$;
6 **foreach** $\{i\} \in I_{closed}$ **do**
7 **if** `IsGenerator` $(\{i\}) = false \vee$ *SUM(i.utilitylist.iutil) + SUM(i.utilitylist.rutil)* $<$ *minutil* **then** $I_{closed} \leftarrow I_{closed} \setminus \{\{i\}\}$;
8 **else if** *SUM(i.utilitylist.iutil)* $\geq$ *minutil* **then**
9 `Output` $(\{i\})$;
10 **if** `IsNotStrictSubsetOfACHUI` $(\{i\}, CHUIs)$ **then**
11 $I_{closed} \leftarrow I_{closed} \setminus \{\{i\}\}$;
12 **else if** `GetClosureOf` $(\{i\}$, *CHUIs*$) \neq null$ **then** `Output` $(\{i\})$
13 **end**
14 `SearchGHUI` (I_{closed}, *minutil*, E, $CHUIs$);

Main Procedure of GHUI-Miner. The main procedure of GHUI-Miner (Algorithm 1) takes as input a transaction database D, the *minutil* threshold and the set of CHUIS. The set of CHUIs need to be first mined using an algorithm for mining CHUIs (CHUD [14] in our implementation). GHUI-Miner first select the set of items appearing in CHUIs (I_{closed}) because only these items may appear

in GHUIs. Then, the algorithm scans D to calculate the TWU of each of those items. Note that considering only items in I_{closed} for TWU calculation reduces the TWU values (an thus provide a more tight upper bounds on the utility). 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 [6]). A second database scan is then performed. During this database scan, items in transactions are re-ordered according to the total order $\succ$, the utility-list of each item $i \in I_{closed}$ is built, and a structure named E is created. The E structure stores the TWU of all pairs of items $\{a, b\}$ such that $u(\{a, b\}) \neq 0$. The E structure could be implemented as a triangular matrix. But, for memory efficiency, we have implemented it as a hashmap of hashmaps since in practice a limited number of pairs of items co-occurs in transactions. After the construction of E, the empty set is output if there is a CHUIs with its support equal to $|D|$ (meaning that the empty set is a LUG). Then, a loop is performed to consider each item $\{i\} \in I_{closed}$. A call to the method $IsGenerator$ is made to verify if $\{i\}$ is a generator (this method will be explained later). If $\{i\}$ is not a generator, then it is removed from I_{closed} because any superset of a non generator cannot be a generator (Property 4). If the sum of *iutil* and *rutil* values in the utility-list of $\{i\}$ is less than *minsup*, $\{i\}$ and all its supersets are low utility, $\{i\}$ is discarded (Property 3). Then, the algorithm checks if $\{i\}$ is a GHUI. If the sum of *iutil* values in the utility-list of $\{i\}$ is no less than *minutil*, $\{i\}$ is a HUG and it is output. If $\{i\}$ is a HUG and $\{i\}$ is not a strict subset of a CHUI, it is removed from I_{closed} because none of its supersets can be a GHUI. If $\{i\}$ is not a HUG, an attempt is made to retrieve the closure of $\{i\}$ from $CHUIs$ (i.e. to retrieve a set Z such that $\{i\} \subseteq Z \wedge sup(\{i\}) = sup(Z)$. If the closure is found, $\{i\}$ is a LUG and it is output. Then, a recursive depth-first search exploration of generators having more than 1 item starts by calling the procedure $SearchGHUI$ with the set of single items I_{closed}, *minutil* and the E structure.

The $SearchGHUI$ **Procedure.** The $SearchGHUI$ procedure (Algorithm 2) takes as input (1) a set of itemsets called $ExtensionsOfP$ having the form Pw meaning that Pw was previously obtained by appending an item w to an itemset P, (2) *minutil*, (3) the E structure and (4) the set of CHUIs. A loop is first performed over each itemset $Px \in ExtensionsOfP$ to explore its extensions. This is performed by merging Px with all extensions Py of $ExtensionsOfP$ such that $y \succ x$ to form extensions of the form Pxy containing $|Px| + 1$ items. For each extension Pxy, several checks are performed to prune the search space. First, if $TWU(\{x, y\}) < minutil$ according to the E structure, then Pxy and all its supersets are low utility and it can be discarded (Property 1). Second, if Pxy is not a subset of a CHUI, it can be discarded since none of its supersets can be GHUIs. Third, if the support of Pxy is equal to the support of Px or Py, Pxy is not a generator and its supersets cannot be generators (Property 4). Thus, Pxy is discarded. Fourth, if the sum of *iutil* and *rutil* values in Pxy utility-lists are less than *minsup*, Pxy and all its supersets are low utility and Pxy is thus discarded (Property 3). Fifth, if Pxy is not a generator (which is checked using the $IsGenerator$ method to be described later), it is also discarded. Then, the

Algorithm 2. The *SearchGHUI* procedure

input : *ExtensionsOfP*: a set of itemsets having a common prefix P, the *minutil* threshold, the E structure, $CHUIs$: the set of CHUIs
output: the set of high utility itemsets

```
1  foreach itemset Px ∈ ExtensionsOfP do
2      ExtensionsOfPx ← ∅;
3      foreach itemset Py ∈ ExtensionsOfP such that y ≻ x do
4          Pxy ← Px ∪ Py;
5          if TWU(x, y) ≥ minutil according to E ∧ IsSubsetOfACHUI
           (Pxy, CHUIs) ∧ |Pxy.utilitylist| ≠ 0 ∧ |Pxy.utilitylist| ≠
           |Px.utilitylist| ∧ |Pxy.utilitylist| ≠ |Py.utilitylist| ∧
           SUM(i.utilitylist.iutil) + SUM(i.utilitylist.rutil) ≥ minutil ∧
           IsGenerator (Pxy) then
6              if SUM(Pxy.utilitylist.iutil) ≥ minsup then Output (Pxy);
7              else if GetClosureOf ( Pxy, CHUIs) ≠ null then Output (Pxy) ;
8              ExtensionsOfPx ← ExtensionsOfPx ∪ {Pxy}
9          end
10     end
11     SearchGHUI (ExtensionsOfPx, minutil, CHUIs);
12 end
```

algorithm check if the sum of *iutil* values in the utility-list of Pxy is no less than *minutil*. If yes, then Pxy is a HUG and it is output. Otherwise, an attempt is made to retrivee the closure of Pxy from the set of CHUIs. If the closure is found, Pxy is a LUG and it is output. Furthermore, if Pxy is a generator, it is added to a set $ExtensionsOfPx$ for storing itemsets that should be considered for further extensions (Property 4). Finally, a recursive call to the *SearchGHUI* procedure with $ExtensionsOfPx$ is done to explore extension(s) of its itemsets.

Since the *SearchGHUI* procedure starts from single items, it recursively explore the search space of generators by appending single items to find all GHUIs. Though, we do not have space to provide the proofs of completeness and correctness, it can be easily seen that this main procedure is correct and complete for finding GHUIs based on previous definitions and properties.

Updating Utility-Lists. In FHM, the *rutil* values in utility-lists are updated assuming that no item m will be added to extend an itemset X if $m \succ k$ for an item $k \in X$. This is a correct assumption for FHM for pruning the search space using Property 3 since items are added to itemsets following the total order $\succ$. However, in GHUI-Miner, we need to consider a larger search space that avoids pruning an itemset X if its closure is a HUI (to keep LUGs), and the closure may contains such an item m. To solve this problem, the *rutil* element of a tuple for an itemset X and a tid *tid* is redefined as $\sum_{i \in T_{tid} \wedge i \notin X} u(i, T_{tid})$. Moroever, utility-lists for the join of two itemsets Px and Py is done as follows. Let $ul(\{x\})$ and $ul(\{y\})$ be the utility-lists of Px and Py. The utility-list of $P \cup \{x, y\}$ is obtained by creating a tuple $(ex.tid, ex.iutil + ey.iutil - ep.iutil, ex.rutil - ey.iutil)$ for each set of tuples $ex \in ul(\{x\})$, $ey \in ul(\{y\})$, $ep \in ul(P)$ such that $ex.tid = ey.tid$.

The $IsGenerator$ **Procedure.** GHUI-Miner integrates an efficient mechanism to determine if an itemset is a generator on-the-fly, that is, without having to compare an itemset with its subsets. The mechanism is inspired by the one used in the DefMe algorithm [10], although this later is desgined for a different search procedure. To describe the generator checking mechanism in GHUI-Miner, we introduce the concept of critical transactions. For an itemset X, the *critical transactions* of an item e in X are denoted as $crit(X, e)$ and defined as $g(X \backslash \{e\}) \backslash g(e)$, where $g(X)$ is the set of transactions containing X. Based on this definition, it can be easily seen that a necessary and sufficient condition for an itemset X to be a generator is that $\forall e \in X$, $crit(X, e) \neq \emptyset$. Performing generator checking using this condition requires to be able to compute critical transactions for any itemset efficiently. Fortunately, it can be done by modifying the GHUI-Miner search procedure. Let X be an itemset. It can be demonstrated that for any items a, b, $crit(X \cup \{b\}, a) = crit(X, a) \cap g(\{b\})$. Consider the join of a pair of itemsets $P \cup \{x\}$ and $P \cup \{y\}$ by GHUI-Miner to generate $P \cup \{x, y\}$. The critical objects of $P \cup \{x, y\}$ with respect to an item $z \in P$ can be calculated using $crit(P \cup \{x\}, z)$ and $g(y)$. Thus critical transactions of an itemset can be calculated using critical transactions of the pairs of itemsets that were joined to obtain the new itemset. Note that for an itemset $\{i\}$ containing a single item, $crit(\{i\}, i) = g(\emptyset) \backslash g(i)$. This generator checking mechanism is efficient. In GHUI-Miner, critical transactions are represented as bitsets for memory-efficiency. Moreover, another optimization is to stop generator checking for an itemset as soon as the itemset is determined to not be a generator (a set of critical transactions is found to be empty).

The $IsSubsetOfACHUI$ **and** $GetClosure$ **Procedures.** These procedures are implemented efficiently by storing CHUIs in a structure that indexes itemsets by their size and their support (a list of lists in our implementation). Thus, when searching for the closure of an itemset X, only itemsets of size greater or equal to $|X|$ and having a support equal to $sup(X)$ are considered. Similarly, when searching for a (strict) superset that is CHUI, only itemsets of size greater than (or equal to) $|X|$ and having a support smaller or equal to $sup(X)$ are considered.

The $HUG-Miner$ **Algorithm.** We also propose a variation of GHUI-Miner named HUG-Miner to mine only HUGs, which does not includes instructions specific to LUGs. More precisely, HUG-Miner does not take the set of CHUIs as parameter and it considers the set of items I instead of I_{closed}. In Algorithm 1, lines 5, 10 and 11 are removed. In Algorithm 2, the call to $IsSubsetOfACHUI$ and line 7 are removed. Moreover, HUG-Miner updates utility-list as described in section 2.1 since it only need to find HUIs.

4 Experimental Study

We performed an experiment to assess the performance of GHUI-Miner and HUG-Miner, and analyze their respective advantages. The experiment was performed 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

GHUI-Miner and HUG-Miner with the state-of-the-art algorithm FHM for high utility itemset mining. CHUD [14] was used to generate the CHUIs needed by GHUI-Miner. All memory measurements were done using the Java API. The experiment was carried on four real-life datasets commonly used in the HUIM litterature: *mushroom*, *retail*, *kosarak* and *foodmart*. These datasets have varied characteristics and represents 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 [2,6,13]. The source code of all algorithms and datasets can be downloaded from `http://www.philippe-fournier-viger.com/spmf/`.

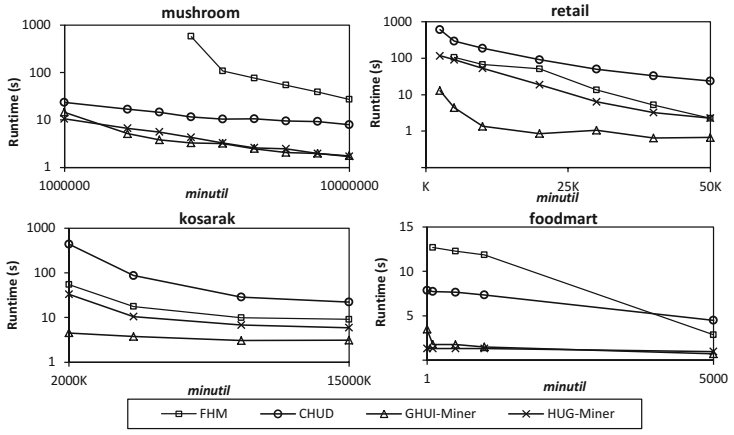

Fig. 3. Execution times

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. In fig. 3, we show the execution times of GHUI-Miner, HUG-Miner, CHUD and FHM. In Table 1, we show the number of HUIs, CHUIs, GHUIs, HUGs and LUGs for the lowest *minutil* value for each dataset.

It can first be observed that mining HUGs using HUG-Miner is much faster than mining all GHUIs by running CHUD and then GHUI-Miner. There are two reasons. First, GHUI-Miner needs to explore a larger search space to avoid pruning LUGs. In some cases, LUGs can have a very low utility, which makes them very expensive to mine. Second, mining the set of CHUIs that is needed

by GHUI-Miner to verify if generators are LUGs is expensive. In the figure, it can be clearly seen that CHUD is in general the main cost to obtain GHUIs. Thus, an interesting possibility to improve the performance of mining GHUIs is to eventually replace CHUD with a faster algorithm.

A second observation is that mining HUGs is up to 100 times faster than mining all HUIs or CHUIs. On the other hand, mining GHUIs is faster than mining all HUIs using FHM on *mushroom* and *foodmart*. The reason why the combination of CHUD and GHUI-Miner is sometimes slower than FHM is the cost of closure computation and the larger search space to avoid pruning LUGs. The reason why the combination of CHUD and GHUI-Miner is up to 46 times faster than FHM on *mushroom* is that a small proportion of itemsets are generators in that dataset. It is also a well-known fact that algorithms for mining a concise representation of patterns perform better on dense datasets [14].

A third observation is that the set of GHUIs and the set of HUGs are up to 36 times smaller than the set of HUIs. This shows that these concise representations of HUIs are very compact. Furthermore, it can be observed that the set of LUGs is often very small. Thus, if one mines only HUGs using HUG-Miner, it may obtain an interesting compromise by missing a small portion of GHUIs but having a short execution time.

Table 1. Number of patterns found

Dataset	$\|HUIs\|$	$\|CHUIs\|$	$\|GHUIs\|$	$\|HUGs\|$	$\|LUGs\|$
mushroom	3,538,181	10,311	98,315	5,979	92,336
retail	14,314	14,090	14,697	14,090	607
kosarak	56	56	67	30	37
foodmart	233,231	6,680	40,178	40,178	0

5 Conclusion

This paper proposes a new framework for mining concise representations of *high utility itemsets using generators*. We investigate the properties of generators and incorporate the concept of generator into HUI mining. We explore two new concise representations of HUIs, called High Utility Generator (HUG) and Generator of High Utility Itemsets (GHUIs). Two efficient algorithms named *HUG-Miner* and *GHUI-Miner* are proposed to respectively mine these representations. The algorithms provide different trade-offs between execution time and completeness. GHUI-Miner captures the complete set of GHUIs but spends more time because it needs to consider generators that are not HUIs. On the other hand, HUG-Miner is over 100 times faster than GHUI-Miner but misses GHUIs that are LUGs. Experimental results on both real-life and synthetic datasets show that the proposed algorithms are very efficient and achieve a massive reduction in terms of number of patterns found. Moreover, HUG-Miner is up to two orders of magnitude faster than the state-of-the-art

algorithms for CHUI mining and HUI mining. Source codes of all algorithms and datasets can be downloaded as part of the SPMF pattern mining library `http://www.philippe-fournier-viger.com/spmf/`. For future work, we will consider utility mining problems in sequential pattern mining [3]

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. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast Vertical Sequential Pattern Mining Using Co-occurrence Information. In: Proc. PAKDD 2014, pp. 40-52. (2014)
4. Fournier-Viger, P., Wu, C.-W., Zida, S., Tseng, V. S.: FHM: Faster High-Utility Itemset Mining using Estimated Utility Co-occurrence Pruning. In: Proc. 21st Intern. Symp. Methodologies Intell. Systems, Springer, pp. 83-92 (2014)
5. Gao, C., Wang, J., He, Y., Zhou, L.: Efficient mining of frequent sequence generators. In: Proc. 17th Intern. Conf. World Wide Web:, pp. 1051–1052 (2008)
6. Liu, M., Qu, J.:Mining High Utility Itemsets without Candidate Generation. In Proceedings of CIKM12, pp. 55–64 (2012)
7. Liu, Y., Liao, W., Choudhary, A.: A two-phase algorithm for fast discovery of high utility itemsets. In: Proc. PAKDD 2005, pp. 689–695 (2005)
8. Pham, T.-T., Luo, J., Hong, T.-P., Vo, B..: MSGPs: a novel algorithm for mining sequential generator patterns. In: Proc. 4th Intern. Conf. Computational Collective Intelligence, pp. 393-401 (2012)
9. Shie, B.-E., Yu, P.S., Tseng, V.S.: Efficient algorithms for mining maximal high utility itemsets from data streams with different models. Expert Syst. Appl. 39(17), pp. 12947-12960 (2012)
10. Soulet, A., Rioult, F.: Efficiently Depth-First Minimal Pattern Mining. In: Proc. 18th Pacific-Asia Conf. Knowledge Discovery and Data Mining, pp. 28–39 (2014)
11. Szathmary, L., Valtchev, P., Napoli, A., Godin., R.: Efficient vertical mining of frequent closures and generators. In: Proc. 8th Intern. Symp. Intelligent Data Analysis, August 31 - September 2, Lyon, France, pp. 393–404 (2009)
12. Szathmary, L. et al.: A fast compound algorithm for mining generators, closed patterns, and computing links between equivalence classes. In: Ann. Math. Artif. Intell. 70(1-2), pp. 81–105 (2014)
13. Tseng, V. S., Shie, B.-E., Wu, C.-W., Yu., P. S.: Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases. In: IEEE Trans. Knowl. Data Eng. 25(8), pp. 1772–1786 (2013)
14. Wu, C.-W., Fournier-Viger, P., Yu., P. S, Tseng, V. S.: Efficient Mining of a Concise and Lossless Representation of High Utility Itemsets. In: Proceedings of ICDM11, pp. 824–833 (2011)