International Journal of Innovative
Computing, Information and Control
Volume **12**, Number **4**, August **2016**

ICIC International ©2016 ISSN 1349-4198
pp. 1115–1128

MINING CONCISE REPRESENTATIONS OF HIGH UTILITY ITEMSETS WITH NEGATIVE UTILITIES FROM SOFTWARE EXECUTING TRACES

Jiadong Ren[1,2], Jiaming Yan[1,2,*], Rui Gao[1,2]
Jiandi Wang[1,2] and Haitao He[1,2]

[1]College of Information Science and Engineering
[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
Yanshan University
No. 438, West Hebei Ave., Qinhuangdao 066004, P. R. China
{ hgy; jdren }@ysu.edu.cn; *Corresponding author: jiamingy@stumail.ysu.edu.cn

Received January 2016; revised May 2016

Abstract. *Software behavior pattern mining is a worthwhile study since it can provide help for software engineers to maintain the correctness of software and detect exceptions efficiently. These high utility software behavior patterns shed light on software behavior and capture unique characteristic of software executing traces. In this paper, we incorporate the concept of generator into high utility itemsets (HUIs) mining with negative utilities and devise a concise representation of HUIs, called high utility generators (HUGs). An efficient algorithm named HUGN (high utility generator with negative utilities) is proposed to mine these representations. We firstly define a new structure to store the positive and negative utility for each item in the software executing traces. Second, we propose a structure called crucial-transaction which contributes to obtain generator patterns. And a new strategy is presented to prune unpromising patterns. Finally, an extensive experimental study with different real-life datasets shows that the proposed algorithm outperforms the state-of-the-art algorithm for the task.*
Keywords: High utility generators, Negative utilities, Software executing trace

1. **Introduction.** Computer software is playing an increasingly important role in our daily life. Software behavior learning is one of the most important tasks in many stages of software development lifecycle [1]. However, due to hard deadlines and "short-time-to-market" requirement, software products often come with many bugs and anomalies. When bugs or anomalies occur in practice, costs can be tremendous. To reduce the harm, representative patterns are important to software maintenance. It is desirable to mine representative software behavior patterns from software executing traces.

Numerous pattern mining algorithms have been applied to software failure detection. To address the issue of software reliability, J. Han et al. [2] presented a framework of frequent pattern-based classification. These frequent patterns have high quality features for classification by analyzing the relationship between pattern frequency and their discriminative power. D. Lo et al. [3] proposed the iterative pattern mining algorithm based on the semantics of several software modeling languages. They extracted frequent repetitive series of events from program executing traces as candidate software specifications. D. Lo et al. [4] put forward a method to classify software behaviors based on past history or runs. Their technique mined a set of discriminative features capturing repetitive series of events, selected the best features for classification and then trained a classifier to detect failures. In the related aspects, C. Li et al. [5] came up with an approach which

used the pattern position distribution to detect software failure instead of occurrence frequency. A method based on relational association rule mining for detecting faulty entities in software systems was put forward [6]. All proposed methods took the representative behavior patterns mining as the first step for software failure detection. Since a software executing trace can be viewed as an itemset, this question is transformed into how to mine representative patterns from itemsets.

An important limitation of pattern mining is just considering the frequency of itemsests while neglecting the importance, such as weight, unit profit or value. Hence, some algorithms were proposed to efficiently mine high utility itemsets (HUIs). Liu et al. [7] presented a significant property called transaction-weighted downward closure to maintain the downward closure property of high utility patterns. Tseng et al. [8] proposed an algorithm called UP-Growth for high utility itemsets mining (HUIM) with a series of strategies for pruning candidate itemsets. Nevertheless, a large number of candidates were produced by these algorithms. To shrink the number of candidates, some methods were presented. Tseng et al. [9] introduced an efficient algorithm called UP-Growth+ which proposed several new strategies and overestimated utilities of candidates. Liu and Qu [10] used a utility-list structure to mine HUIs without candidate generation. It avoided the costly generation and utility computation of numerous candidate itemsets. In order to improve the efficiency of HUIM, Fournier-Viger et al. [11] proposed an FHM algorithm which applied a strategy based on the analysis of item co-occurrences. EIHI [12] introduces several ideas to more efficiently maintain high-utility itemsets in dynamic databases. However, these algorithms are not designed to handle items having negative weights or unit profits, despite that such items occur in software executing traces. For example, the software executing traces can be divided into some levels. When they are too long, we intend to just record some of them which appear on the first level, so items on the other levels can be defined having negative utilities. Chu et al. [13] designed an algorithm while considering negative unit profits is HUINIV-Mine. However, mining HUIs with negative unit profits remained very costly in terms of execution time and memory. The FHN [14] extended the HUIM algorithm named FHM [11] and can handle negative unit profits efficiently. FOSHU [15] mined HUIs with negative unit profits while considering on-shelf time periods of items.

Many studies have been carried to develop efficient HUIM algorithms. However, a crucial problem of HUIM is that the set of HUIs generated by these algorithms can be very large. It is very inconvenient for a user to analyze a very large set of HUIs. To address this issue, some algorithms were proposed to mine concise representations of HUIs rather than all HUIs. GUIDE [16] integrated the concept of maximal pattern from FIM to mine maximal HUIs (HUIs having no proper supersets that are high utility). Another work was CHUD [17], which adapted the concept of closed pattern from FIM to discover closed HUIs. Wu et al. [18] proposed an efficient algorithm called CHUI-Miner for discovering CHUIs without producing candidates and avoid repeatedly scanning the original database. Although these representations are useful, generators have shown to provide several benefits over all/closed/maximal patterns in many applications [19,20]. 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). Second, generators can provide higher classification accuracy and are more useful for model selection than using all or only closed patterns. Lastly, mining generators is more efficient than discovering all patterns because it is a very small subset of all patterns. Based on the above issues, some work about mining high

utility generator patterns with negative utilities is done in this paper. The contribution of the paper is the following.

- We gain software executing traces through tracing the executing process of software dynamically. We define a structure to store the positive and negative utility for each item in the software executing traces.
- We integrate the concept of generator in HUIM to define meaningful concise representations of HUIs. We create a structure called crucial-transaction for items and the strategy of downward closure for generator patterns can prune unpromising patterns. And a new efficient HUGN algorithm is proposed to mine high utility generators.
- We evaluate the performance of HUGN algorithm on different real-life datesets.

The rest of paper is organized as follows. Section 2 gives the problem definitions. Section 3 details the crucial-transaction structure and the HUGN algorithm. Experimental results and evaluations are presented in Section 4. Section 5 concludes the work.

2. Problem Statement and Preliminaries.

A software behavior can be viewed as a series of items. In this paper, an item corresponds to the execution of a function. A set of the relevant concepts are defined as follows.

2.1. High utility itemset mining with negative utilities.

Dynamic software executing trace (DSET) is a software executing trace which is defined as $I = \{i_1, i_2, \ldots, i_m\}$, i_j $(1 \leq j \leq m)$ represents an item (function) in a software executing traces. A set of DSETs constitute a software executing trace database (STD). Each DSET T_c $(T_c \in I)$ has a unique identifier c called its T_{id}.

Definition 2.1. *The utility of function in DSET. Each item $i \in I$ is associated with a number $eu(i)$, called its external utility. For each T_c such that $i \in T_c$, a positive number $iu(i, T_c)$ is called the internal utility of i. We denote the utility of an item i in a DSET as $u(i, T_c) = eu(i) * iu(i, T_c)$. An itemset is a set of items (functions). The utility of an itemset X in a T_c is defined as $u(X, T_c) = \sum_{i \in X} u(i, T_c)$. The set of DSET containing X is denoted as $d(X)$. The utility of X in a database is defined as $u(X) = \sum_{T_c \in d(X)} u(X, T_c)$.*

For convenience, we use transaction to replace dynamic software executing trace in following paper.

Example 2.1. *Figure 1(a) shows an STD containing five transactions $(T_0, T_1, \ldots, T_4)$. The utility of the itemset c, d in T_0 is $u(\{c, d\}, T_0) = u(\{c\}, T_0) + u(\{d\}, T_0) = 1*2+2*10 = 22$. The utility of c, d is $u(\{c, d\}, T_0) + u(\{c, d\}, T_1) + u(\{c, d\}, T_2) = 22 + 46 + 24 = 92$.*

Tid	transactions
0	(a,3)(b,14)(c,1)(d,2)(e,17)
1	(b,6)(c,8)(d,3)(e,9)
2	(a,3)(c,2)(d,2)
3	(b,3)(e,1)
4	(a,2)(c,5)(e,1)

(a) a transaction database

Item	a	b	c	d	e
Utility	-5	4	2	10	-3

(b) external utility values

FIGURE 1. A transaction database and external utility values

An itemset X is a high utility itemset (HUI) 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 (LUI).

Definition 2.2. *Problem of HUI mining with negative utilities. The problem of high utility itemset mining with negative utilities is to discover all high utility itemsets in a database where external utility values may be positive or negative.*

According to these definitions, two lemmas of HUIs with respect to items having negative utility are the following.

Lemma 2.1. *HUI may contain items having negative external utilities. Items having negative external utilities may appear in an HUI.*

Example 2.2. *In our running example, $\{b, d, c, a\}$ is an HUI. It contains item a, which has an external utility of -5.*

Lemma 2.2. *HUI must contain at least an item having a positive external utility.*

Although an HUI may or may not contain items having negative external utility values, it has to contain at least an item having a positive external utility value, otherwise its utility would be negative and it would not be an HUI.

It is obvious that the utility measure is neither monotonic nor anti-monotonic. Hence, the strategies that are used in FIM to prune the search space cannot be directly applied to discovering high utility itemsets especially items with negative utilities, because these strategies are based on the anti-monotonicity of the support. To address this issue, we can overestimate the utility of itemsets using a measure called the transaction positive utility (TPU), which is anti-monotonic.

Definition 2.3. *Transaction positive utility (TPU). The transaction positive utility (TPU) of a transaction T_c is the sum of the utilities of the items in T_c having positive external utility, i.e., $TPU(T_c) = \sum_{x \in T_c \wedge eu(x) > 0} u(x, T_c)$.*

Definition 2.4. *Transaction-weighted utilization with positive external utility (TWUP). The transaction-weighted utilization with positive external utility (TWUP) of an itemset X is defined as the sum of the transaction positive utilities of transactions containing X, i.e., $TWUP(X) = \sum_{T_c \in d(X)} TPU(T_c)$.*

Example 2.3. *Figure 2(a) shows the TPU of transactions $T_0, T_1, \ldots, T_4$ for the running example. Figure 2(b) shows the TWUP of single items based on the transaction positive utility values. Let us consider itemsets $\{b, c, d\}$ and $\{b, c\}$, the values $TWUP(\{b, c, d\})$ and $TWUP(\{b, c\})$ are equal to 108, which are overestimations of $u(\{b, c, d\}) = 108$ and $u(\{b, c\}) = 58$.*

Property 2.1. *Let X be an itemset, if $TWUP(X) <$ minutil, then X and its supersets are low utility.*

We can use the TWUP to prune search space. Now, we use the running example to explain why the definition of TWUP just uses the positive external utilities. For example, item a has an external utility of -5, if we calculate the TWUP using both positive and negative external utilities, $TWUP(\{b, c\}) = TWUP(T_0) + TWUP(T_1) = 9$ and $u(\{b, c\}) = 58$, which would be a violation of Property 2.1 stating that the TWUP of an itemset is an overestimation of its utility. The consequence is that if $minutil = 10$, the itemset $\{b, c\}$ would not be output even though this itemset is an HUI (it would be pruned because $TWUP(\{b, c\}) < minutil$).

Tid	TPU
0	50
1	58
2	24
3	12
4	10

(a) TPU

Item	TWUP
a	84
b	120
c	142
d	132
e	70

(b) TWUP

Items	a	b	c	d
b	50			
c	84	108		
d	74	108	132	
e	60	120	118	108

(c) AUCS

FIGURE 2. TPU, TWUP of single items and AUCS

To mine high utility generators while considering both positive and negative utilities, we propose an algorithm named HUGN that utilizes the depth-first search procedure and ULPN structure to explore the search space of itemsets, and provides an efficient optimization named AUCS (approximated utility co-occurrence structure). ULPN are defined as follows.

Definition 2.5. *Utility-list with positive and negative utilities (ULPN). Let $\succ$ be any total order on items from I; for an itemset X, $uP(X) \subseteq X$ be the set of positive items in X, as well $uN(X) \subseteq X$ be the set of negative items in X. The ULPN of an itemset X in a database D is a set of tuples such that there is a tuple (tid, iPutil, iNutil, rPutil) for each transaction T_{id} containing X. The iPutil and iNutil elements of a tuple is respectively the utility of positive items and negative items in T_{id}, i.e., $u(uP(X), T_c)$ and $u(uN(X), T_c)$. The rPutil element of a tuple is defined as $\sum_{i \in T_{id} \wedge i \succ x \forall x \in X \wedge u(i,T_{id})>0} u(i, T_{id})$.*

Definition 2.6. *The utility of an itemset X is the sum of iPutil and iNutil values in its ULPN, i.e., $u(X) = \sum_{T_c \in T_{id}} [u(uP(X), T_c) + u(uN(X), T_c)]$.*

Example 2.4. *The order is according to ascending TWUP values and negative items succeeding positive items. So, the total order is $\{b\}$, $\{d\}$, $\{c\}$, $\{a\}$, $\{e\}$. The ULPN of $\{d\}$ is $\{(T_0, 20, 0, 2), (T_1, 30, 0, 16), (T_2, 20, 0, 4)\}$. The ULPN of $\{a\}$ is $\{(T_0, 0, -15, 0), (T_2, 0, -15, 0), (T_3, 0, -10, 0)\}$. The ULPN of $\{d, a\}$ is $\{(T_0, 20, -15, 0), (T_2, 20, -15, 0)\}$. The utility of $\{d, a\}$ is 10.*

Property 2.2. *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 iPutil and rPutil values in $ULPN(X)$ is less than minutil, then X and its extensions are low utility.*

Property 2.3. *Let X be an itemset such that $u(X) <$ minutil and only negative items can be used to extend X based on the total order $\succ$. Therefore, all transitive extensions of X with these items will be low utility and can be pruned.*

Proof: An itemset may be composed of positive items or negative items or both of them. When an itemset is composed of positive and negative items, if the sum of utilities of positive items is less than *minutil*, the sum of utilities of positive and negative items is certain less than *minutil*. When the $uP(X)$ of an itemset is less than *minutil* and negative items are used to extend it, the $uP(X)$ of the extended itemset is not changed. So, the new $uP(X)$ is also less than *minutil*, and the sum of $uP(X)$ and $uN(X)$ must be less than *minutil*. Hence, it cannot be an HUI and should be pruned. So, it can be used to prune some itemsets containing negative items.

Definition 2.7. *Approximated utility co-occurrence structure (AUCS). AUCS is a collection of tuples where* $(a, b, c) \in I^{'} * I^{'} * R$, $c = TWUP(\{a, b\})$. *AUCS stores the TWUP of all pairs of items* $\{a, b\}$ *such that* $u(\{a, b\}) \neq 0$.

Lemma 2.3. *For a given itemset* Px *(*Px *is itemset* P *combined with item* x*) and an item* y*, if there is no tuple* (x, y) *in AUCS that* $TWUP(x, y) \geq minutil$*, there is no need to extend* Px *with* y.

Proof: Itemset xy is a subsequence of Pxy. According to Property 2.1, if $TWUP(x, y)$ is less than $minutil$, then all extensions of xy must not be HUI. Thus we can conclude that $u(Pxy)$ is less than $minutil$ and we do not need to extend pattern Px with y.

2.2. **High utility generator.**

Definition 2.8. *Generator (key pattern or minimal pattern). An itemset* X *is a generator iff there is no itemset* Y *such that* $Y \subset X$ *and* $sup(X) = sup(Y)$.

The concept of generator pattern is directly related to the concept of closed pattern. An equivalence class is 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 $\{\{b, c\}, \{b, c, d, e\}\}$ of itemsets appearing in T_0 and T_1. $\{b, c\}$ is a generator and $\{b, c, d, e\}$ is the closed pattern. Our algorithm for mining generator patterns uses the following property to prune the search space. It is a property of downward closure for generator patterns.

Property 2.4. *An itemset* X *is not a generator pattern if there exists a strict subset of* X *that is not a generator.*

Definition 2.9. *High utility generator (HUG). An itemset is a high utility generator if it is a generator and the utility of it is no less than a user-specified minimum utility threshold minutil given by the user.*

The equivalence classes shown in Figure 3 for the running example explain how the concept of generator can be applied to HUIM. Each equivalence class is represented as a rectangle and is labelled with the supporting transactions and support of its itemsets. For example, the equivalence class of T_0, and T_1 contains itemsets $\{b, c\}$, $\{b, d\}$, $\{b, e\}$,

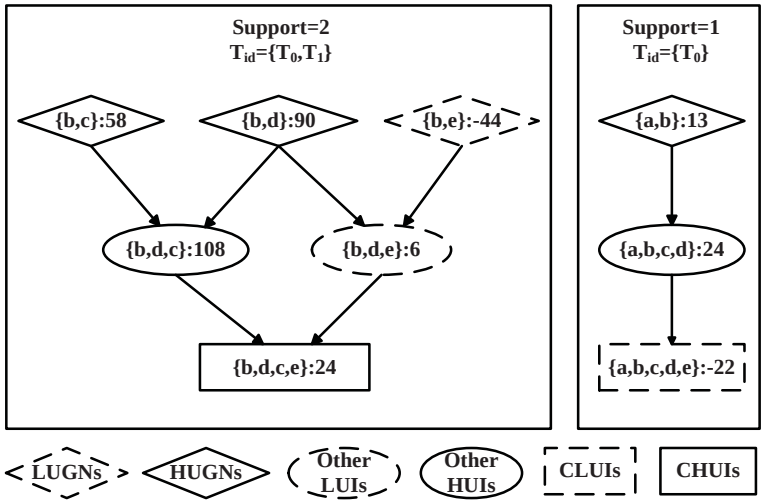

FIGURE 3. HUIs and their equivalence classes

$\{b, c, d\}$, $\{b, c, e\}$ and $\{b, c, d, e\}$ which have a support of 2. The generators of the equivalence class are $\{b, c\}$, $\{b, d\}$ and $\{b, e\}$, and the high utility generators are $\{b, c\}$ and $\{b, d\}$.

3. **Mining Concise Representations of HUIs with Negative Utilities from Software Executing Traces.** In this section, we introduce our method in detail. We propose a new structure crucial-transaction and prune strategy to estimate if an itemset is a generator. Based on this substance, the HUGN algorithm is designed to mine high utility generators from software executing traces. We construct crucial-transaction and prune strategy in Section 3.1. The algorithm HUGN is explained in Section 3.2. The main process of our method can be seen as Figure 4.

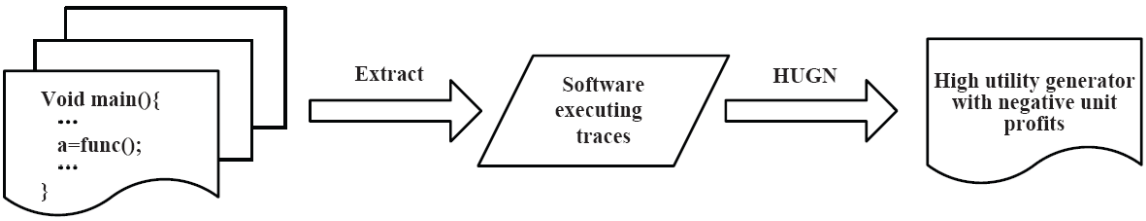

FIGURE 4. The main framework of our work

3.1. **Generator patterns mining.** Our algorithm adopts a mechanism without having to compare an itemset with its subsets to verify if an itemset is a generator. In order to interpret the mechanism in HUGN checking generator, we describe the concept of crucial-transaction firstly. It inspires from critical edges in case of minimal traversals.

Definition 3.1. *Crucial-transaction. For an itemset X, the crucial-transactions of an item e in X are denoted as $Cru(X, e)$, the set of transactions that belong to the transactions containing X without e and non to the transactions containing e: $Cru(X, e) = con(X \setminus \{e\}) \setminus con(e)$, where $con(X)$ is the set of transactions containing X.*

Let us illustrate the crucial-transaction with our running example. Consider the itemsets $\{b, c\}$ and $\{a, c\}$. $Cru(bc, b) = con(bc \setminus \{b\}) \setminus con(b) = \{0, 1, 2, 3\} \setminus \{0, 1, 4\} = \{3\} \neq \phi$, it is relative to b that bc is not contained in the transactions $\{3\}$. $Cru(ca, c) = con(ca \setminus \{c\}) \setminus con(c) = \{0, 2, 3\} \setminus \{0, 1, 2, 3\} = \phi$, and it means that the addition of c to a has no impact on the transactions containing ac.

According to Definition 3.1 we know that an itemset is generator if the transactions of it differ from that of its subset. So, it can be easily seen a necessary and sufficient condition for an itemset X to be a generator.

Property 3.1. *For an itemset X, item e in X and Suf represents all the suffixes of X. X is a generator if $\forall e \in X$ such that $X \setminus \{e\} \in Suf, Cru(X, e) \neq \phi$.*

It is a property of downward closure for generator patterns with critical transaction. It means that checking whether an itemset X is a generator pattern only requires to know the crucial-transactions of all the items in X. Unlike the usual definition, no information is required on the subsets. Therefore, we can design a depth-first algorithm if computing the crucial-transactions.

In a depth-first traversal, we intend to update the crucial-transactions of an item a for the itemset X when a new item b adds to X. In such case, we show that the crucial-transactions can efficiently be computed by intersecting the old set of crucial-transactions $Cru(X, a)$ with containing the new item b.

Property 3.2. *The following equality holds for any itemset X and any two items a, b:* $Cru(X \cup \{b\}, a) = Cru(X, a) \cap con(b)$.

Example 3.1. *In our running example, the order of items is* $\{b\}$, $\{d\}$, $\{c\}$, $\{a\}$, $\{e\}$. *Consider the join of a pair of itemsets* $\{b\} \cup \{d\}$ *and* $\{b\} \cup \{c\}$ *to generate* $\{b\} \cup \{d, c\}$. *Definition 3.1 gives* $Cru(\{b, d\}, d) = con(b) \setminus con(d) = \{4\}$. *As* $con(c) = \{0, 1, 2, 3\}$, *we obtain that* $Cru(bd, c) = Cru(\{bd\}, d) \cap con(c) = \{4\} \cap \{0, 1, 2, 3\} = \phi$. *Interestingly, Property 3.1 allows us to compute the crucial-transactions of any element included in an itemset X on a single branch. This is an ideal situation for a depth-first search algorithm.*

Thus crucial-transactions of an itemset can be calculated using crucial-transactions of the pairs of itemsets that are joined to obtain the new itemset. We should note that for an itemset $\{i\}$ containing a single item, $Cru(\{i\}, i) = con(\phi) \setminus con(i)$.

This generator checking mechanism is efficient. In HUGN, crucial-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 crucial-transactions is found to be empty).

3.2. **Main procedure.** We propose an algorithm named HUGN for mining high utility generators. It is a depth-first algorithm without generating candidate itemsets.

Algorithm 1 The HUGN algorithm

Input: D: a transaction database, minutil: a user-specified threshold.
Output: The set of generator of high-utility itemsets.

```
1. Scan D to calculate the TWUP of single items;
2. I* ← each item i such that TWUP(i) ≥ minutil;
3. Let ≻ be the total order of TWUP ascending values on I* while negative items
   succeeding all positive items;
4. Scan D to build the ULPN of each item i ∈ I* and build the AUCS structure;
5. for each {i} ∈ I* do
6.     if IsGenerator ({i}) = true∧ SUM(i.utilitylist.iPutil+i.utilitylist.iNutil) ≥
   minutil then
7.         Output ({i});
8.     end if
9. end for
10. SearchHUGN (φ, I*, minutil, AUCS);
```

Main Procedure of HUGN. The main procedure of HUGN (Algorithm 1) takes as input a transaction database D with utility values, and the *minutil* threshold. HUGN first scans D to calculate the TWUP of each item and get the set I^* (cf. Property 2.1). The TWUP values of items are used to build a total order $\succ$ on items, which is the order of ascending TWUP values and negative items succeeding positive items. In a second database scan, the ULPN of each item i is built, and AUCS is created. In our algorithm, we use hash map structure to implement the AUCS structure. Building the AUCS is very fast (it is performed with a single database scan) and it needs very little memory. Then, it executes a loop to consider each item in I^*. A call to the method IsGenerator is made to determine whether an item is a generator. If the item is a generator and its utility no less than *minutil*, it is an HUG and can be output. Then, HUGN adopts a recursive depth-first search to explore generators having more than 1-item by calling the procedure SearchHUGN.

The SearchHUGN Procedure. The SearchHUGN procedure (Algorithm 2) takes as input an itemset P, extensions of P, *minutil* and the AUCS. First SearchHUGN executes a loop over each positive itemset Px to explore its extensions. If the sum of *iPutil* and *rPutil* utility of Px are no less than *minutil*, it means that extensions of Px should be explored (cf. Property 2.2). This is performed by merging Px with all extensions Py. If $TWUP(\{x, y\})$ is no less than *minutil*, the ULPN Construct procedure is called (cf. Algorithm 3) to join the ULPNs of Pxy. Then, it estimates whether Pxy is a generator by calling the procedure of IsGenerator. If Pxy is an HUG, its supersets maybe a generator, and Pxy should be output and added to a set for storing itemsets that should be considered for further extensions (cf. Lemma 2.3). If the utility of Pxy is less than *minutil* and the items succeeding Pxy contain a positive item at least, Pxy should be considered for further extensions (cf. Property 2.3). Finally, a recursive call to the SearchHUGN procedure is done to explore extensions of its itemsets.

Algorithm 2 The SearchHUGN Procedure

Input: P: an itemset, ExtensionsOfP: a set of extensions of P, the minutil threshold, the AUCS structure.
Output: The set of high utility generator

```
1.  for each (Px ∈ ExtensionsOfP ∧ Px.utilitylist.iPutils > 0) do
2.   if SUM(Px.utilitylist.iPutils) + SUM(Px.utilitylist.rPutils) ≥ minutil then
3.    for each itemset Py ∈ ExtensionsOfP such that y ≻ x do
4.      Pxy ← Px ∪ Py;
5.      if (TWUP({x, y}) ≥ minutil according to AUCS) then
6.        Pxy.utilitylist ← ULPN Construct (P, Px, Py);
7.      end if
8.      if(IsGenerator({Pxy} ∧ |Pxy.utilitylist| ≠ |Px.utilitylist| ∧ |Pxy.utilitylist|
        ≠ |Py.utilitylist|) then
9.        if (SUM(Pxy.utilitylist.iPutil + Pxy.utilitylist.iNutil) ≥ minutil) then
10.         Output (Pxy);
11.         ExtensionsOfPx ← ExtensionsOfPx ∪ Pxy;
12.       end if
13.       else if (Pxy.utilitylis.rPutil > 0) then
14.         ExtensionsOfPx ← ExtensionsOfPx ∪ Pxy;
15.       end else
16.     end if
17.   end for
18.   SearchHUGN (Px, ExtensionsOfPx, minutil);
19.  end if
20. end for
```

The ULPN Construct Procedure. The ULPN construct procedure (Algorithm 3) takes as input an itemset P, the extension of P with an item x, the extension of P with an item y. The join operation for two itemsets $P \cup \{x\}$ and $P \cup \{y\}$ such that $x \succ y$ is performed as follows. For each set of tuples ex, ey and e, such that they have the same *tid*, the ULPN of $P \cup \{x, y\}$ is obtained. If P is null, for each pairs of tuples ex and ey, such that they have the same *tid*, the ULPN of $\{x, y\}$ is obtained.

Algorithm 3 The ULPN 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 ULPN of Pxy.

```
1.  UtilityListOfPxy ← φ;
2.   for each tuple ex ∈ Px.utilitylist do
3.    if ∃ ey ∈ Py.utilitylist ∧ ex.tid = exy.tid then
4.     if P.utilitylist ≠ φ then
5.       Search element e ∈ P.utilitylist such that e.tid = ex.tid;
6.       exy ← (ex.tid, ex.iPutil + ey.iPutil − e.iPutil, ex.iNutil + ey.iNutil −
         e.iNutil, ey.rPutil);
7.     end if
8.     else
9.       exy ← (ex.tid, ex.iPutil + ey.iPutil, ex.iNutil + ey.iNutil, ey.rPutil);
10.    end else
11.    UtilityListOfPxy ← UtilityListOfPxy ∪ exy;
12.   end if
13.  end for
14. return UtilityListPxy;
```

Algorithm 4 The IsGenerator Procedure

Input: X: an itemset, tail: the set of the remaining items to be used in order to generate the candidates.
Output: The generator patterns.

```
1.  if ∀ e ∈ X, Cru (X, e) ≠ φ then
2.   Output (X);
3.   for each e ∈ tail do
4.     tail = tail \ {e};
5.     Y = X ∪ {e};
6.     con(Y) = con(X) ∩ con(e);
7.     Cru (Y, e) = Cru (X) ∩ con(e);
8.     for each e ∈ X do
9.       Cru (Y, e) = Cru (X, e) ∩ con(e);
10.    end for
11.    IsGenerator (Y, tail);
12.  end for
13. end if
```

The IsGenerator Procedure. The algorithm IsGenerator takes as input an itemset X, the set of the remaining items to be used in order to generate the candidates. First IsGenerator checks whether X is a generator. If the crucial-transaction of X is not null, it is a generator and should be output (cf. Property 3.1). Then itemsets containing X based on the remaining items are explored. For each item e where $X \cup \{e\}$ is an itemset of D, $X \cup \{e\}$ to itemset Y is constructed, and according to Property 3.2 the *con* and Cru are updated. Finally, the function IsGenerator is recursively called with the updated remaining items.

4. **Performance Evaluation.** To evaluate HUGN, we compare the performance with HUINIV-Miner and FHN. We make some improvements on HUINIV-Miner such that

it can mine HUGs. Experiments are performed on a computer with a 64 bit Core i5 processor running Windows 7 and 4GB of free RAM. Algorithms were implemented in Java. During all experiments, memory measurements are done with the standard Java memory API.

Experiments are carried on five datasets. We choose a real software dataset cflow (for static analysis of C language code) to test the HUGN algorithm. We get the experiment data of cflow with the help of pvtrace, Gephi and Graphviz on Linux. It contains 101,174 transactions with 130 distinct items and average transaction length of 11 items. The other four datasets are chosen because they are real-life datasets having varied characteristics and represent four kinds of data. The first dataset is chess. It contains 3,196 transactions with 75 distinct items and an average transaction length of 35 items. The second dataset is mushroom. It is a dense dataset with 120 distinct items, 8,124 transactions, and an average transaction length of 23 items. The third dataset is pumsb. It contains 49,046 transactions with 7,116 distinct items and an average transaction length of 74 items. The forth dataset is retail. It contains 88,162 transactions having an average length of 8.09 items, and 16,470 distinct items.

For all datasets, external utilities for items are generated between $-1{,}000$ and 1,000 by using a log-normal distribution and quantities of items are generated randomly between 1 and 5, similarly to the settings of [9-11].

For each dataset, we run HUGN and HUINIV-Miner algorithms, while decreasing the *minutil* threshold until the algorithms became too long to execute, run out of memory or a clear winner is observed. We record the different running time on each dataset for different algorithms. The comparison of execution time is shown in Figure 5, Figure 6, Figure 7, Figure 8 and Figure 9. We can see that the running time of two algorithms is reducing constantly by increasing the minimum utility threshold. For all the five datasets, HUGN was respectively up to 15 times, 10 times, 10 times, 20 times and 40 times faster than HUINIV-Miner.

In terms of memory usage, HUGN uses much less memory than HUINIV-Miner. On cflow, chess, mushroom and pumsb datesets, HUINIV-Miner runs out of memory under our 4GB memory limit while HUGN is respectively using 566MB, 42MB, 70MB, 678MB for the lowest *minutil* values. Lastly, for the retail dataset, the memory usage of HUGN was about two times less than HUINIV. Overall, HUGN uses pretty less memory than HUINIV-Miner.

There are several reasons why HUGN performs better than HUINIV-Miner. The first reason is that HUINIV-Miner strictly relies on the TWU [12] model for pruning the search space. However, the TWU model provides a less strict upper bound on the utility of

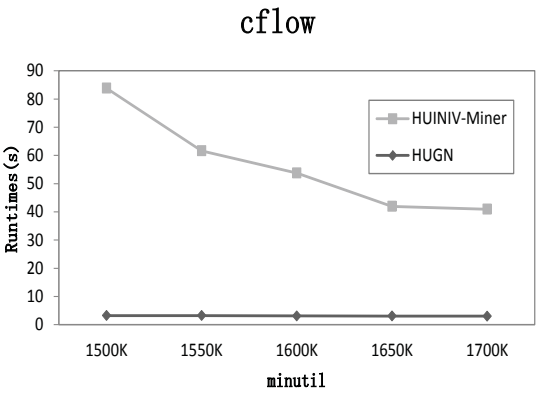

FIGURE 5. Running time on cflow

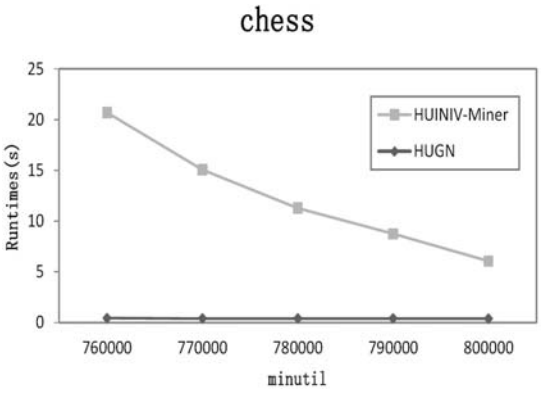

FIGURE 6. Running time on chess

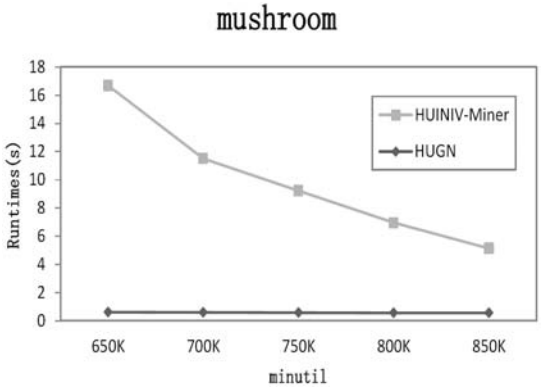

FIGURE 7. Running time on mushroom

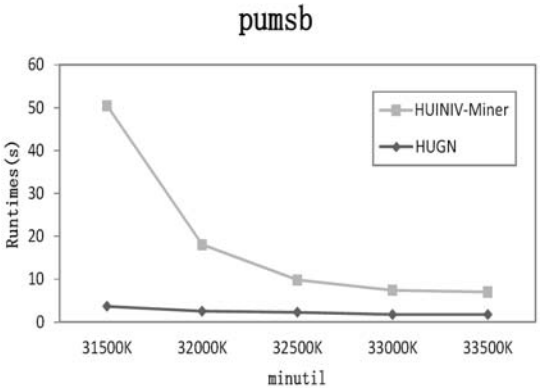

FIGURE 8. Running time on pumsb

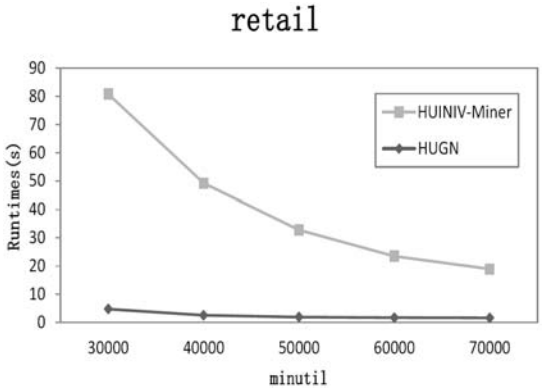

FIGURE 9. Running time on retail

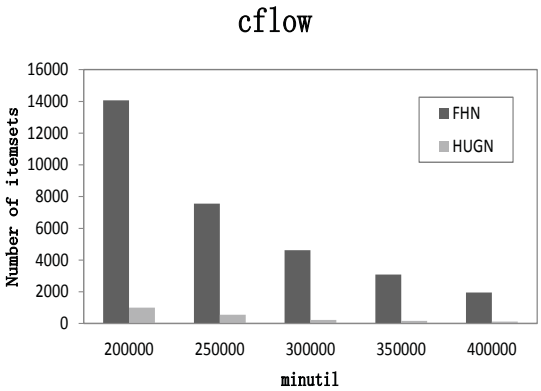

FIGURE 10. Number of itemsets on cflow

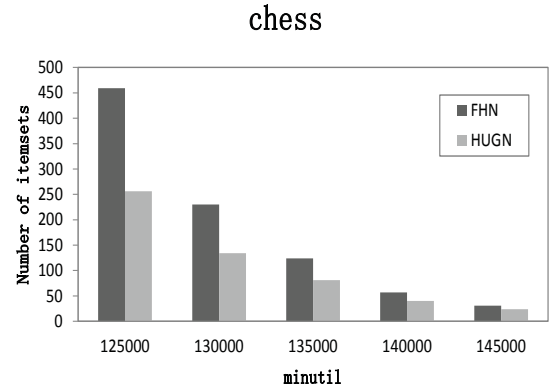

FIGURE 11. Number of itemsets on chess

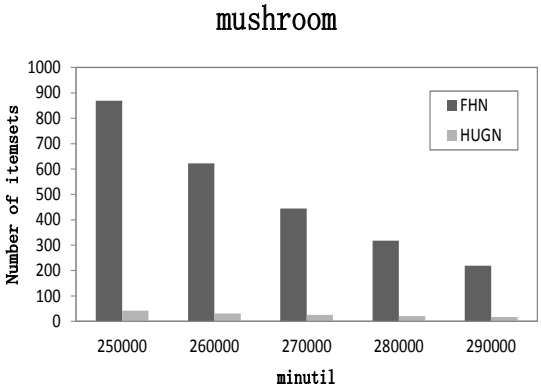

FIGURE 12. Number of itemsets on mushroom

itemsets than utility-lists [12]. HUGN uses both ULPN and TWUP of itemsets containing two items (the AUCS) to prune the search space. Thus, it can prune a larger part of the search space. Second, HUINIV-Miner is a level-wise algorithm that needs to maintain a large amount of itemsets in memory to find larger patterns. Furthermore, since it is using a two-phase approach it suffers from the problem of generating and maintaining a huge amount of candidates in memory before low-utility itemsets can be pruned. In HUGN, these problems are avoided by using a depth-first search that mines high utility generators without generating candidates. This is what allows HUGN to consume much less memory than HUINIV.

On the other hand, we run HUGN and FHN on datasets cflow, chess and mushroom and record the number of result itemsets. These results are shown in Figure 10, Figure 11 and Figure 12. It is observed that the number of candidate patterns is decreasing with the increasing minimum utility threshold. With the minimum utility threshold increasing, more unpromising patterns are pruned. We can see from the result that the number of itemsets produced by HUGN is less than FHN algorithm.

What is more, an interesting observation is that HUGN performs very well on dense datasets such as cflow and mushroom compared to FHN. In these datasets, there are lots of itemsets belonging to the same equivalence class, and a large number of itemsets could be discarded by HUGN. It shows that these concise representations of HUIs are very compact.

5. **Conclusions.** In this paper, we mine high utility generator with negative utilities from the software executing traces. Considering that the utility of each item can be positive or negative, we use ULPN to store positive and negative utility respectively. Pruning strategy and extending strategy are designed to avoid the costly utility computation. In practice, the number of HUIs obtained from software executing traces is very large. With the purpose of shrinking the result of HUIs, we design a novel algorithm called HUGN to mine HUGs which have no proper subsets that are HUIs and have the same support. We start from single items and explore the search space of itemsets recursively by appending single item. An extensive experimental study on various databases shows that HUGN has better efficiency.

In the future, we want to make this algorithm handle the large real software executing traces. We assume the databases are static in this paper, so we will adjust the algorithm to maintain concise representations of high-utility itemsets in dynamic databases.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61572420, No. 61472341 and the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2013203324, No. F2014203152 and No. F2015203326.

REFERENCES

[1] J. F. Bowring, J. M. Rehg and M. J. Harrold, Active learning for automatic classification of software behavior, *ISSTA*, vol.29, no.4, pp.195-205, 2004.

[2] H. Cheng, X. Yan, J. Han et al., Discriminative frequent pattern analysis for effective classification, *ICDE*, pp.716-725, 2007.

[3] D. Lo, S.-C. Khoo and C. Liu, Efficient mining of iterative patterns for software specification discovery, *KDD*, pp.460-469, 2007.

[4] D. Lo, H. Cheng, J. Han et al., Classification of software behaviors for failure detection: A discriminative pattern mining approach, *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.557-566, 2009.

[5] C. Li, Z. Chen, H. Du et al., Using pattern position distribution for software failure detection, *International Journal of Computational Intelligence Systems*, vol.6, no.2, pp.234-243, 2013.

[6] G. Czibula, Z. Marian and I. G. Czibula, Detecting software design defects using relational association rule mining, *Knowledge and Information Systems*, vol.42, no.3, pp.545-577, 2015.

[7] Y. Liu, W. Liao and A. Choudhary, A two-phase algorithm for fast discovery of high utility itemsets, *PAKDD*, vol.3518, pp.689-695, 2005.

[8] V. S. Tseng, C. W. Wu, B. E. Shie et al., UP-Growth: An efficient algorithm for high utility itemset mining, *Proc. of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.253-262, 2010.

[9] V. S. Tseng, B. E. Shie, C. W. Wu et al., Efficient algorithms for mining high utility itemsets from transactional databases, *IEEE Trans. Knowledge and Data Engineering*, vol.2, no.8, pp.1772-1786, 2013.

[10] M. Liu and J. Qu, Mining high utility itemsets without candidate generation, *Proc. of the 21st ACM International Conference on Information and Knowledge Management*, pp.55-64, 2012.
[11] P. Fournier-Viger, C. W. Wu, S. Zida et al., FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning, *Foundations of Intelligent Systems*, pp.83-92, 2014.
[12] P. Fournier-Viger and J. C.-W. Lin, Efficient incremental high utility itemset mining, *Ase International Conference on Big Data*, 2015.
[13] C.-J. Chu, V. S. Tseng and T. Liang, An efficient algorithm for mining high utility itemsets with negative item values in large databases, *Applied Math. Comput.*, vol.215, pp.767-778, 2009.
[14] P. Fournier-Viger, FHN: Efficient mining of high-utility itemsets with negative unit profits, *Advanced Data Mining and Applications*, vol.215, pp.16-29, 2014.
[15] P. Fournier-Viger and S. Zida, FOSHU: Faster on-shelf high utility itemset mining-with or without negative unit profit, *ACM*, 2015.
[16] B.-E. Shie, P. S. Yu and V. S. Tseng, Efficient algorithms for mining maximal high utility itemsets from data streams with different models, *Expert Syst. Appl.*, vol.39, no.17, pp.12947-12960, 2012.
[17] C.-W. Wu, P. Fournier-Viger et al., Efficient mining of a concise and lossless representation of high utility itemsets, *Proc. of ICDM11*, pp.824-833, 2011.
[18] C.-W. Wu, P. Fournier-Viger, J.-Y. Gu et al., Mining closed+ high utility itemsets without candidate generation, *Technologies and Applications of Artificial Intelligence*, 2015.
[19] T.-T. Pham, J. Luo, T.-P. Hong and B. Vo, MSGPs: A novel algorithm for mining sequential generator patterns, *Proc. of the 4th Intern. Conf. Computational Collective Intelligence*, pp.393-401, 2012.
[20] C. Gao, J. Wang, Y. He et al., Efficient mining of frequent sequence generators, *Proc. of the 17th Intern. Conf. World Wide Web*, pp.1051-1052, 2008.