International Journal of Innovative Computing, Information and Control
Volume **12**, Number **3**, June **2016**

ICIC International ©2016 ISSN 1349-4198
pp. **959–971**

AN EFFICIENT ALGORITHM FOR MINING HIGH UTILITY CONTIGUOUS PATTERNS FROM SOFTWARE EXECUTING TRACES

Guoyan Huang[1,2], Rui Gao[1,2,*], Jiandi Wang[1,2], Jiaming Yan[1,2] and Jiadong Ren[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: krystal_gaor@163.com

Received December 2015; revised March 2016

Abstract. *Software behavior pattern mining has important significance since it can provide help for software engineers to maintain the correctness of software and detect exceptions as soon as possible. These high utility software behavior patterns shed light on software behavior and capture unique characteristic of software traces. In this paper, we propose a novel approach HUCP-Miner (high utility contiguous pattern mining) to mine high utility contiguous patterns from the software executing traces. First of all, this work presents a maximum utility measure which is used to simplify the utility calculation for contiguous patterns. Second, we propose a novel structure called UL-list (utility and location list) to store utility and location information of patterns which contributes to backward extension. Based on UL-list, a remaining utility upper bound model (ruub) and extension strategy are put forward to prune the unpromising patterns early. Finally, an extensive experimental study with different real-life datasets shows that the proposed algorithm has impressive performance.*

Keywords: High utility contiguous patterns, Maximum utility measure, Software executing trace

1. Introduction.

With the development of information age, software is becoming more and more important in our life. Most of our activities depend on the correct working of software systems. Bugs and anomalies in software can cause huge financial losses each year in addition to privacy and security threats. Software behavior learning is one of the most important tasks in many stages of software development lifecycle [1]. The pattern-based software failure detection approach is a method in this area which helps software engineers to find the failures quickly and accurately.

A large number of pattern mining algorithms have been applied to software failure detection. Lo et al. [2] presented a novel method to classify software behaviors based on past history or runs. They mined a set of discriminative features to capture repetitive series of events from program execution traces. It was possible to generalize past known errors and mistakes to capture failures and anomalies. 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. On a related front, Li et al. [4] used the pattern position distribution as features to detect software failure occurring through misused software patterns. A method based on relational association rule mining for detecting faulty entities in software systems was put forward [5]. It was a common phenomenon that these methods took the representative behavior patterns mining as the first step for software failure detection.

Since a software executing trace can be viewed as a sequence with a specific order [6], this question transforms into how to mine representative patterns from sequences.

An important limitation of frequent pattern mining is that it assumes that all items have the same importance (weight, unit profit or value). So utility was introduced into frequent pattern mining to mine for high utility patterns. Liu et al. [7] presented a significant property (transaction-weighted downward closure) to maintain the downward closure property of high utility patterns. Tseng et al. [8] employed a UP-Tree structure in an efficient UP-Growth algorithm. However, these algorithms generated a very large number of candidate itemsets. To solve these problems, some methods were proposed. HUI-Miner [9] used a utility-list structure to mine high utility itemsets without candidate generation. It avoided the costly generation and utility computation of numerous candidate itemsets. FHM [10] applied a novel strategy based on the analysis of item co-occurrences. Compared to HUI-Miner, FHM was more efficient since it reduced the number of join operations that needed to be performed. FOSHU [11] mined HUIs while considering on-shelf time periods of items.

In fact, the software executing trace is a list of events with an especial order which reflects the function calling relationship. High utility itemset mining was unsuitable, so high utility sequential pattern mining was born at the right moment. According to Ahmed et al.'s definitions [12], in addition, they designed three utility calculation ways to identify the utility of a pattern in a sequence under corresponding conditions of the three ways. However, the utilities of patterns are difficult to be calculated since a pattern in a sequence has to firstly determine which condition the pattern belongs to. To avoid such case and simplify the utility calculation of subsequences in sequences, Lan et al. [13] previously proposed a simple utility measure which was called maximum utility measure to achieve this goal. Since the main purpose of sequential pattern mining is to find most traces' software behavior, rather than general software behavior, the proposed maximum utility function may be more appropriately used to find high utility sequential patterns from software executing traces. Yin et al. [14] also presented similar maximum utility concept to handle the problem of mining sequential patterns with high-utility, and proposed a tree-based mining approach named USpan, which adopted the Lexicographic Qsequence Tree (LQS-Tree) to keep the utility information of items in quantitative sequences. The LQS-Tree structure needs to keep the utilities of all items in sequences for determining the maximum value among utilities of a pattern in a sequence. To get tighter upper bound, PHUI [15] algorithm was proposed. It used an improved projection-based pruning strategy for mining high utility sequential patterns and applied the maximum measure to calculating the utilities. Due to the tighter upper bound, many unpromising patterns were pruned. Most algorithms focus on mining high utility sequential patterns from sequence databases. So it is necessary to mine high utility patterns from complex event sequences. UBER-Mine [16] applied UR-Tree to mining complete set of high utility episode rules in complex event sequences.

These items in software executing trace were not only ordered but also consecutive. The general sequential pattern mining algorithms are not applicable. The contiguous pattern has a characteristic that the items appearing in this pattern must be adjacent with respect to the underlying order as defined in the pattern. Zhou et al. [17] established a Two-Phase utility mining method to discover high utility path traversal patterns from weblog databases. Ahmed et al. [18] designed an algorithm to mine utility-based web path using a pattern growth sequential mining approach. However, long patterns may result in very high utility value. And then, an efficient algorithm [19] to discover effective web traversal patterns was proposed based on average utility model. However, these existing algorithms cannot handle the software executing traces directly. Based on the above issues, some works about high utility contiguous pattern mining are done in the paper. The contribution of the paper is the following.

- The maximum utility measure is proposed to simplify the utility evaluation for subsequences in a set of software executing sequences.
- We create a utility-location list (UL-list) structure to store the patterns' location and utility information. The tighter upper-bound utility and extension strategy can prune more unpromising patterns as early as possible. With the above structure and strategy, a new efficient algorithm (HUCP-Miner) is proposed to mine high utility contiguous patterns.
- We evaluate the performance of HUCP-Miner algorithm through a set of experiments.

The rest of paper is organized as follows. Section 2 reviews the problem definitions. Section 3 details the UL-list structure, strategy and the HUCP-Miner algorithm. Experimental results and evaluation are presented in Section 4. Section 5 concludes the work.

2. Definitions.

A software behavior can be viewed as a series of events with special order. In this paper, an event corresponds to the execution of a function. A set of the relevant concepts is defined as follows.

Definition 2.1. *Dynamic software executing sequence (DSES). DSES is a software executing trace. It is defined as* $S =< S_{start}, S_{start+1}, \ldots, S_{end} >$, S_i *(start* $\le i \le$ *end) represents an event in a software executing sequence.*

Definition 2.2. *Software executing sequence database (SSD). A set of DSESs constitutes an SSD.*

We focus on the calling relationships between these events (functions). So we intend to mine these contiguous patterns from SSD.

Definition 2.3. *Contiguous pattern (CP). A contiguous pattern* $C =< e_1, e_2, \ldots, e_n >$ *is considered as a subsequence of DSES* $S =< s_1, s_2, \ldots, s_m >$*, and it satisfies that these events appearing in C must be adjacent in sequence S. This sequence S has integers* $1 \le i_1 \le i_2 \le \ldots \le i_n \le m$ *where* $e_1 = s_{i1}$, $e_2 = s_{i2}$, $\ldots$, $e_n = s_{in}$.

For convenience, we use pattern to replace contiguous pattern in following paper.

Definition 2.4. *r-pattern. If* $|CP| = r$*, this contiguous pattern is called an r-pattern.*

Definition 2.5. *S/X and S/X+. Given a CP X and a DSEC S with* $X \subseteq S$*, the set of all events after X in S is denoted as S/X; and S/X+ means the set of all events after X in S, containing X.*

Definition 2.6. *The local utility of function in DSEC S. We denote the local utility of an event* i_j *in a DSEC as* $lu(i_j, S) = eu(i) * iu(i_j, S)$.

The local remaining utility of function in DSEC S. It is denoted as $lru(i_j, S)$*, is the summation of the utilities of all events in* S/i_j+*, where* $lru(i_j, S) = \sum_{e \in S/i_j+} lu(e, S)$.

Remark 2.1. *In a DSES, every event has a different utility which represents the importance of the software behavior. It should be noted that an event appears many times in a DSES. The symbol* i_j *that arises in Definition 2.6 is the j-th same event i in a DSEC.* $eu(i_j)$ *which is defined in 2.6 represents the external utility of an event* i_j*. In Figure 1(b), it shows the external utility of each event.* $iu(i_j, S)$ *is the internal utility of an event* i_j *in sequence S which means the occurrence count of this event in a DSES.*

Definition 2.7. *The local utility of pattern in DSES S. The local utility of a contiguous pattern* C_j *in a DSES is the summation of the events' local utility which appears in* C_j*.*

sid	software executing sequence	tsu
S1	<A(3), B(2), C(1), A(4), B(1)>	48
S2	<A(4), C(1), A(5), D(4)>	57
S3	<B(1), D(2), E(1), B(2), C(1)>	18

(a) three sequences with internal utility

event	A	B	C	D	E
utility	5	3	4	2	1

(b) external utility of 1-patterns

event	A	B	C	D	E
twu	105	66	123	75	18

(c) the upper-bound utility of 1-patterns

Figure 1. Database

Definition 2.8. *The utility of pattern in DSES S. The maximum valve among the utility values of pattern c in a sequence is regarded as the utility of pattern c. We denote it as* $u(c, S) = \max(lu(c_1, S), \ldots, lu(c_j, S))$.

The remaining utility of pattern in DSES S. It is the maximum local remaining utility of pattern c where S/c is not null. We denote it as $ru(c, S) = \max(lru(c_1, S), \ldots, lru(c_j, S))$ *where* $S/c_j \neq \phi$.

Example 2.1. *In Figure 1(a), there are two patterns* $< AB >$ *in the sequence S1, and the two local utilities of* $< AB >$ *are calculated as 21 (*$= 3*5+2*3$*) and 23 (*$= 4*5+1*3$*), respectively. The value 23 (*$= \max(21, 23)$*) is regarded as the utility of the 2-pattern* $< AB >$ *in S1 by the maximum utility measure. The local remaining utility of* $< AB >$ *in S1,* $lru(< AB >, S1)$*, are 48 and 23, respectively. However, the second 2-pattern* $< AB >$ *is in the end of S1, so it cannot be calculated.* $ru(< AB >, S1) = 48$.

Definition 2.9. *Actual utility. The actual utility of a pattern X in SSD is the summation of the utility in all the sequences where X appears. It is denoted as* $au(X) = \sum_{X \subseteq Si \wedge Si \in SSD} u(X, Si)$.

Remaining utility. The remaining utility of a pattern X in SSD is the summation of the remaining utility in all the sequences where X appears and is not in the end. It is denoted as $ru(X) = \sum_{X \subseteq Si \wedge Si \in SSD} ru(X, Si)$.

Definition 2.10. *Total sequence utility. The total sequence utility is the summation of the local utility of events which appear in a DSES S. It is denoted as* $tsu(S) = \sum_{x \subseteq S} lu(x, S)$.

In Figure 1(a), the last column is the total utility of each sequence.

Definition 2.11. *High utility contiguous pattern (HUCP). Let* λ *be a pre-defined minimum utility threshold. A CP X is called a HUCP if au(X) is not less than* λ.

However, the downward-closure property in traditional sequential pattern mining is not applicable for utility sequential pattern mining. Suppose a pattern is a high utility pattern, but its super-pattern may be not. To keep the downward-closure property, we use the upper-bound of utility of any subsequence in that sequence. These definitions related to the upper-bound model are described below.

Definition 2.12. *The utility upper-bound (twu). The summation of the sequence utility of all the sequences containing P in SSD is the utility upper-bound of a pattern P.* $twu = \sum_{P \subseteq Si \wedge Si \in SSD} tsu(Si)$.

Example 2.2. *The twus of distant 1-patterns are shown in Figure 1(c). For example, consider the 1-pattern* $< B >$*,* $twu(B) = tsu(S1) + tsu(S3) = 48 + 18 = 66$.

Lemma 2.1. *The twu of a CP P is an upper bound utility of P,* $twu(P) \geq au(P)$*. If the twu(P) is less than a pre-defined minimun utility threshold* λ*, for* $\forall Q$*, where* $P \subseteq Q$*, Q cannot be HUCP.*

Proof: Suppose Sp is the set of sequences containing P in SSD and Sq is the set of sequences containing Q in SSD.

$$\begin{aligned}
\because P \subseteq Q &\Rightarrow Sq \subseteq Sp \\
\therefore au(Q) &= \sum_{Si \in Sq} u(Q, Si) \\
&\leq \sum_{Si \in Sq} tsu(Si) \\
&\leq \sum_{Si \in Sp} tsu(Si) \\
&= twu(P) < \lambda
\end{aligned}$$

Definition 2.13. *The remaining utility upper-bound (ruub). The CP's remaining utility upper-bound is its remaining utility in SSD.*

In this paper, we aim to mine these contiguous patterns with the backward extension method. The calculation method of ruub is the same with remaining utility ru(X) which is defined in 2.9. The value of ruub is smaller than the value of twu. Hence, the remaining upper bound model reduces the number of candidates.

Lemma 2.2. *The ruub of a CP P is a remaining upper bound utility with P as prefix, $twu(P) \geq ruub(P) \geq au(P)$. If the ruub(P) is less than a pre-defined minimum utility threshold λ, for $\forall Q$, where Q is the backward extension of P, Q cannot be HUCP.*

Proof: Let an event X be a high remaining utility upper-bound pattern and Sx be the set of sequences containing X in SSD. Suppose Y is a backward extension of X, so $X \subset Y$. Sy is the set of sequences containing Y in SSD. Then Y cannot be presented in any sequence where X is absent, so $Sy \subseteq Sx$. In addition, $X \subset Y \subseteq S$ and $S \in SSD$.

$$\begin{aligned}
\because X \subset Y \subseteq S \text{ and } S \in SSD &\Rightarrow Sy \subseteq Sx \\
\therefore au(Y) &= \sum_{Si \in Sy} u(Y, Si) \\
&= \sum_{Si \in Sy} u(X, Si) + u(Y/X, Si) \\
&\leq \sum_{Si \in Sy} u(X, Si) + u(S/X, Si) \\
&\leq \sum_{Si \in Sx} u(X, Si) + u(S/X, Si) \\
&= ru(X) = ruub(X) < \lambda
\end{aligned}$$

There the remaining utility upper-bound $ruub$ value of X is $ruub(X)$. If $ruub(X)$ is less than a pre-defined minimum utility threshold λ, so $au(Y)$ is also less than λ, then Y must not be HUCP. Here Lemma 2.2 is given to prove that $ruub$ has the downward-closure property and it can be used for pruning unpromising patterns.

3. Mining High Utility Contiguous Patterns from Software Executing Traces.

In this section, we introduce our method in detail. We propose a new structure UL-list to store position, next adjacent 1-pattern and utility information. Based on this structure, the HUCP-Miner algorithm is designed to mine high utility contiguous patterns from software executing traces. We construct UL-list structure in Section 3.1 and describe the pruning strategy and extension strategy based on UL-list in Section 3.2. The algorithm

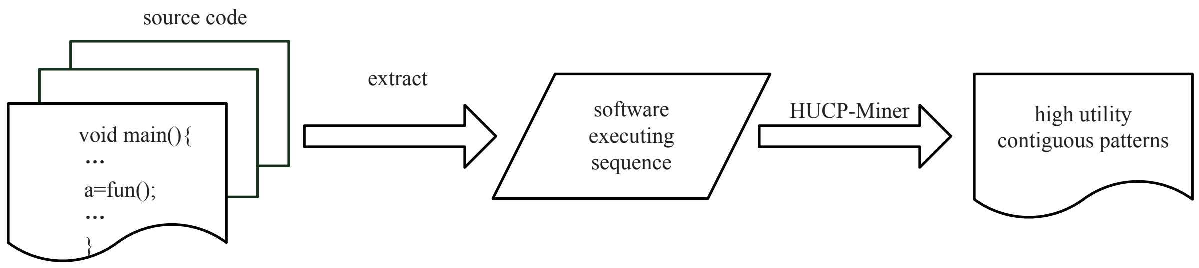

FIGURE 2. The main framework of our work

HUCP-Miner is explained in Section 3.3. The main process of our method can be seen as Figure 2.

3.1. **Utility-location list (UL-list) structure.** First, we require a structure for storing the whole information efficiently in terms of storage space. Second, the structure should be efficiently updatable when backward extend. We address all these challenges by improving the utility-list structure which is used for mining high utility itemset. What is more, we can get the value of actual utility and remaining utility upper-bound ($ruub$) from UL-list. Each element in the UL-list of P contains five fields: sid, iutil, rutil, index, nitem.

- sequence sid indicates a sequence containing P
- the location utility of P in S, i.e., $lu(P, S)$
- the location remaining utility of P in S, i.e., $lru(P, S)$
- the location index of the last item of P
- the next-adjacent 1-pattern of the last item of P

Considering that a contiguous pattern $< x >$ has a UL-list structure, $< y >$ is a remaining upper-bound 1-pattern and belongs to the next-adjacent 1-patterns of $< x >$. Algorithm 1 introduces the process of constructing the UL-list of a new contiguous pattern $< xy >$. For each element in x.UL-list, it chooses the corresponding element in y.UL-list. That is, two elements should have same tid and the index of Ey should be equal to that of Ex plus 1 (line 2). The utility and location information of Exy is calculated according to the line 3. We construct these 1-patterns' UL-lists firstly, and then obtain UL-lists of all patterns by iterative growing without scanning the database.

Algorithm 1 : Construct

```
Inputs: the utility-location list of $x$: x.UL-list, the utility-location list of $y$: y.UL-list
Output: the utility-location list of $xy$: xy.UL-list
1.for each $Ex \in$ x.UL-list do
2.  if $Ey \in$ y.UL-list, Ex.tid = Ey.tid and Ex.index = Ey.index-1 then
3.      $Exy$ =<Ex.tid, Ex.iutil+Ey.iutil, Ey.rutil, Ey.index, Ey.nitem>;
4.  end if
5.   append $Exy$ to Exy.UL-list;
6.end for
7.return Exy.UL-list;
```

Example 3.1. *As is shown in Figure 1(c), the 1-patterns set $I = \{A, B, C, D, E\}$. Suppose the minimum utility threshold λ is 45, after the first scan, the 1-pattern $< E >$ can be pruned, because $twu(E) = 18 < \lambda$ (by Lemma 2.1). The UL-list structures of all upper-bound 1-patterns and partial contiguous 2-patterns are shown in Figure 3. Considering*

sid	iutil	rutil	index	nitem
1	15	48	1	B
1	20	23	4	B
2	20	57	1	C
2	25	33	4	D

(a)UL-list of <A>

sid	iutil	rutil	index	nitem
1	6	33	2	C
1	3	3	5	Φ
3	3	18	1	D
3	6	10	4	C

(b)UL-list of <B>

sid	iutil	rutil	index	nitem
1	4	27	3	A
2	4	37	2	A
3	4	4	5	Φ

(c)UL-list of <C>

sid	iutil	rutil	index	nitem
2	8	8	5	Φ
3	4	15	2	E

(d)UL-list of <D>

sid	iutil	rutil	index	nitem
1	21	48	2	C
1	23	23	5	Φ

(e)UL-list of <AB>

sid	iutil	rutil	index	nitem
2	24	57	2	A

(f)UL-list of <AC>

FIGURE 3. The UL-list structures of patterns

the 1-pattern $< A >$, it appears two times in $S1$, so the two elements $< 1, 15, 48, 1, B >$ and $< 1, 20, 23, 4, B >$ can exist in the UL-list of $< A >$. Algorithm 1 can construct these UL-lists of 2-patterns without scanning the database. For example, the UL-list of $< AB >$ has two elements $< 1, 21, 48, 2, C >$ and $< 1, 23, 23, 5, \phi >$ by constructing the UL-list of $< A >$ and that of $< B >$. The rest of patterns can be constructed with the same process as above.

3.2. **Pruning strategy and extension strategy.** We design the process of backward extending contiguous k-patterns from contiguous $(k-1)$-patterns. This process will produce a large number of candidates. The pruning strategy and extension strategy are also proposed to ensure the efficiency of the backward extension.

Pruning Strategy. To reduce the search space, we can use the iutils and rutils in the UL-list of a contiguous pattern. Given the UL-list of a pattern P, the sum of all the maximum rutils in UL-list of P is its actual utility according to Definition 2.8 and Definition 2.9. If the result value is not less than pre-defined minimum utility threshold, this pattern P is a high utility contiguous pattern (by Definition 2.11).

In addition, the remaining utility upper-bound of P ($ruub(P)$) can be obtained by its UL-list structure (Definition 2.13). This value decides whether this pattern should be pruned or not. If the $ruub(P)$ is not less than minimum utility threshold, the P can be backward extending. Otherwise, any backward extensions of P will not be high utility contiguous patterns, so this pattern is no longer considered in the subsequent mining process and can be pruned (by Lemma 2.2).

Extension Strategy. Based on the continuity of contiguous patterns, when a contiguous pattern is backward extended, only its next-adjacent 1-patterns can be considered. In the UL-list of contiguous pattern, we record its next-adjacent 1-pattern as nitem. The contiguous pattern can be backward extended with 1-patterns which appears in the nitem of its UL-list. This manner decreases the number of candidates generated during the extension process. And if these next-adjacent 1-patterns are not remaining utility upper-bound patterns, this contiguous pattern stops backward extending (by Lemma 2.2). The extension strategy is used before constructing the UL-list of a new pattern. By applying this strategy, we can not only get the contiguous patterns but also avoid producing some unpromising contiguous patterns.

Example 3.2. *Figure 4 describes the whole process of backward extension. Suppose minimum utility threshold* λ *is 45. By scanning the UL-list of* $< B >$*, we can obtain that* $au(B) = 12 < \lambda$ *and* $ruub(B) = 51 > \lambda$*, so* $< B >$ *is not a high utility contiguous pattern but a remaining utility upper-bound pattern. The* $< D >$ *cannot be extended due to restrictions of remaining utility upper-bound (by pruning strategy). For 1-pattern* $< A >$*, its next-adjacent 1-patterns have* $< B >$*,* $< C >$*,* $< D >$ *in the original sequences, but according to extension strategy, we cannot backward extend* $< A >$ *with* $< D >$*. The remaining utility upper-bound (ruub) of* $< D >$ *is not less than minimum utility threshold.*

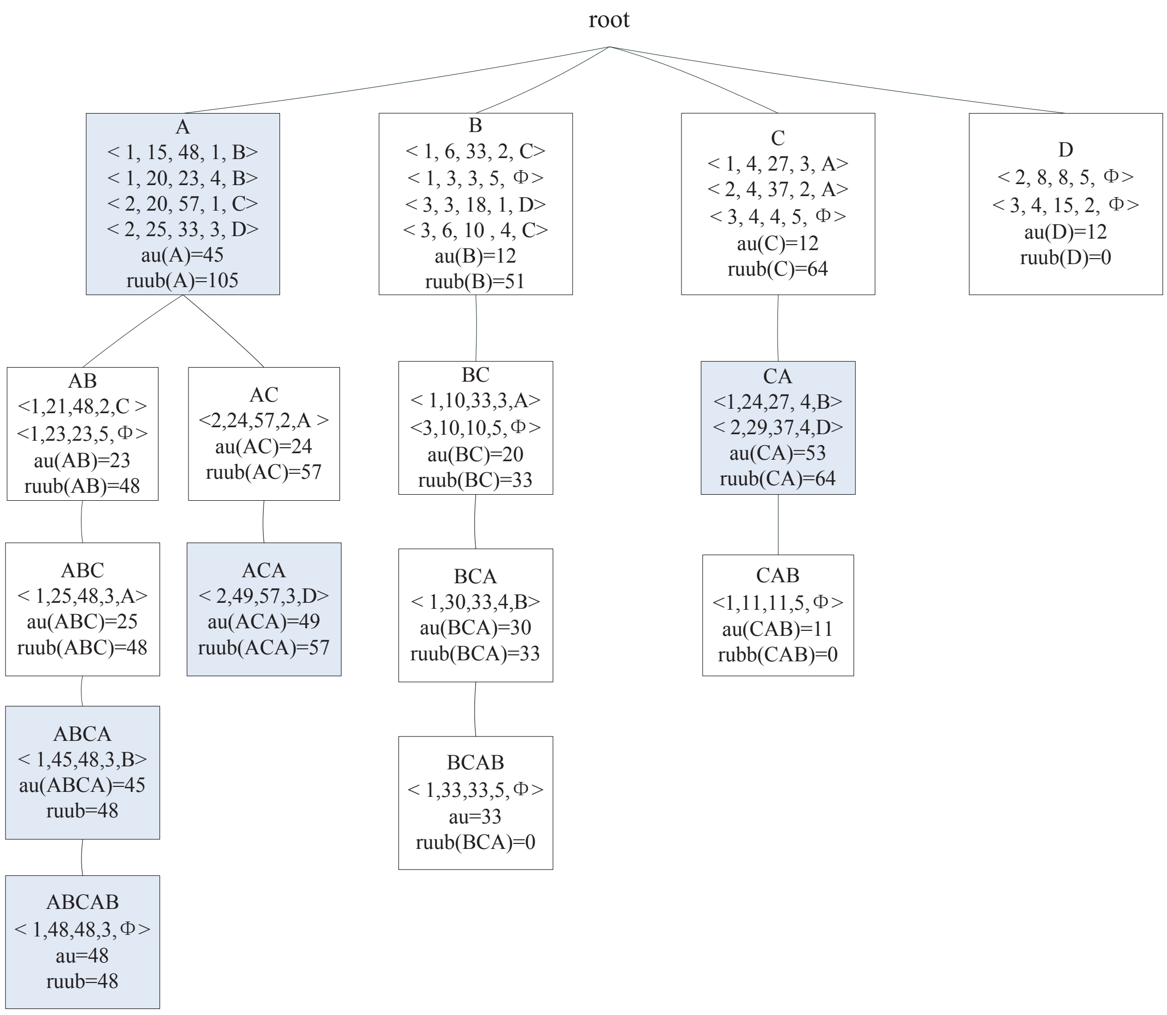

FIGURE 4. The process of backward extension

3.3. **The HUCP-Miner algorithm.** The main principle of the HUCP-Miner approach is purely extended from the HUI-Miner utility approach [9]. In Algorithm 2, it inputs a software executing sequence database D and the minimum utility threshold λ. The algorithm first scans the database to calculate the global twu of each 1-pattern i. Then, these 1-patterns which $twu(i) < \lambda$ should be ignored since they cannot be part of the high utility contiguous patterns as well as their any extension (by Lemma 2.1). It is important to note that the *twu* needs to be used here to prune 1-pattern before constructing the utility-location utility-list (UL-list) structure. So the twu pruning strategy can reduce the memory for UL-list structures of these unpromising 1-patterns. After that, starting to second scan the database, we construct UL-lists of upper-bound 1-patterns. These unpromising 1-patterns which are pruned by *twu* strategy do not appear in UL-lists, they are replaced by ϕ. For each upper-bound 1-patterns x, it calls the Procedure Extension to output all HUCPs with x as prefix. Finally, the algorithm obtains the set of HUCPs.

For Procedure Extension, it inputs a prefix contiguous pattern x and generates all the high utility contiguous patterns with x as prefix. It operates as follows. It scans the UL-list of x to calculate the $au(x)$ and the remaining utility upper-bound of x ($ruub(x) = ru(x)$). If $au(x)$ is not less than minimum utility threshold λ, x is a high utility contiguous pattern and output. Then, if $ruub(x)$ is not less than λ, it means that any backward extensions of x should be pruned (pruning strategy). Before backward extending x with y, we should make sure that the y is the next 1-pattern of y and apply the contiguous patterns extension strategy to avoiding unnecessary extensions (line 12). Finally, until all patterns' *ruub* are less than λ or their next-adjacent 1-patterns are not remaining utility upper-bound patterns, we stop mining contiguous patterns and can obtain the set of HUCPs.

Algorithm 2 : HUCP-Miner

```
Inputs: software executing sequence database: D, a user-sepecified minimum utility
threshold: $\lambda$
Output: high utility contiguous patterns
1.Scan D to calculate the twu of 1-patterns;
2.I $\leftarrow$ each 1-pattern $i$ such that $twu(i) \geq \lambda$;
3.Scan D again to construct UL-list structures for all upper-bound 1-patterns I;
4.for each pattern $x \in I$ do
5.   Extension($x$, UL-list, $I$, $\lambda$);
6.end for

Procedure: Extension
Inputs: a prefix pattern: $x$, the UL-list set: exULs, all upper-bound 1-pattern: $I$,
minimum utility threshold: $\lambda$
Output: all the high utility contiguous patterns with $x$ as prefix
7. if $au(x) \geq \lambda$ then
8.    output $x$ ;
9. end if
10. if $ruub(x) \geq \lambda$ then
11.    exULs = NULL;
12.    Next $\leftarrow$ each x.nitem that x.nitem $\in I$;
13.    for each $y \in$ Next do
14.       exULs = exULs+Call Construct($x, y$);
15.    end for
16.    if exULs! = NULL then
17.       Extension($x$, exULs, $I$, $\lambda$);
18.    end if
19.end if
```

4. **Experiments and Analysis.** In this section, to evaluate the performance of HUCP-Miner, we conduct extensive experiments on various databases, two different real-life datasets and compare HUCP-Miner with the state-of-the-art mining algorithm EUWPTM [18]. Both of algorithms are mining high utility contiguous patterns. These algorithms are

implemented in Java and the experiments are performed on 64 bits Windows 7 ultimate, Core i5-4440 CPU@3.10GHz and 4G Memory.

4.1. **Experimental datasets.** Two real-life datasets are used in our experiments. The datasets can be downloaded from FIMI Repository [20]. BMS WebView1 is click-stream data from a web store used in KDD-Cup 2000, another is anonymous retail market basket data from an anonymous Belgian retail store which is named Retail. The dataset BMS WebView1 has 59602 sequences, 497 distinct items and the average length of sequence is 2.5 items. The dataset Retail has 88162 sequences with 16470 distinct items and an average sequence length of 10.3 items. These datasets do not provide item utility, so we assign utility to each item ranging from 0.01 to 1 randomly.

4.2. **Running time.** We change the minimum utility threshold from 0.002 to 0.010 and test the performance of the HUCP-Miner algorithm on two real-life dataset Retail and BMS WebView1. We record the different running times on each dataset for different algorithms. As is shown in Figure 5 and Figure 6, we can see that the running times of two algorithms are reducing constantly by increasing the minimum utility threshold. Both of algorithms get more and more high utility contiguous patterns by increasing the minimum utility threshold. Compared with EUWPTM algorithm, HUCP-Miner algorithm's running time is faster. That is because the HUCP-Miner algorithm only needs two datasets scan by UL-list structure and applies the remaining utility upper bound pruning strategy and extension strategy to prune more unpromising patterns which avoid the costly generation and utility computation of unpromising patterns. On the other hand, we record the number of candidates on datasets BMS WebView1 and Retail. These results are shown in Figure 7 and Figure 8. 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 candidates produced by HUCP-Miner with UL-list structure is less than EUWPTM algorithm. What is more, the gap on Retail is bigger than that on BMS WebView1. Retail has many distinct items, and it needs more running time with EUWPTM due to projecting.

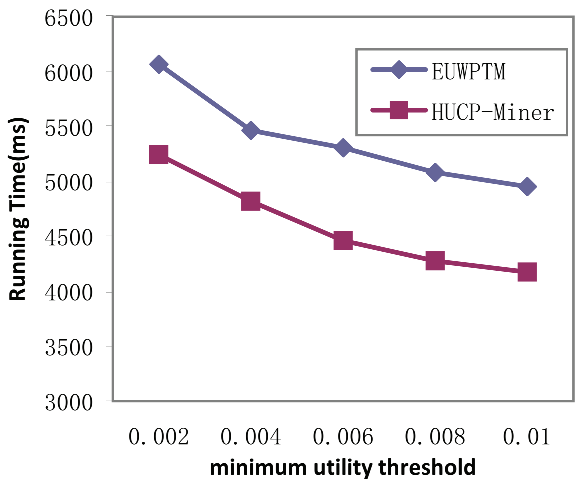

FIGURE 5. Running time on BMS WebView1

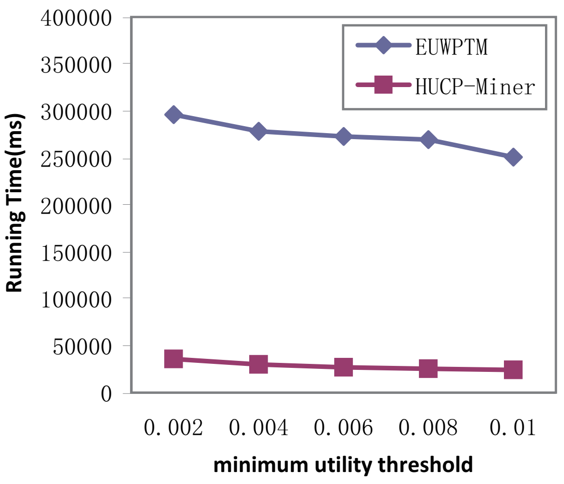

FIGURE 6. Running time on Retail

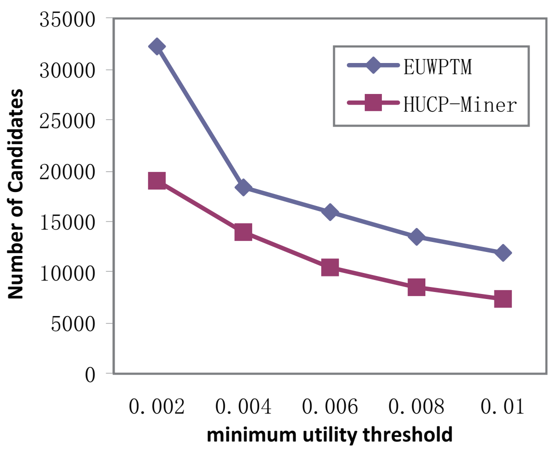

FIGURE 7. The number of candidates on BMS WebView1

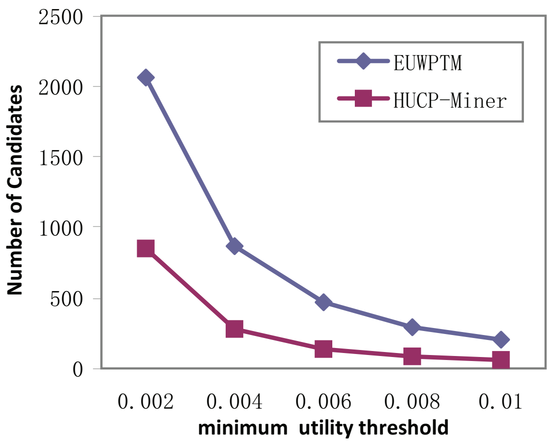

FIGURE 8. The number of candidates on Retail

4.3. **Scalability.** We test the scalability of the HUCP-Miner algorithm on memory consumption. As is shown in Figure 9 and Figure 10, we can find that the HUCP-Miner algorithm's memory consumption is always smaller than EUWPTM algorithm's. It means that our algorithm has better scalability than EUWPTM algorithm. The HUCP-Miner algorithm uses the UL-list structure to store patterns' information. By tighter upper bound, our algorithm can prune more unpromising patterns, so more memory spaces are saved.

From Figure 11 and Figure 12, we can see the different running times on various database sizes. Running time is tested by varying the number of sequences in the dataset Retail and BMS WebView1. No matter what number of sequences, the running time of HUCP-Miner is always less than that of EUWPTM. The UL-list structure, pruning strategy and extension strategy contribute to the efficiency of HUCP-Miner.

5. **Conclusions and Future Work.** In this paper, we mine these high utility contiguous patterns from the software executing traces. Considering that a software executing trace has many same events, we apply the maximum utility measure to calculate this utility of contiguous patterns. After the critical first steps, we propose UL-list structure to record these patterns' utility and location information. At the same time, this structure takes

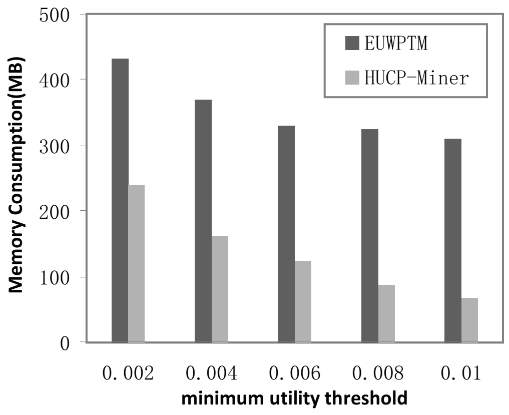

FIGURE 9. Memory consumption on BMS WebView1

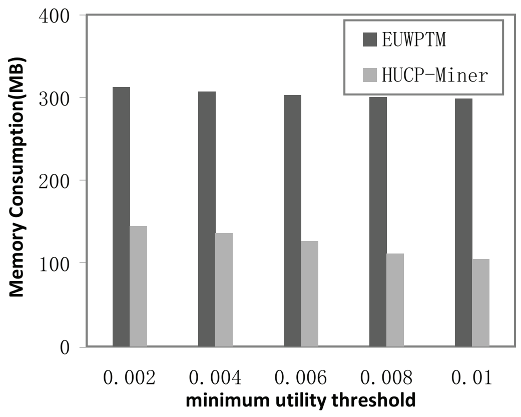

FIGURE 10. Memory consumption on Retail

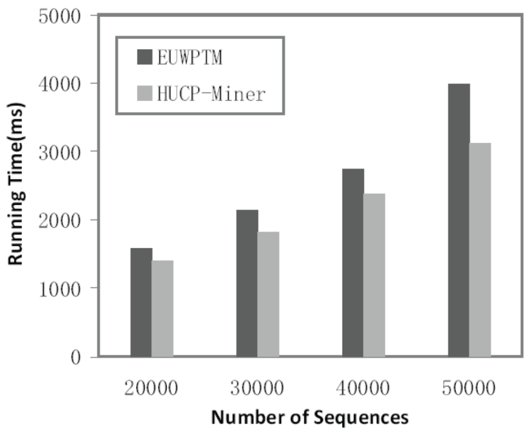

Figure 11. Running time on BNS WebView1 ($\lambda = 0.008$)

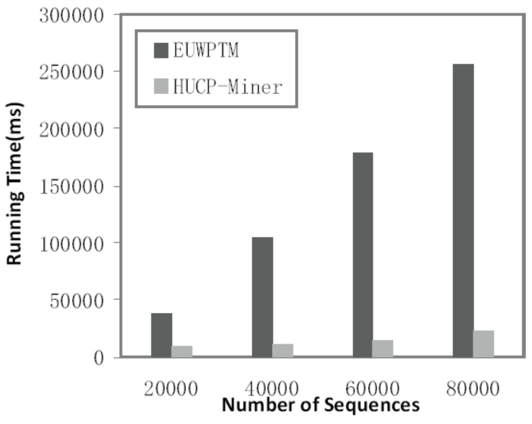

Figure 12. Running time on Retail ($\lambda = 0.008$)

a tighter utility upper bound (*ruub*) and extension strategy for eliminating unpromising patterns. We generate contiguous k-patterns from contiguous $(k-1)$-pattern recursively by backward extension without scanning the database. Based on the UL-list structure, we present a new algorithm named HUCP-Miner for mining high utility contiguous patterns from the software executing traces. An extensive experimental study on various databases shows that HUCP-Miner has better efficiency in both running time and scalability.

In the future, we want to make this algorithm to handle the real software executing traces. Software executing traces have many loops, so they have many contiguous repetitive patterns which are not useful for us. We intend to eliminate this contiguous repetitive patterns firstly, and then apply our algorithm to these treated software executing traces.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61572420 and No. 61472341, and the Natural Science Foundation of Hebei Province of 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] 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.

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

[4] 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.

[5] 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.

[6] Z. Xing, A brief survey on sequence classification, *J. ACM SIGKDD Explorations Newsletter*, vol.12, no.1, pp.40-48, 2010.

[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] 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.

[10] 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.
[11] P. Fournier-Viger and S. Zida, FOSHU: Faster on-shelf high utility itemset mining – With or without negative unit profit, *ACM*, 2015.
[12] C. F. Ahmed, S. K. Tanbeer and B. S. Jeong, A novel approach for mining high-utility sequential patterns in sequence databases, *ETRI Journal*, vol.32, no.5, pp.676-686, 2010.
[13] G. C. Lan, T. P. Hong and V. S. Tseng, Sequential utility mining with the maximum measure, *The 29th Workshop on Combinatorial Mathematics and Computation Theory*, pp.115-119, 2012.
[14] J. Yin, Z. Zheng and L. Cao, USpan: An efficient algorithm for mining high utility sequential pattern-s, *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.660-668, 2012.
[15] G. C. Lan, T. P. Hong, V. S. Tseng et al., Applying the maximum utility measure in high utility sequential pattern mining, *Expert System with Applications*, vol.41, no.11, pp.5071-5081, 2014.
[16] Y. F. Lin, C. W. Wu, C. F. Huang et al., Discovering utility-based episode rules in complex event sequences, *Expert Systems with Applications*, pp.1089-1091, 2015.
[17] L. Zhou, Y. Liu, J. Wang et al., Utility-based web path traversal pattern mining, *ICDM*, pp.373-380, 2007.
[18] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong et al., Efficient mining of utility-based web path traversal patterns, *ICACT*, pp.2215-2218, 2009.
[19] M. Thilagu and R. Nadarajan, Efficiently mining of effective web traversal patterns with average utility, *ICCCS*, vol.1, no.4, pp.444-451, 2012.
[20] *Frequent Itemset Mining Dataset Repository*, http://fimi.ua.ac.be/, 2012.