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

ICIC International ©2016 ISSN 1349-4198
pp. **1227–1239**

MINING HIGH UTILITY PATTERNS FROM SOFTWARE EXECUTING TRACES

Haitao He[1,2], Jiandi Wang[1,2,*], Cui Xu[1,2], Hao Wang[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
{ haitao; jdren }@ysu.edu.cn; *Corresponding author: wjdysu@sina.com

Received October 2015; revised February 2016

Abstract. *Software behavior mining is quite meaningful work. Finding meaningful patterns can help program maintainers detect the exception and improve work efficiency. These high utility software executing patterns shed light on software behavior and capture unique characteristic of software traces. In this paper, an efficient method called SHUP-Miner (Software High Utility Pattern Mining) is proposed to mine high utility patterns from software executing traces. In the algorithm, we firstly define the utility of each event in the software executing traces according to their importance of the software behavior. Secondly, a novel structure called improved utility list shorted as IUL is put forward. It stores patterns position and utility information which contributes to pruning space and extending patterns. Moreover, pattern extending strategy and IUCS (Improved Utility Co-occurrence Structure) are incorporated into SHUP-Miner to improve the efficiency. At last, we conduct experiments on both real and synthetic datasets. The results show that SHUP-Miner incorporating the efficiency-enhanced strategies demonstrates impressive performance.*

Keywords: High utility, Pattern mining, Software executing traces, Software behavior

1. **Introduction.** Billions of dollars are spent annually on software related cost. Improving software quality is a very significant goal of software engineering because software plays a very important role in many systems. Software behavior learning is one of the most important tasks in many stages of software development lifecycle [1]. Representative patterns are important to software maintenance, so it is desirable to mine representative software behavior patterns from software executing traces. As a result, we can reduce the maintenance cost by detecting failures in these patterns.

Software failure detection is a quite important application of software behavior. Usually software executing traces have many important behavior patterns which can capture the forward backward and in between constraint. Detecting these patterns can help us identify software executing violations. For example, considering the pattern <login, authenticate, logout>, after we login the system, it begins to authenticate, and then we logout the system. In order to mine representative patterns from software executing traces, some algorithms were developed. Li and Zhou [2] proposed a method called PR-Miner that used frequent itemset mining to efficiently extract implicit programming rules from large software code. To address software reliability issue, Han et al. [3] put forward a framework of frequent pattern-based classification. That frequent patterns have high quality features for classification through analyzing the relationship between pattern frequency and their discriminative power. Lo et al. [4] proposed a method to classify software behaviors based on past history or runs. The algorithm firstly extracts discriminative features to capture

failures, and then selects some important features for classification which are used to detect failures. However, patterns based on frequency neglect the pattern's position within the sequence; patterns occurring in the different positions of a trace are likely to represent different meanings. So it is appropriate to consider using positional information to enhance the discriminating power of patterns. Li et al. [5] came up with a method which used the pattern position distribution as features to detect software failure occurring through misused software patterns. To deepen research on software pattern mining, Lo et al. [6] introduced algorithms for mining iterative generators from program traces, and produced a set of representative rules which can capture the forward backward and in between constraints. A method based on relational association rule mining for detecting faulty entities in software systems was put forward [7].

From data mining viewpoint, software behavior is a series of entities which can be functions, methods, states of program running in the process of software running. On the base of algorithm charm [8], Negara et al. [9] presented a method to mine frequent code change patterns from fine-grained sequence of code changes. While these algorithms only consider the frequency of the pattern, other information is not considered such as the weight of the pattern. Some algorithms were proposed to efficiently mine high utility itemsets based on the idea of FP-Growth. Tseng et al. [10] proposed an efficient algorithm called UP-Growth for mining high utility itemsets with a set of techniques for pruning candidate itemsets. The information is maintained in UP-Tree data structure. However, many candidates were produced due to the loose upper bound. For this reason, an algorithm called UP-Growth$^+$ [11] was proposed for high utility itemsets mining. Several new strategies were proposed and the overestimated utilities of candidates can be reduced. To reduce the overestimated upper bound, Yun et al. [12] designed a method called MWS for mining weighted maximal frequent patterns over data streams. It used a partial maximum value MW to prune the low utility itemets. Since the downward-closure property cannot be directly used in high utility itemsets mining, designing a proper upper bound satisfying the downward-closure property is necessary.

In software executing traces, events in the sequence are ordered, so the high utility itemset mining algorithms are not applicable. Many algorithms were proposed for sequential pattern mining. Yun and Leggett [13] proposed an algorithm WSpan for weighted sequential pattern mining. Zhou et al. [14] designed a Two-Phase utility mining method to discover high utility path traversal patterns from weblog databases. Ahmed et al. [15] provided an algorithm using a pattern growth sequential mining approach to mine utility-based web path. However, many candidates are produced because of the loose upper bound. To get tighter upper bound, the IUA [16] algorithm was proposed. IUA was a projection-based algorithm using an improved strategy for mining weighted sequential pattern mining. Due to the tighter upper bound, many unpromising patterns are pruned and the candidate number is reduced. In that long patterns may result in very high utility value. And then, an efficient algorithm [17] to discover effective web traversal patterns was proposed based on average utility model. To resolve the problem due to pattern length, the algorithm mined high average utility patterns rather than actual utility. However, these algorithms cannot deal with the sequences with both forward and backward references. Based on the above issues, some works are done in the paper. The contribution of the paper is the following:

- We propose a new algorithm SHUP-Miner (Software High Utility Pattern Mining) to mine high utility patterns from software executing traces without window size constraint. In the algorithm, we integrate the IUCS (Improved Utility Co-occurrence Structure).

- A new data structure called IUL (Improved Utility List) is put forward to store pattern position and utility information. Extending strategy and pruning strategy have been designed to improve the efficiency of the algorithm.
- We evaluate our algorithm through a set of experiments.

The remaining paper is organized as follows. Section 2 gives the definitions. Section 3 presents the proposed algorithm SHUP-Miner and the Improved Utility Structure. Section 4 presents the experiments to analyze performance of the algorithm. Section 5 presents the conclusion and future work.

2. **Definitions.** In data mining terminology, an event corresponds to the execution of a statement, method, class, interface, etc. We can take it for grant that a software behavior can be viewed as a series of events. Let S be a series of distinct functions, and we denote S as $< e_{start}, e_{start+1}, \cdots, e_{end} >$, where e_i $(start \leq i \leq end)$ is an event in software executing traces. Let I denote all the distinct events in software executing traces and R denote the real number which is the utility value.

Definition 2.1. *SEP (Software Executing Pattern). We denote SEP as* $P(< e_i, \cdots, e_j >)$ *where* e_i *to* e_j *are the members of the sequence* S.

In fact, we focus on not only continuous patterns but also discontinuous patterns with arbitrary number of events between the neighbor events in the patterns. For the pattern <login, logout>, the two events are discontinuous and there can be any events not contained in the pattern between them in software executing traces. In the paper, each of these patterns reflects a software behavior. It can be mined by analyzing a set of software executing traces which is a series of method invocations.

Definition 2.2. *Subsequence. A SEP* $P =< e_1, e_2, \cdots, e_n >$ *is considered as a subsequence of SEP* $Q =< s_1, s_2, \cdots, s_m >$ *if there exist integers* $1 \leq i_1 < i_2 < i_3 < i_4 \cdots < i_n \leq m$ *where* $e_1 = s_{i_1}$, $e_2 = s_{i_2}$, $\cdots$, $e_n = s_{i_n}$. *We denote* $P \subseteq Q$.

Definition 2.3. *The utility of an event in a software executing sequence. We denote the utility of an event* e *in a software executing sequence* S *as* $u(e, S)$, *and it is defined as* $u(e, S) = iu(e, S) \times eu(i)$.

Here the utility of the event means the importance to the software behavior, different events with different utilities. We should note that $iu(e, S)$ is the internal utility of the event which means the occurrence times of the event in software execution sequence. $eu(e)$ is the external utility of event e (shown in Figure 1(b)) and it is defined according to the importance of the event.

Definition 2.4. *The utility of software executing pattern* P *in a sequence* S *is defined as* $u(P, S) = \sum_{e \in P} u(e, S)$.

Definition 2.5. *The utility of software executing pattern* P *is denoted as* $u(P)$ *and defined as* $u(P) = \sum_{P \subseteq S \wedge S \in D} u(P, S)$.

It is the sum of the pattern utilities in sequences containing P. Consider the example in Figure 1(a), pattern $u(abc) = u(abc, S_1) + u(abc, S_2) + u(abc, S_4) = (1 + 2 + 3) + (1 + 2 + 3) + (1 + 2 + 3) = 18$.

Definition 2.6. *The utility of the sequence* S *is the sum of all events utilities in* S. *It is denoted as* $su(S) = \sum_{e \in S} u(e, S)$.

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

Sid	Sequences	SU
S1	<a b c>	6
S2	<a b d c>	10
S3	<b d e c>	14
S4	<a b d c f e>	18

(a) The sample database

Event	a	b	c	d	e	f
Utility	1	2	3	4	5	3

(b) The external utility table

Event	a	b	c	d	e	f
SWU	34	48	48	42	32	18

(c) SWU of events

FIGURE 1. An example

Definition 2.7. *The sequence-weighted utility of the software executing pattern P is defined as $swu(P) = \sum_{P \subseteq S \bigwedge S \in D} su(S)$.*

$swu(P)$ is the sum of the utilities of all the sequences containing P and it is the upper bound of $u(P)$.

Definition 2.8. *High Utility Pattern. If $u(P)$ is no less than minutil specified by user, we call this pattern high utility pattern.*

Lemma 2.1. *If the $swu(P)$ is less than minimum utility shorted as minutil, for $\forall Q$ where $P \subseteq Q$, Q cannot be high utility pattern.*

Definition 2.9. *Improved Utility Co-occurrence Structure (IUCS). IUCS is a collection of tuples where $(s_i, s_j, u_{ij}) \in I' \times I' \times R$ such that $swu(s_i s_j) = u_{ij}$.*

In [18], Fournier-Viger et al. proposed EUCS structure used in high utility itemsets mining. The EUCS structure is a triangular matrix while the IUCS is a matrix. In EUCS, tuple (a, b, c) is the same as (b, a, c) while they are different in IUCS. Tuple (a, b, c) means pattern $swu(ab) = c$ while tuple (b, a, c) is $swu(ba) = c$. The IUCS structure can be used in candidates pruning. Because the SWU value in IUCS is lower than it in EUCS, many patterns having low SWU value can be pruned. Our algorithm shows better pruning efficiency and candidates number is greatly reduced. We implement the IUCS structure by hash map. Due to very few events co-occurring with other events, the IUCS is space saving. This structure requires memory bounded by $|I| \times |I|$ (I is all the events).

Lemma 2.2. *For a given pattern Px (Px is pattern P combined with event x) and an event y, if there is no tuple (x, y, c) in IUCS that $c \geq$ minutil, it is no need to extend pattern Px with y.*

	a	b	c	d	e
a		34	34	28	18
b			48	42	32
c					18
d			42		32
e			14		

(a) IUCS Structure

Event	a	b	c	d	e
SWU	31	45	45	39	29

(b) new SWU of events

FIGURE 2. IUCS structure and SWU

Proof: Pattern xy is a subsequence of Pxy. According to Lemma 2.1, if $swu(xy)$ is less than $minutil$, then all extensions of pattern xy must not be high utility patterns. Thus we can conclude that $u(Pxy)$ is less than $minutil$ and we do not need to extend pattern Px with y. □

3. **Mining High Utility Patterns from Software Executing Traces.** In this section, we describe our method at length. In Section 3.1, a new structure IUL is proposed. The structure stores pattern position and utility information which contributes to pruning space and extending patterns. Sections 3.2 and 3.3 are the pruning strategy and extending strategy respectively. In Section 3.4, we propose an algorithm called SHUP-Miner (Software High Utility Pattern Mining) mining high utility patterns from software executing traces. The main process of our method can be seen as Figure 3.

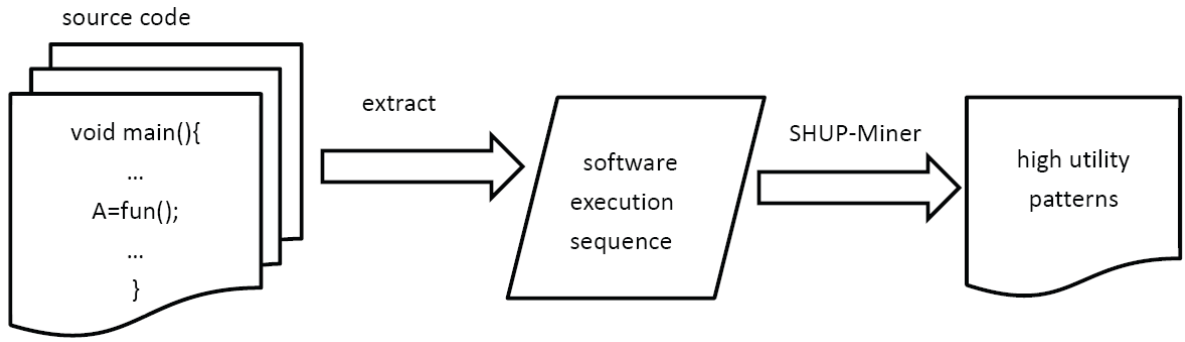

FIGURE 3. The main framework of SHUP-Miner

3.1. **IUL (Improved Utility List).** The improved utility list of P is a collection of tuples $(sid, index, util, rutil)$.

- sid is a sequence serial number containing P
- $index$ represents the last event's position of the pattern P in the sequence S
- $util$ is the utility of pattern P in S
- $rutil$ is the remaining utility of the pattern P in S.

$rutil$ is the sum of the utilities of all events not in P of S. We define $rutil$ as $rutil = \sum_{i \in S_{sid}/P} u(i, S_{sid})$. We denote the improved utility list of P as IUL(P). For pattern $P =< bd >$ in Figure 1(a), the IUL(P) is $\{(2, 2, 6, 3), (3, 1, 6, 8), (4, 2, 6, 8)\}$.

3.2. **Pruning strategy.** Due to the fact that the number of candidate patterns is quite large, it is excessively time-consuming and memory-consuming to discover all high utility patterns. To reduce the search space, we can exploit the $utils$ and $rutils$ in the IUL of a pattern. The sum of all the $utils$ in IUL of a pattern is the utility of the pattern and thus the pattern is high utility pattern if the sum exceeds a given $minutil$ according to Definition 2.8. The sum of all the $utils$ and $rutils$ in the improved utility list provides SHUP-Miner with the key condition about whether the pattern should be pruned or not. Then we get the following lemma.

Lemma 3.1. *Given the improved utility list of pattern P, if the sum of all the utils and rutils in the utility-list is less than a given minutil, any extension P' of P is not high utility pattern.*

Proof: Suppose $sid(S)$ denotes the sid of S, $P.sids$ denotes the sid set in IUL of P, and $P'.sids$ that in P', then for $\forall S \supseteq P'$:

$$
\begin{aligned}
&\because P' \text{ is an extension of } P \Rightarrow (P' - P) = (P'/P) \\
&\qquad P \subset P' \subseteq S \Rightarrow (P'/P) \subseteq (S/P) \\
&\therefore u\,(P', S) = u(P, S) + u((P' - P), S) \\
&\qquad = u(P, S) + u((P'/P), S) \\
&\qquad = u(P, S) + \sum_{e \in (P'/P)} u(e, S) \\
&\qquad \leq u(P, S) + \sum_{e \in (S/P)} u(e, S) \\
&\qquad = u(P, S) + ru(P, S) \\
&\because P \subset P' \Rightarrow P'.sids \subseteq P.sids \\
&\therefore u(P') = \sum_{sid(S) \in (P'.sids)} u(P', S) \\
&\qquad \leq \sum_{sid(S) \in (P'.sids)} (u(P, S) + rutil(P, S)) \\
&\qquad \leq \sum_{sid(S) \in (P.sids)} (u(P, S) + rutil(P, S)) \\
&\qquad < minutil
\end{aligned}
$$

□

Example 3.1. *Take the sequences in Figure 1 as an example. For event c, the sum of its all utils and rutils* $3 + 3 + 3 + 3 + 5 = 17$ *is less than minutil 20. It is unnecessary to extend pattern c, because it is unpromising pattern.*

3.3. **Pattern extending strategy.** For a given pattern Px and another pattern Py which is a pattern P concatenated with event y, if IUL(Px).index $\geq$ IUL(Py).index, the candidate pattern Pxy cannot be generated.

We know that the events in the pattern are ordered, and thus we can exploit the position information of the event in sequence to decrease the number of candidates generated during the mining process of the algorithm. Before extending pattern Px with Py, we should make sure that the position of y is after the event x in a sequence. The conditional judgment can be implemented through the field index of IUL structure. We just need to make sure that IUL(Px).index is less than IUL(Py).index, because index is the last events position of the pattern P in the sequence S. By this strategy, we can not only get the continuous patterns but also discontinuous patterns without window size constraint.

Example 3.2. *In* S_1*, index of a is 0 and index of b is 1. The former is less than the latter and we extend a with b in* S_1*. The index in IUL(ab) is updated into 1 and the util of ab in sid1 is 3 and the remaining utility is 3. In* S_2 *and* S_4*, the extending process is similar to* S_1*. We can see the result of extending a with b in Figure 4.*

3.4. **The SHUP-Miner algorithm.** The SHUP-Miner mining the high utility patterns from software executing traces is given in Algorithm 1. The algorithm integrates IUCS and IUL structure. Pruning strategy and extending strategy are used to avoid costly utility computation and construction IUL of candidates. Before constructing the IUL of a new pattern, we compare the utility in IUCS with minimum utility threshold. And then extend the pattern with pruning strategy and extending strategy through which the patterns with low utilities are pruned and the candidates number is reduced. The following paragraph is the detail implementing steps of the algorithm SHUP-Miner.

In Algorithm 1, a software executing sequence database with utility values and the minimum utility threshold are the input information of the main procedure. We firstly assign each event a utility value randomly. In our algorithm, we only need to scan the database two times. The following are the detail steps. At the first time scan the database,

pattern	a		
sid	index	util	rutil
1	0	1	5
2	0	1	9
4	0	1	14

pattern	b		
sid	index	util	rutil
1	1	2	3
2	1	2	7
3	0	2	12
4	1	2	12

pattern	c		
sid	index	util	rutil
1	2	3	0
2	3	3	0
3	3	3	0
4	3	3	5

pattern	d		
sid	index	util	rutil
2	2	4	3
3	1	4	8
4	2	4	8

pattern	e		
sid	index	util	rutil
3	2	5	3
4	4	5	0

pattern	ab		
sid	index	util	rutil
1	1	3	3
2	1	3	7
4	1	3	12

FIGURE 4. IUL of events

the algorithm calculates the SWU of each event. The algorithm filters out these events having the SWU value less than *minutil* according to Lemma 2.1 and puts these left events into the set E at line 2. And then we scan the database at second time. During this database scan, we recalculate the SWU of each event and build the IUL of each event $e \in E$ and the IUCS structure. In our algorithm, we use hash map structure to implement the IUCS structure. Building the IUCS is very fast (it is performed with a single database scan) and it needs very little memory. After the construction of the IUCS, we begin to explore patterns at depth-first by calling the recursive procedure Growth.

In the Growth procedure, the inputs of the algorithm are a pattern P, a set of improved utility lists of extensions of P, *minutil* and the IUCS structure. The following are the detail steps. For each extension Px of P, if the *sumutil* of IUL(Px) is no less than *minutil*, then Px is a high utility pattern and it is output (reference to Definition 2.8) at line 6 to 8. Then, we need to judge the sum of *util* and *rutil* in IUL(Px) whether it is bigger than *minutil* or not. If the sum value is bigger than *minutil*, pattern Px should be explored at line 9. We should initialize the extensions of Px empty at line 10. And then we extend Px with each element Py in extensions of P. Before we extend Px with Py to get Pxy, we need to check the $swu(xy)$ in IUCS. According to Lemma 2.2, if $swu(xy)$ in IUCS is less than *minutil*, we do not extend Px with Py. Otherwise, we extend Px with Py by calling procedure Construct. After extension of Pxy, we put IUL(Pxy) into IULs of extensions of Px at line 14. After extending Px, we call the procedure Growth to extend each pattern in extensions of Px recursively. The Growth procedure starts from single events and explores the search space of patterns recursively by appending single event.

In the Construct procedure, there are some differences between HUI-Miner [19] and our algorithm SHUP-Miner. When we extend the pattern Px with new pattern Py, the pattern extending strategy is applied. We should make sure that the new y's position is after the last event of pattern Px. In other words, the condition IUL(Px).index $<$ IUL(Py).index should be satisfied. IUL(Px).index is the position of event x, the last event of pattern Px. The conditional judgment can be implemented through the IUL easily.

Algorithm 1 : SHUP-Miner

Input: D: software executing sequence database, *minutil*: a user-sepecified threshold
Output: the set of high utility patterns
1. Scan D to calculate the SWU of single event;
2. E $\leftarrow$ each event e such that $swu(e) > minutil$;
3. Scan D again to recalculate the SWU and build the improved utility list of each event $e \in$ E and build the IUCS structure;
4. **Growth**(ϕ, E, *minutil*, IUCS);

Procedure: Growth
Input: P: a pattern, exIULs(P): a set of improved utility lists of all P's extension, *minutil*, IUCS;
Output: the set of high utility patterns;
5. **foreach** IUL(Px) $\in$ exIULs(P) **do**
6. **if** IUL(Px).sumutils $\geq$ *minutil then*
7. output Px;
8. **end if**
9. **if** (IUL(Px).sumutils + IUL(Px).sumrutils) $\geq$ *minutil* **then**
10. exIULs(Px) $\leftarrow \phi$;
11. **foreach** IUL(Py) $\in$ exIULs(P) **do**
12. **if** $(x, y, c) \in$ IUCS such that $c \geq$ *minutil* **then**
13. IUL(Pxy) $\leftarrow$ **Construct**(IUL(P), IUL(Px), IUL(Py));
14. exIULs(Px) $\leftarrow$ exIULs(Px) $\bigcup$ IUL(Pxy);
15. **end if**
16. **end for**
17. **Growth**(Px, exIULs(Px), *minutil*, IUCS);
18. **end if**
19. **end for**

Procedure: Construct
Input: IUL(P), IULs(Px), IUL(Py);
Output: IUL(Pxy);
20. IUL(Pxy) $\leftarrow \phi$;
21. **foreach** tuple $ex \in$ IUL(Px) **do**
22. **if** $\exists\, ey \in$ IUL(Py) and ex.tid==ey.tid and ex.index < ey.index **then**
23. **if** IUL(P) $\neq \phi$ **then**
24. Search element $e \in$ IUL(P) such that e.tid==ex.tid
25. IUL(exy)$\leftarrow$(ex.tid, ey.index, ex.util+ey.util-e.util, ey.rutil);
26. **end if**
27. **else**
28. IUL(exy)$\leftarrow$(ex.tid, ey.index, ex.util+ey.util, ey.rutil);
29. **end if**
30. IUL(Pxy) $\leftarrow$ IUL(Pxy) $\bigcup$ IUL(exy);
31. **end if**
32. **end for**
33. **return** IUL(Pxy);

Example 3.3. *Take the sample database in Figure 1 as an example. The external utility of the events in the database is shown in Figure 1(b). Assume the minimum utility threshold is 20. Follow the algorithm. At the beginning, scan the database to calculate the SWU of each event according to Definition 2.6. The result is shown in Figure 1(c). Eliminate the event f because its SWU is 18 which is less than minimum utility 20. Then scan the database again to recalculate the SWU shown in Figure 2(b) and build IUL(a), IUL(b), IUL(c), IUL(d), IUL(e) shown in Figure 4 and build the IUCS structures shown in Figure 2(a). Then we will mine high utility patterns. After building the IUL of each pattern, we begin to extend these patterns.*

Example 3.4. *Figure 5 depicts the extending process. For pattern a, the sum of its current utility and remaining utility is 31 which is bigger than minutil 20, and then look to the IUCS extending pattern a. The utility of pattern ab is 31 which is bigger than minutil 20, and then we extend a with b. In S_1, index of a is 0 and index of b is 1. The former is less than the latter and extend a with b in S_1. The index in IUL(ab) is updated into 1, the util of ab in S_1 is 3 and the remaining utility is 3. In S_2 and S_4, the extending process is similar to S_1. The IUL of pattern ab is shown in Figure 4.*

Back to IUCS, we extend a with c. We do not extend pattern c and e, because they are not promising high utility patterns. Then we will extend pattern ab with ac. After finishing extending a, we need to extend pattern b, c, d, e. We will extend each pattern in IULs recursively. Shaded rectangles represent high utility patterns. We can see that the

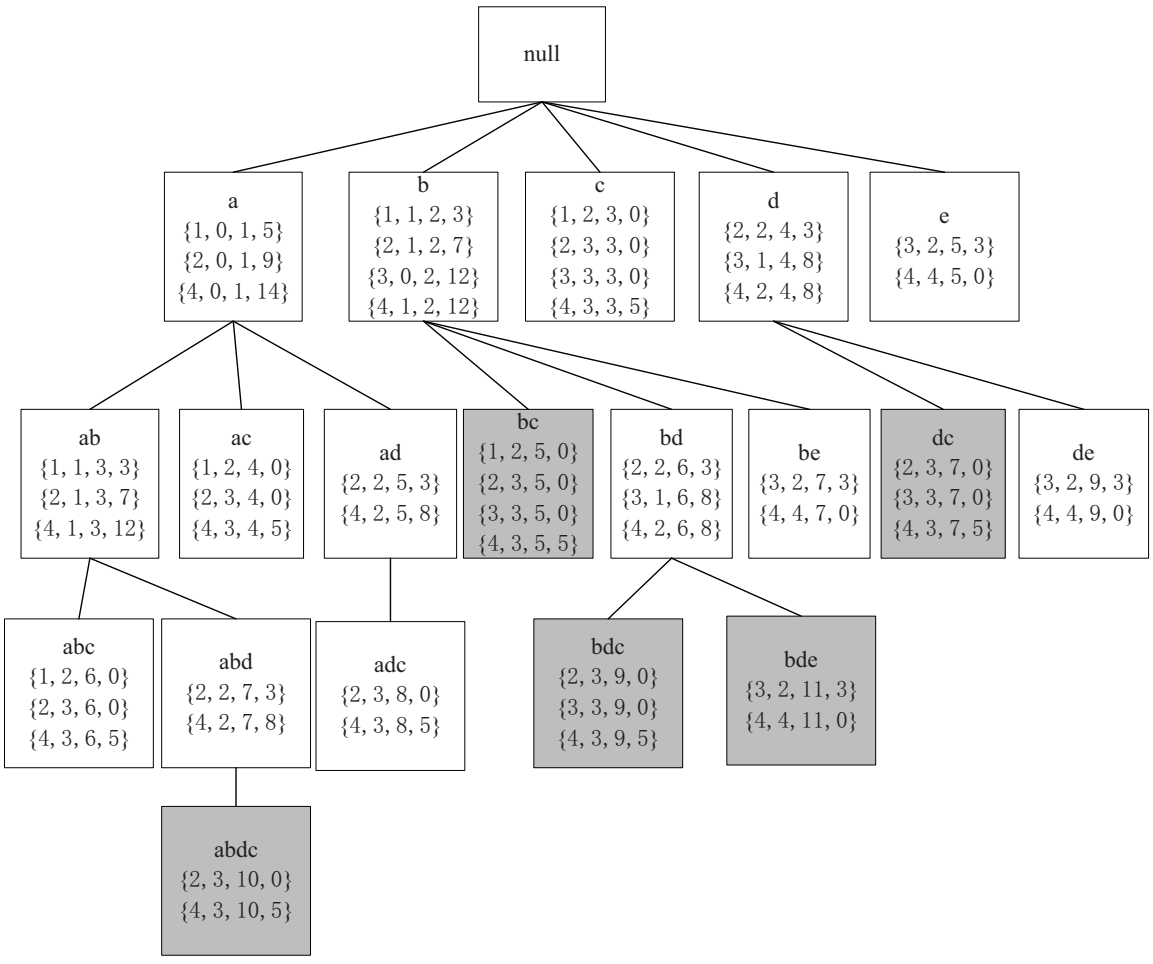

FIGURE 5. The procedure of SHUP-Miner

high utility patterns are not always consecutive pattern. For pattern bc, it is consecutive in S_1 while in S_2, it is not.

4. **Experiment.** To evaluate the performance of SHUP-Miner algorithm, we have done extensive experiments on both real-life and synthetic datasets. The datasets can be downloaded from FIMI Repository [20]. In the experiment, SHUP-Miner is compared with the state-of-the-art mining algorithm IUA [16] and EUWPTM [15]. Both of the algorithms are used to mine high utility pattern. We record the running time and memory consumption of these algorithms to assess the efficiency and scalability of SHUP-Miner. In this section, experimental results are reported and discussed.

Experiments were performed on a computer with a third generation 64 bit Core i8 processor running Windows 8 and 4 GB of free RAM. All memory and time measurements were done using the Java API.

4.1. **Running time.** We run the three algorithms on datasets retail and T10I4D100K. We change the minimum utility of each threshold from 0.001 to 0.011. The dataset retail has 88162 transactions with 16470 distinct items and an average transaction length of 10.3 items. The T10I4D100K has 100000 transactions with 870 distinct items and an average transaction length of 10.1 items. We randomly assign utility to each item. The utility of items in dataset is from 0.01 to 1.

For each dataset we record the run time. We can see the results in Figure 6 and Figure 7 and we can conclude that the running time of these algorithms gets deceased by increasing the minimum utility threshold. The result also shows that SHUP-Miner is faster than EUWPTM and IUA. That is because we use the structure IUL and pruning strategy which can prune more unpromising patterns without constructing their Improved Utility List. Only two times database scanning is needed and the running time of the algorithm is greatly reduced.

The numbers of candidates on dataset Retail and T10I4D100K are shown in Figure 8 and Figure 9. It is observed that the number of candidate patterns is decreasing with the increasing minimum utility threshold. With the minimum threshold increasing, more unpromising patterns are pruned. We can see from the result that the number of candidates produced by SHUP-Miner is less than other two algorithms by taking advantage of pruning strategy and extending strategy. The lower minimum utility threshold is, the

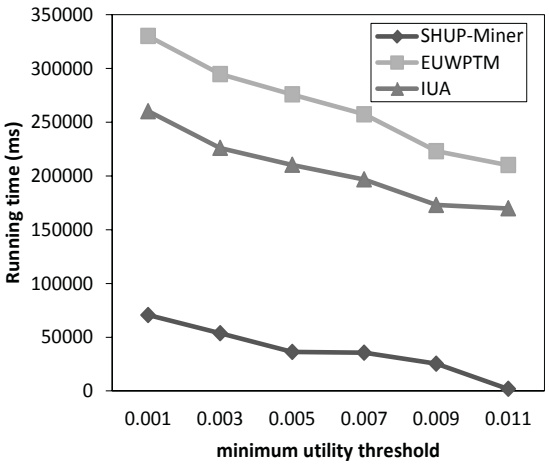

FIGURE 6. Running time on Retail

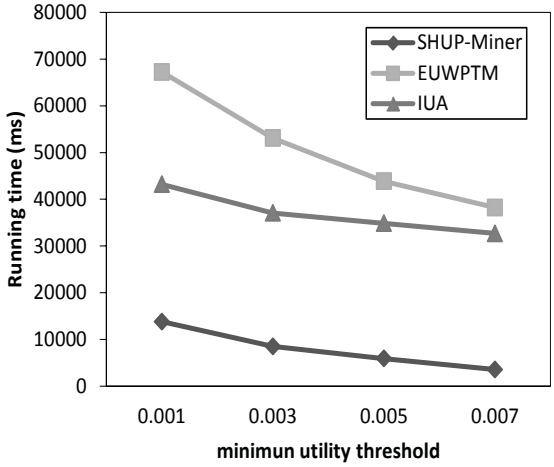

FIGURE 7. Running time on T10I4D100K

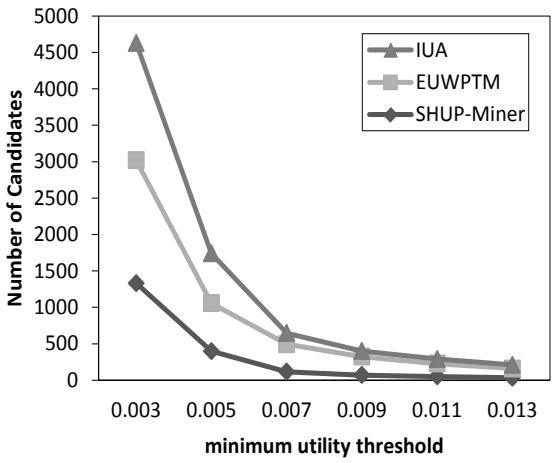

FIGURE 8. Number of candidates on Retail

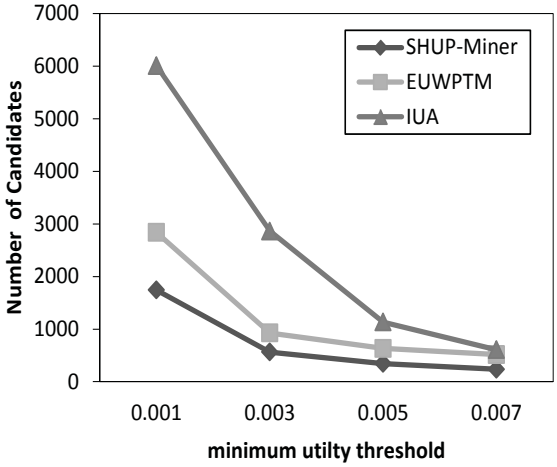

FIGURE 9. Number of candidates on T10I4D100K

bigger the gap between them. When the minimum threshold is larger than 0.007 on retail, the candidate number change trend tends to be stable.

4.2. **Scalability.** We studied the scalability of SHUP-Miner algorithm on memory consumption shown in Figure 10 and Figure 11. We can see that the memory consumption is decreasing with the minimum utility threshold increasing. It is observed that the memory consumption of SHUP-Miner changing is stable with the minimum threshold decreasing. The algorithm has better scalability than other two algorithms. The memory consumption of SHUP-Miner is less than EUWPTM and IUA. It benefits from the less number candidates produced by SHUP-Miner. Pruning strategy and IUCS structure lead to the tighter upper bound and eliminate more unpromising patterns and then candidates number is then decreased. In summary, the algorithm SHUP-Miner has good scalability.

From Figure 12 and Figure 13, we can see that the running time on different database sizes. Running time is studied by varying the number of sequences in the dataset Retail and T10I4D100K. The running time of SHUP-Miner is less than IUA. Pruning strategy and pattern extending strategy contribute to the better efficiency of SHUP-Miner.

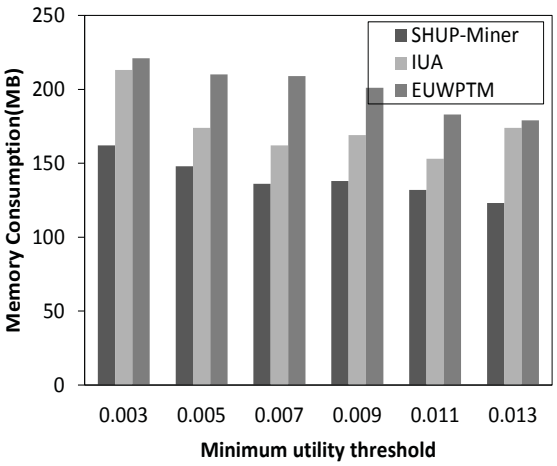

FIGURE 10. Memory consumption on Retail

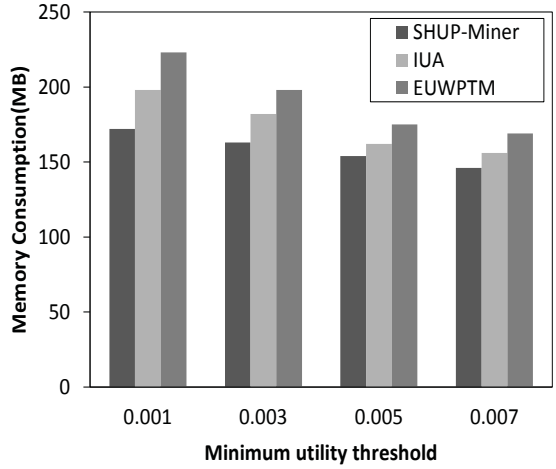

FIGURE 11. Memory consumption on T10I4D100K

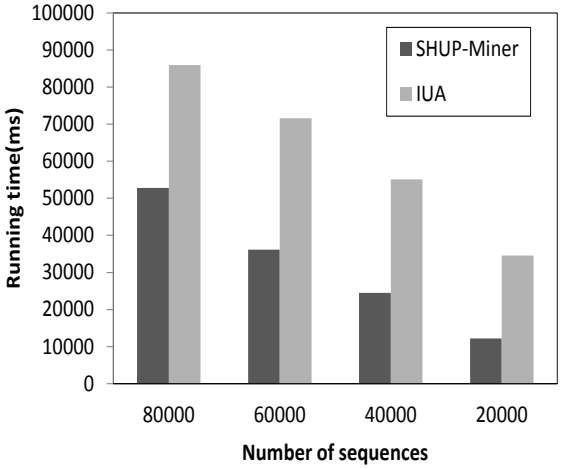

FIGURE 12. Running time with different database sizes on Retail according to $\varepsilon = 0.003$

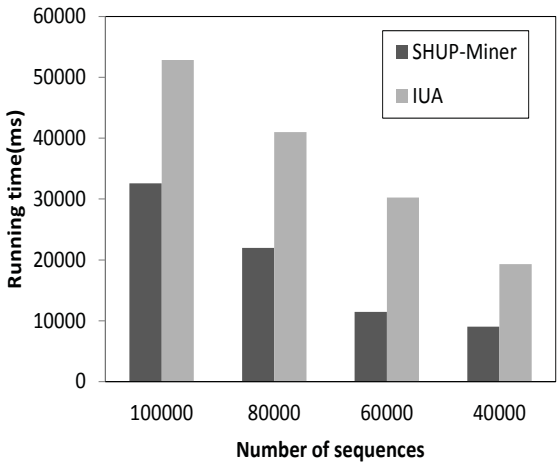

FIGURE 13. Running time with different database sizes on T10I4D100K according to $\varepsilon = 0.003$

5. **Conclusions and Future Work.** In this paper, we proposed an efficient algorithm SHUP-Miner for mining high utility patterns from software executing traces. First we construct the IUL structure to store the pattern position and utility information. Pruning strategy and extending strategy are designed in the algorithm to avoid the costly utility computation and construct IUL of candidates. The algorithm needs two times database scan. During the first time database scan, we calculate the SWU value of each event and filter out the events having SWU value less than *minutil*. In the second database scan, we construct IUL and IUCS structure and mine high utility patterns at depth-first recursively. We start from single events and explore the search space of patterns recursively by appending single event. We have demonstrated the performance of SHUP-Miner algorithm on both real and synthetic datasets. Experimental results show that SHUP-Miner has better efficiency in run time and good scalability than other algorithms. This benefits from the pruning strategy and pattern extending strategy which can lead to the tighter upper bound and eliminate more unpromising patterns.

In the future, we will analyze the representative patterns mined by our research further to identify software failures occurring through misused software patterns so as to detect software exception.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61170190 and 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] Z. Li and Y. Zhou, PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code, *ACM SIGSOFT Software Engineering Notes*, vol.30, no.5, pp.306-315, 2005.

[3] H. Cheng, X. Yan, J. Han et al., Discriminative frequent pattern analysis for effective classification, *ICDE*, pp.716-725, 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] D. Lo, J. Li, L. Wong et al., Mining iterative generators and representative rules for software specification discovery, *IEEE Trans. Knowledge and Data Engineering*, vol.23, no.2, pp.282-296, 2011.

[7] 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.

[8] M. J. Zaki and C.-J. Hsiao, CHARM: An efficient algorithm for closed itemset mining, *SDM*, vol.2, pp.457-473, 2002.

[9] S. Negara, M. Codoban, D. Dig et al., Mining fine-grained code changes to detect unknown change patterns, *Proc. of the 36th International Conference on Software Engineering*, pp.803-813, 2014.

[10] 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.

[11] 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.

[12] U. Yun, G. Lee and K. H. Ryu, Mining maximal frequent patterns by considering weight conditions over data streams, *Knowledge-Based Systems*, vol.55, pp.49-65, 2014.

[13] U. Yun and J. J. Leggett, WSpan: Weighted sequential pattern mining in large sequence databases, *2006 the 3rd International IEEE Conference on Intelligent Systems*, pp.512-517, 2006.

[14] L. Zhou, Y. Liu, J. Wang et al., Utility-based web path traversal pattern mining, *ICDM*, pp.373-380, 2007.

[15] 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.

[16] G. C. Lan, T. P. Hong and H. Y. Lee, An efficient approach for finding weighted sequential patterns from sequence databases, *Applied Intelligence*, vol.41, no.2, pp.439-452, 2014.

[17] M. Thilagu and R. Nadarajan, Efficiently mining of effective web traversal patterns with average utility, *ICCCS*, vol.1, no.4, pp.444-451, 2012.

[18] 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.

[19] 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.

[20] *Frequent Itemset Mining Dataset Repository*, http://fimi.ua.ac.be/, 2012.