ICIC Express Letters ICIC International ©2016 ISSN 1881-803X
Volume **10**, Number **1**, January **2016** pp. **211–218**

EFFICIENTLY MINING TOP-K HIGH UTILITY PATH PATTERNS IN SOFTWARE DYNAMIC CALL GRAPH

Haitao He[1,2], Tengteng Yin[1,2,*], Hongfei Wu[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
[*]Corresponding author: yin_teng_teng@sina.com

Received June 2015; accepted September 2015

Abstract. *Software behavior mining is a very meaningful work. Finding that desirable patterns can assist the program maintainers to comprehend the software adequately. Although the existing high utility pattern mining algorithms can discover all the patterns satisfying a given minimum utility, it is often difficult for users to set a proper minimum utility. In this paper, an efficient algorithm TUPP is presented to mine top-k high utility path patterns in software dynamic call graph. In TUPP, software paths are extracted from software dynamic call graph and the utility of functions is defined firstly. Secondly, a novel structure called PUAList is put forward. The structure stores both the adjacency items and the utility information which is heuristic information for pruning the search space. In addition, two strategies for raising the threshold and one pruning strategy are incorporated into TUPP to increase the efficiency. At last, our experiments are conducted on both synthetic and real datasets. The results show that TUPP incorporating the efficiency-enhanced strategies demonstrates impressive performance.*
Keywords: Software dynamic call graph, Top-k pattern, Software behavior mining

1. **Introduction.** Improving software quality is an important goal of software engineering because software plays a critical role in businesses, governments and societies. Some sequential pattern mining algorithms have been successfully adopted to some complex behaviors analysis [1]. Software behavior learning is one of the most important tasks in all stages of software development lifecycle [2]. When a software runs, it will produce a trace which can be considered as a software dynamic call graph. Some interesting behavior patterns can be mined by analyzing these dynamic program traces. As a result, how to mine desirable patterns from large amounts of software traces is a very meaningful work.

In pattern mining, users generally set a minimum threshold to extract crucial patterns from the databases. However, it is not easy for the users to determine an appropriate minimum threshold. To address this issue, the concept of extracting top-k patterns had been proposed in [4, 5] to select the patterns with the highest frequency. This makes it much easier and more intuitive than determining a minimum support. Nevertheless, the traditional frequency-based sequential pattern mining algorithms assume that all the items have the same significance. Therefore, some patterns with high importance but low frequency may not be discovered by using these approaches. To handle this, the high utility pattern mining [3] model was proposed. However, the downward-closure property in frequency-based mining cannot be directly used in utility mining. Therefore, it is hard to directly apply the techniques of top-k frequent pattern mining into top-k utility pattern mining. Top-k utility itemset mining (TKU) [6] which was an algorithm based on two phase, was proposed for top-k high utility pattern mining. In phase I, TKU generated potential top-k high utility patterns (PKHUPs) with the help of the Pre-Evaluation (PE) strategy and a global UP-Tree. In phase II, TKU calculated real utility for each candidate

starting from one with the highest estimated utility. Yin et al. [7] proposed an efficient algorithm named TUS for mining top-k high utility sequential patterns from utility-based sequence databases. In their approach, a pre-insertion and a sorting strategy had been introduced to raise the minimum utility threshold. Ryang and Yun [8] put forward an efficient algorithm for mining top-k high utility patterns. They also suggested three strategies that could raise the minimum utility threshold effectively. The three strategies made a considerable contribution to reducing the search space.

Considering that the items in software function call sequences are ordered and consecutive, the general sequential pattern mining algorithms are not applicable. The path sequential pattern has a property that the items appearing in the pattern must be adjacent with respect to the underlying ordering as defined in the pattern. Zhou et al. [9] proposed a Two-Phase utility mining method to discover high utility path traversal patterns from weblog databases. Due to that their upper bound is loose, the number of candidates is large. Ahmed et al. [10] provided a very efficient algorithm for utility-based web path traversal mining by using a pattern growth sequential mining approach. However, longer patterns with less item utility may result in higher values. For this reason, Thilagu and Nadarajan [11] proposed an efficient algorithm to discover effective web traversal patterns based on average utility model. To reveal better results and resolve the problem occurring due to pattern length, the algorithm mined high average utility patterns rather than patterns with actual utility. Despite all this, the algorithm cannot deal with the sequences with both forward and backward references. Ahmed et al. [12] proposed a framework to mine high utility web access by two new tree structures, avoiding the level-wise candidate generation and test methodology.

In this paper, combining utility pattern mining with path pattern mining, we propose an efficient approach TUPP to mine top-k high utility path patterns. Considering the characteristics of software, the utility of functions is defined. A novel structure called PUAList is proposed which stores both the adjacency items and the utility information. Two strategies for raising the threshold and one pruning strategy are put forward. The experimental results show that TUPP has better efficiency.

The remaining of the paper is organized as follows. Section 2 describes the definitions. Section 3 presents TUPP algorithm. Section 4 is experiment analysis. The paper is concluded in Section 5.

2. Basic Definitions.

When a system executes a task, it traverses a trace in the code to perform the requested functions. The trace identifies the methods called by the program of the execution which is the dynamic call graph G. The node of G has an utility which is the importance of the node. If we traverse G in Depth-First way, all function call paths from the root to leaf will be obtained. These paths produce a database D $= \{S_1, S_2, \ldots, S_n\}$ which is a set of sequences.

Definition 2.1. *The utility of node i in dynamic call graph G, is denoted as $u(i, G)$. The utility value presents the importance of function i in the software dynamic call graph G. It can be defined as follows.*

$$u(i, G) = w_1 \times deg(i) + w_2 \times b(i). \tag{1}$$

The $deg(i)$ is the degree about node i. The importance of position about node i, denoted as $b(i)$, is the result that all paths number through the node i divides the total paths number of graph G. The entropy weight of the two properties are w_1 and w_2. It can be calculated as follows. Assuming we have m properties and n objects of the evaluation, it can be presented by a evaluation matrix R in the view of the combination

of quantitative and qualitative.

$$R = \begin{vmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & & \vdots \\ r_{m1} & r_{m2} & \cdots & r_{mn} \end{vmatrix}. \tag{2}$$

Assume graph G has n nodes, so R is a two rows and n columns matrix. By matrix R we can define the entropy of the ith property H_i. Furthermore we can define the entropy weight of the ith property w_i.

$$H_i = -k\sum_{j=1}^{n} f_{ij}\ln f_{ij}. \left(f_{ij} = \frac{r_{ij}}{\sum\limits_{i=1}^{n} r_{ij}} \text{ and } k = \frac{1}{\ln n} \right) \qquad w_i = \frac{1 - H_i}{m - \sum\limits_{i=1}^{m} H_i}. \tag{3}$$

Definition 2.2. *Software Function Call Path (SFCP). Software Function Call Path is a sequence of consecutive nodes from the root node to leaf node of a dynamic call graph. It can be expressed as* $SFCP = < f_{root}, f_2, \ldots, f_{leaf} >$, *in which each element* f_k *means the* k*-th node in the SFCP* ($k \geq 1$, k *is an integer) and there exists call relationship between* f_k *and* f_{k+1}.

Definition 2.3. *Path pattern* $P1 =< e_1, e_2, \ldots, e_n >$ *is considered as a subsequence of software function call path* $P2 =< f_1, f_2, \ldots, f_m >$ *if there exist integers* $1 \leq i_1 < i_2 < i_3 < i_4 \cdots < i_n \leq m$, *where* $e_1 = f_{i_1}$, $e_2 = f_{i_2}, \ldots, e_n = f_{i_n}$.

Remark 2.1. *It should be noted that the path pattern in Definition 2.3 satisfies that items appearing in a software function call path containing the path pattern must be adjacent with respect to the underlying order as defined in the path pattern. A path pattern* $\alpha =< \alpha_1, \alpha_2, \ldots, \alpha_n >$ *is called a sub path pattern of another path pattern* $\beta =< \beta_1, \beta_2, \ldots, \beta_m >$ *if there exist integers* $1 \leq i_1 < i_2 < i_3 < i_4 \cdots < i_n \leq n \leq m$, *where* $\alpha_1 = \beta_{i_1}$, $\alpha_2 = \beta_{i_2}$, $\ldots$, $\alpha_n = \beta_{i_n}$. *We express this relation as* $\alpha \subseteq \beta$.

Definition 2.4. *The utility of software function call path* S_i, *denoted as* $util(S_i)$, *is the sum of the utilities of all the items in* S_i *and the total utility of* D *is the sum of the utilities of all the paths in* D.

Definition 2.5. *Given a path pattern* X *and a software function call path (or path pattern)* S *with* $X \subseteq S$, *the set of all the items after* X *in* S *is denoted as* S/X. *If* X' *is an extension of* X, $(X' - X) = (X'/X)$.

Definition 2.6. *For a path pattern* $X =< i_1, i_2, \ldots, i_n >$ ($X \subseteq S_i$ *and* $|X|$ *represents the length of* X), *its utility in a software function call path* S_i *denoted as* $u(X, S_i)$ *and the utility of* X *denoted as* $u(X)$ *are defined respectively as follows.*

$$u(X, S_i) = \sum_{X \subseteq S_i \wedge i \in X} u(i, S_i). \qquad u(X) = \sum\nolimits_{S_i \in D \wedge X \subseteq S_i} u(X, S_i). \tag{4}$$

Definition 2.7. *PUAList. Each element in the PUAList of path pattern* X *contains four fields that are named sid, aitem, util and rutil, representing a software function call path* S_i *containing* X, *adjacent item of* X, *the utility of* X *and the remaining utility of* X *in* S_i *respectively.*

Lemma 2.1. *Given the PUAList of path pattern* X, *if the sum of all the utils and rutils in the PUAList is less than a given* ε *(The least utility of the top-k high utility path patterns), any extension* X' *of* X *will not be included in top-k high utility path patterns.*

Proof: Suppose $id(S)$ denotes the sid of S, $X.sids$ denotes the sid set in the PUAList of X, and $X'.sids$ that is in X', and then: For $\forall$ software function call path $S \supseteq X'$

$$\because X' \text{ is an extension of } X \Rightarrow (X' - X) = (X'/X)$$
$$X \subset X' \subseteq S \Rightarrow (X'/X) \subseteq (S/X)$$
$$\begin{aligned}\therefore u(X', S) &= u(X, S) + u((X' - X), S) \\ &= u(X, S) + \sum_{i \epsilon (X'/X)} u(i, S) \\ &\le u(X, S) + \sum_{i \epsilon (S/X)} u(i, S) \\ &= u(X, S) + rutil(X, S)\end{aligned}$$
$$\because X \subset X' \Rightarrow X'.sids \subseteq X.sids$$
$$\begin{aligned}\therefore \quad u(X') &= \sum_{id(S) \epsilon X'.sids} u(X', S) \\ &\le \sum_{id(S) \epsilon X'.sids} (u(X, S) + rutil(X, S)) \\ &\le \sum_{id(S) \epsilon X.sids} (u(X, S) + rutil(X, S)) < \varepsilon\end{aligned}$$ □

Definition 2.8. *Item extension order. Given a sequence s and two items a, b. Let $s + a$ and $s + b$ be the sequences of a and b extended to s respectively, where $s + a \neq s + b$. We say a is prior to b, denoted as $a > b$, if the following conditions are true:*

$$\sum_{s \subset S \wedge s.aitem = a} rutil(s, S) > \sum_{s \subset S \wedge s.aitem = b} rutil(s, S).$$

3. **The TUPP Algorithm.**

In this section, we specify and present an efficient algorithm TUPP for mining top-k high utility path patterns. Instead of using a user specified minimum utility, TUPP engages a structure named TUPPList to maintain the top-k high utility path patterns on-the-fly. TUPPList is a fixed-size sorted list which is used to maintain the top-k high utility path patterns dynamically. In TUPPList, a minimum utility ε is set to prune the unpromising candidates in the mining process. In the previous algorithms, the TUPPList is empty and ε is set to 0 initially. Although the previous algorithms correctly extract the top-k high utility patterns, they traverse too many invalid pattern candidates since the minimum threshold starts from 0. This directly degrades the performance of the mining task. To overcome this problem, we propose an effective strategy for raising the minimum utility threshold.

Strategy 1: Pre-insertion. The pre-insertion strategy inserts the utilities of the 1-length patterns to the TUPPList before the mining process.

After the software function call sequences are successfully stored in the memory, it needs to calculate the utility of each sequence. In this phase, we use a hash table to record the utility of every distinct item in the sequences. The pre-insertion strategy effectively raises the minimum threshold to a reasonable level before mining, and prevents from generating unpromising candidates. The algorithm TUPP follows a pattern-growth method to mine the expected patterns. What item extension order should we adopt? In threshold-based high utility pattern mining, there is no such concern. However, in the top-k framework, the order of extending items does matter. Since ε is dependent on the candidates inside the TUPPList, we should put the high utility candidates to TUPPList as soon as possible so that ε increases to ε' shortly. Now we present item extension order strategy.

Strategy 2: Item extension order. Given a pattern P, and the adjacent items that can be extended to P are $e_1, e_2, \ldots, e_n$. Then $e_{k_1}, e_{k_2}, \ldots, e_{k_n}$ is the order to be extended to P, where $e_{k_1} > e_{k_2} > \cdots > e_{k_n}$.

Algorithm 1: TUPP Algorithm

Input: software dynamic call graph G, k
Output: Top-k high utility path patterns
1. obtain all paths by traversing the graph G in DFS way and store these paths in D.
2. calculate the utility for every node of G by Equation (1), and attach utility to every item of paths in database D.
3. calculate utility of different items in D, and put top-k highest utility items to TUPPList.// strategy 1
4. put the items whose sum of utils and rutils value is no less than ε to PTUPP1
5. PUAL1s ← construct the PUALists for all potential high utility l-length patterns of PTUPP1
6. sort PTUPP1 according to strategy 2
7. call procedure **TUPP-Gen(PTUPP1)**
8. **return** TUPPList

Procedure: TUPP-Gen
Input: PTUPP //the set of potential high utility path patterns
9. **for** each element Ex ∈ PTUPP **do**
10. **if** (u(Ex) > TUPPList.ε) **then**
11. Ex → TUPPList and remove the pattern with the least utility of TUPPList
12. **if** (sum utils and rutils of Ex > TUPPList.ε) **then** //the pruning strategy
13. CanItems ← sort the items of Ex.PUAList.aitem whose sum of utils and rutils exceed TUPPList.ε //the strategy 2
14. **for** each element Ey ∈ CanItems **and** Ey ∈ PTUPP1 **do**
15. exPUALs = exUNLs + **Combine**(Ex, Ey)
16. TUPP-Gen(exPUALs)

Procedure: Combine
Input: pattern Px; pattern y
17. **for** each element x ∈ Px.PUAList **do**
18. **if**(y.PUAList.sids.contains(x.sid)) **then**
19. xy = <x.sid, y.PUAList.aitem, x.util + y.util, y.rutil>
20. append xy to Pxy.PUAList

Exploiting the two strategies and the pruning strategy presented in Lemma 2.1, TUPP is described as Algorithm 1. In line 2, calculate utility for all nodes of G by Equation (1). Lines 5 to 8 describe the process of calculating the potential high utility of 1-length patterns. We put these potential 1-length path patterns in the set PTUPP1 and construct PUALists for them. Then a sub-procedure TUPP-Gen is devised to mine top-k high utility path patterns efficiently from D. Lines 9 to 16 present the sub-procedure TUPP-Gen which is a recursive process. For each pattern Ex in PTUPP, if the utility of Ex exceed the least utility in TUPPList, then insert the pattern to TUPPList and remove the pattern with the least utility. Only when the sum of *utils* and *rutils* of pattern exceeds the least utility of TUPPList, can it be extended further for possibly producing the top-k high utility path patterns.

Example 3.1. *For example, Figure 1 represents the path sequences. Assume k is 3 and the utilities of items a, b, c, d, e and f are 3, 8, 12, 12, 10 and 3. At beginning, the top-3 path patterns of TUPPList is* $\{c, d, e\}$. *The sum of utils and rutils of the items is 34, 45, 20, 34, 13 and 8 respectively. As the sum of util and rutil of f is less than 10, the*

PTUPP1 is $\{b, a, d, c, e\}$ *in order. Next, we take b for example to illustrate the extending process. The aitems in PUAList of b is* $\{c, d\}$ *and the sum of utils and rutils is 5 and 40 respectively. As the least utility in TUPPList is 10 currently, the item c cannot be extended to b. The patterns in bold in Figure 2 are unnecessary to be extended. The utility of pattern* $< bd >$ *is 18. Therefore, put* $< bd >$ *into TUPPList and remove pattern* $< e >$*. All the other patterns can be similarly processed. The results are* $\{< bd >, < bdc >, < abdc >\}$.

sid	software function call sequence with utility
s1	<a(1), b(2), c(3)>
s2	<a(1), b(2), d(4), c(3)>
s3	<b(2), d(4), e(5), c(3)>
s4	<a(1), b(2), d(4), c(3), f(3),e(5)>

Figure 1. The database

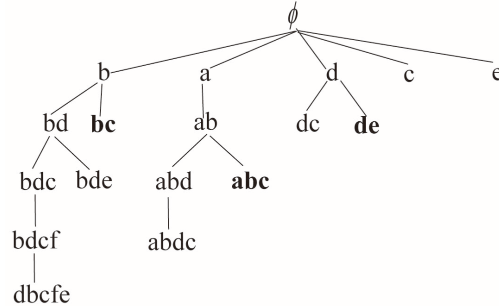

Figure 2. Extending process

4. **Experiment.** In this section, we evaluate the performance of TUPP on a variety of datasets. Since there is no algorithm that can solve the top-k high utility path patterns mining, and it is not easy to upgrade the existing method such as [7] either, we thus compare the four algorithms. They are TUPP, TUPPNaive which is a baseline approach, TUPPS1 incorporating strategy 1 and TUPPS2 incorporating strategy 2. These algorithms were implemented in Java and the experiments were performed on 64 bit Windows 7 ultimate, Xeon CPU E5-2603 @1.80GHz, 8G Memory. The synthetic datasets were generated by IBM data generator. The real database is mushroom which can be downloaded from FIMI Repository [13]. The weight distributions of items in datasets are generated from Gauss distribution ($\mu = 5$, $\sigma = 1.5$) [14]. We abbreviate the average length of sequences as Ave-L.

Running Time. We first ran four algorithms on datasets mushroom and T40I10D20K varying k to assess the influence of k on execution time and candidates number. As it can be seen in Figure 3 and Figure 4, we can find that the running time of the four algorithms gets increased by increasing the k. Besides, the gap between TUPP and TUPPNaive increases with the increasing of k. The results also show that TUPP, TUPPS1 and TUPPS2 are generally faster than TUPPNaive. That indicates the proposed three optimization measures, including pruning strategy, sorting and pre insertion, are effective for top-k pattern mining. From the experimental results given in Figure 5 and Figure 6, it is also observed that the number of candidate patterns is increasing gradually by increasing the k. We understand that the number of candidate patterns produced by TUPP is greatly reduced compared to the TUPPNaive and TUPPS1. This benefits from superior pruning and sorting strategies of TUPP, which can raise the threshold more quickly and prune more unpromising patterns.

Scalability. We studied the scalability of TUPP algorithm on running time by varying the number of sequences in the dataset and the average length of sequences. Figure 7 and Figure 9 show the time cost and candidates by varying the number of sequences from 2000 to 10000 when k is 50. Figure 8 and Figure 10 show the trend of the running time and the candidates of four algorithms with respect to different average length of sequences varying from 6 to 14 based on $|D| = 20000$ when k is 100. From the results, it is observed that the scalability of TUPP outperforms the three algorithms. This is because TUPPNaive starts the mining from 0 while TUPP does not. The sorting strategy more directly prunes unpromising branches than the pre-insertion. The sorting strategy always traverses the

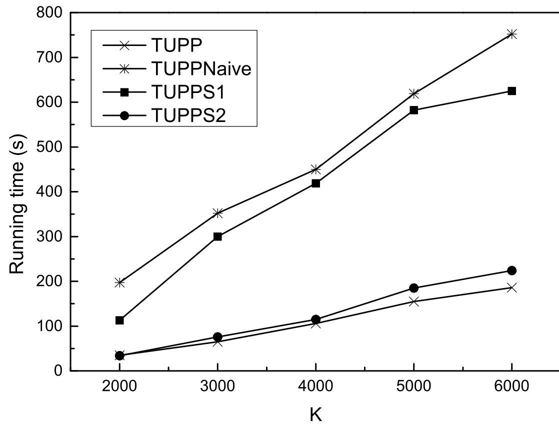

Figure 3. Time cost varying K on mushroom

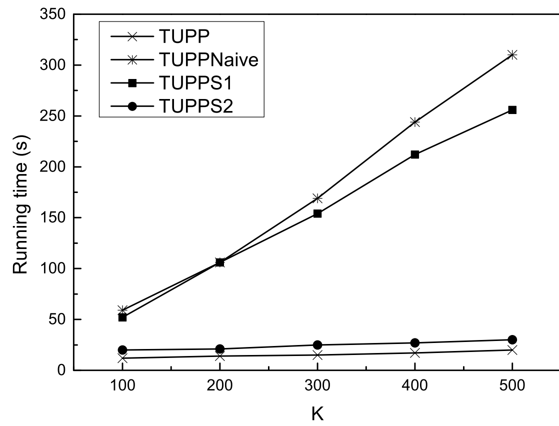

Figure 4. Time cost varying K on T40I10D20K

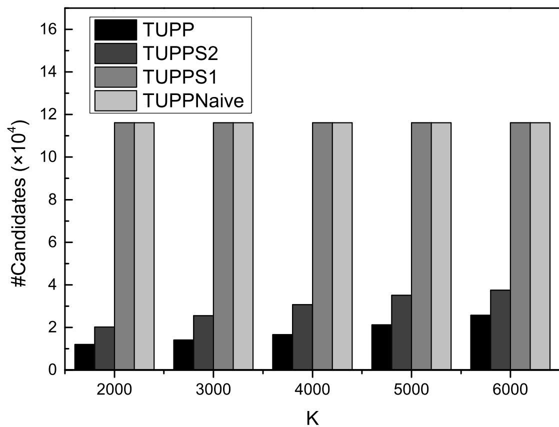

Figure 5. Candidates varying K on mushroom

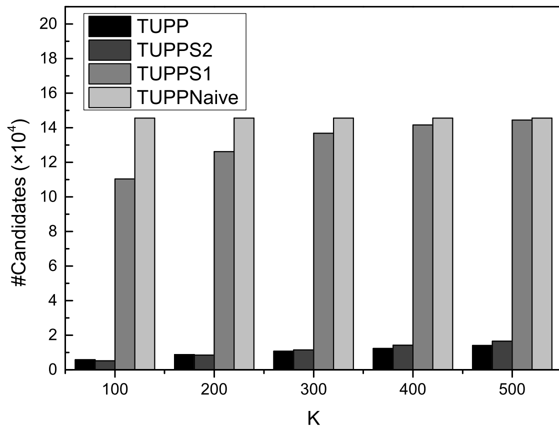

Figure 6. Candidates varying K on T40I10D20K

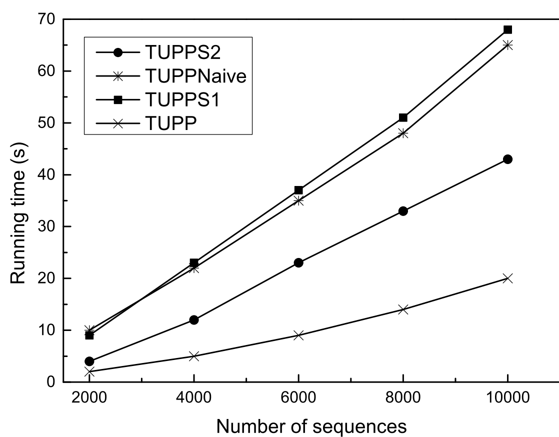

Figure 7. Time cost under different database sizes

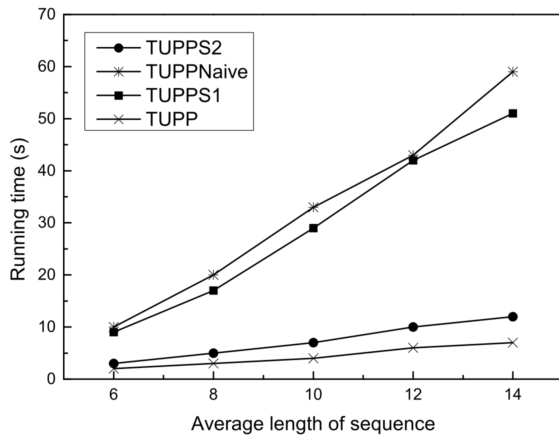

Figure 8. Time cost under different Ave-L

higher estimated utility candidates first. This guarantees ε raising to ε' shortly. So when the datasize is large, sorting is better than pre-insertion.

5. **Conclusions.** In this paper, we have proposed an efficient algorithm named TUPP for mining top-k high utility path patterns from software dynamic call graph. We have defined the utility of functions and developed a new pruning strategy for effectively filtering the

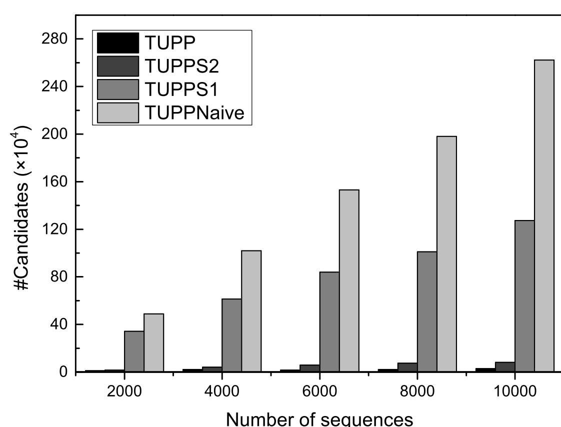

Figure 9. Candidates under different database sizes

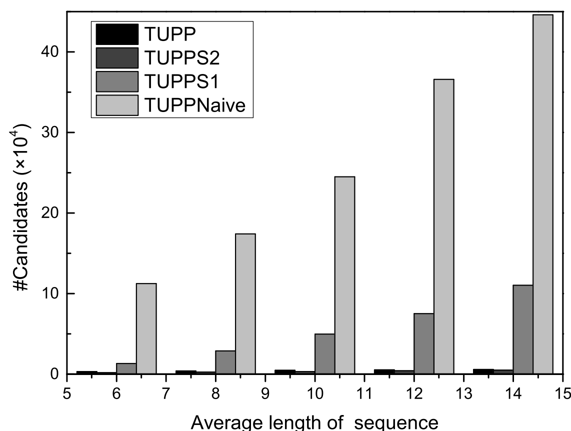

Figure 10. Candidates under different Ave-L

unpromising candidates. Moreover, a pre-insertion and a sorting strategy have been introduced to raise the minimum utility threshold. From the experiments, we can know that the mining performance is enhanced significantly since both the search space and the number of candidates are effectively reduced by the proposed strategies. In the future, we plan to apply our proposed algorithm in some real softwares.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61170190, No. 61472341 and the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2013203324, No. F2014203152 and No. F2015203326.

REFERENCES

[1] T. Xie, S. Thummalapenta, D. Lo et al., Data mining for software engineering, *Computer*, vol.42, no.8, pp.55-62, 2009.
[2] J. F. Bowring, J. M. Rehg and M. J. Harrold, Active learning for automatic classification of software behavior, *ISSTA*, pp.195-205, 2004.
[3] 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.
[4] G. Pyun and U. Yun, Mining top-k frequent patterns with combination reducing techniques, *Appl. Intell.*, vol.41, no.1, pp.76-98, 2014.
[5] Q. Huynh-Thi-Le, T. Le, B. Vo et al., An efficient and effective algorithm for mining top-rank-k frequent patterns, *Expert Systems with Applications*, pp.156-164, 2015.
[6] C.-W. Wu, B.-E. Shie, V. S. Tseng and P. S. Yu, Mining top-K high utility itemsets, *KDD*, pp.78-86, 2012.
[7] J. Yin, Z. Zheng, L. Cao, Y. Song and W. Wei, Efficiently mining top-k high utility sequential patterns, *ICDM*, pp.1259-1264, 2013.
[8] H. Ryang and U. Yun, Top-k high utility pattern mining with effective threshold raising strategies, *Knowledge-Based Systems*, pp.109-126, 2014.
[9] L. Zhou, Y. Liu, J. Wang et al., Utility-based web path traversal pattern mining, *International Conference on Data Mining Workshops*, pp.373-380, 2007.
[10] C. F. Ahmed, S. K. Tanbeer, B. Jeong et al., Efficient mining of utility-based web path traversal patterns, *ICACT*, pp.2215-2218, 2009.
[11] M. Thilagu and R. Nadarajan, Efficiently mining of effective web traversal patterns with average utility, *ICCCS*, vol.1, no.4, pp.444-451, 2012.
[12] C. F. Ahmed, S. K. Tanbeer and B. Jeong, A framework for mining high utility web access sequences, *IETE Technical Review*, vol.28, no.1, pp.3-16, 2011.
[13] *Frequent Itemset Mining Dataset Repository*, http://fimi.ua.ac.be/, 2012.
[14] R. Geng, X. Dong and W. Xu, Efficient algorithm for mining weighted sequential patterns based on graph traversals, *Journal of Control and Decision*, vol.24, no.5, pp.663-668, 2009.