ICIC Express Letters
Volume **11**, Number **2**, February **2017**
 ISSN 1881-803X
pp. **413–420**

A NOVEL ALGORITHM FOR MINING TOP-K HIGH UTILITY TRAVERSAL PATTERNS FROM SOFTWARE DYNAMIC EXECUTING GRAPH

Changwu Wang[1,2], Cui Xu[1,2], Hao Wang[1,2], Liwen Yue[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, Hebei Ave., Qinhuangdao 066004, P. R. China
{ cwwang; ylw; jdren }@ysu.edu.cn; xucui_ys@sina.com

Received July 2016; accepted October 2016

Abstract. *It is a significant work to analyze the software behavior. Finding high utility patterns from software executing network is useful to comprehend the software. Compared with frequent patterns mining, the concept of utility provides more reliable and available knowledge. However, it is difficult for users to determine a proper minimum threshold to predict the exact number of patterns mined by the threshold. To address the issue, a novel algorithm for mining top-k high utility traversal patterns (named THTP-Miner) from software dynamic executing graphs is proposed, where k is the desired number of patterns. In THTP-Miner, software executing traversal sequences are extracted from software executing graphs firstly. Secondly, a structure local information list of pattern is designed to store the useful information. Thirdly, three effective strategies are introduced to resolve the issue of efficiency, including two methods for raising the threshold and one pruning for filtering unpromising patterns. Comprehensive experimental results show that THTP-Miner has excellent performance.*
Keywords: High utility traversal patterns mining, Top-k traversal patterns, Software executing graph

1. **Introduction.** Software features mining can discover meaningful information and knowledge from complicated software systems and its executing process. Software searching [1] modeled a software package as a network, which can reproduce the particular features exhibited by real-world software packages. A dynamic software executing graph is obtained when a software system performs a task, which is composed of vertices and directed edges. Every software executing path acquired from software executing graphs can be considered as a sequence, which has benefit for understanding the software execution and software security [2,3]. Therefore, how to discover the desirable sequences from the software execution path is an important work.

Frequent sequential pattern mining [4-6] is to find a set of sequences whose support is no less than a minimum support. However, it is often difficult for the users to specify a minimum utility threshold. To handle this, Pyun and Yun [7] proposed the concept of top-k patterns to extract the pattern of the highest frequency. It follows that finding only top-k patterns is more attractive and intuitive than producing all the patterns whose support is above a given threshold. Nevertheless, the traditional frequent patterns consider only binary frequency values of items and do not disclose the impact of items in databases. Therefore, the top-k high utility pattern mining emerges as an important topic in data mining. Yin et al. [8] proposed a novel algorithm to identify top-k high utility sequential patterns without threshold. In addition, Ryang and Yun [9] raised an efficient algorithm for mining top-k high utility patterns with highly decreased candidates.

Software behavior mining plays a significant role in realizing the failure prediction of software and the features of software system. Li et al. [10] presented an approach to detect software failure by using the pattern position distribution as features. Note that the items in software function executing sequences are ordered and continuous, and the traditional algorithms for mining sequential patterns are not applicable. Therefore, a two-phase utility mining method [11] is proposed to discover high utility path traversal patterns. In order to reduce the number of candidates, Ahmed et al. [12] designed an algorithm for utility-based web path traversal pattern mining. However, these methods consider only forward references and are not applicable for incremental mining. A novel framework [13] was presented to handle both forward and backward references, which can avoid the level-wise candidate generation-and-test methodology.

In this paper, combining with the advantages of utility and path pattern mining, an effective algorithm (named THTP-Miner) is proposed to mine top-k high utility traversal patterns from software executing graph. In order to analyze software executing sequences and rules, a novel structure, local information list of pattern, is designed to store both the adjacency item and the utility information. Moreover, three strategies are presented to connect potential high utility traversal patterns as early as possible, which are good for discovering desirable sequences efficiently. Based on the three strategies, the novel algorithm can be used to analyze software executing paths and software security. Experiments are executed to verify that THTP-Miner has better efficiency.

The remaining is organized as follows. Section 2 describes the definitions. Section 3 proposes the algorithm THTP-Miner. Section 4 shows the experimental results. Conclusions are presented in Section 5.

2. **Basic Definitions.** In this paper, we discuss the software executing traces from function perspective. In the dynamic execution process of software, it traverses a path to perform all the requests of functions. This path is a branch of the software executing graph which represents the call relationships among different functions in a system.

Definition 2.1. *Software Executing Graph. Software executing graph called SEG is a directed graph made up of vertices and directed edges, i.e., G is a 3-tuples (V_G, E_G, Γ) where V_G is a set of vertices and $E_G \subseteq V_G \times V_G$ is a set of directed edges. Γ is a function that maps labels to vertices and edges.*

The vertices and edges represent the functions and their call relationships respectively.

Definition 2.2. *Software Executing Traversal Sequence. Software executing traversal sequence named SETS is a branch of the software executing graph, which is obtained from root to a leaf. Let $F = \{f_1, f_2, \ldots, f_n\}$ be a set of functions. Software executing traversal sequence is an ordered list of functions, denoted by $< f_{root}, f_2, \ldots, f_m, f_{leaf} >$, where f_j $(2 \leq j \leq m \leq n)$ in the SETS signifies the jth function called by the sequence. Besides, f_{root} and f_{leaf} represent the first and the last called function respectively. All of these sequences form software executing traversal sequence database, which is denoted as TSD.*

Definition 2.3. *The Utility of Function. The utility of function f in a software executing traversal sequence S is denoted as $u(f, S)$, which reflects the importance or preference of function f in the sequence S.*

Definition 2.4. *The Utility of Pattern. The utility of a pattern P in a sequence is denoted as $u(P, S)$ and defined as $u(P, S) = \Sigma_{f \in P} u(f, S)$. Besides, the utility of a pattern P in a database TSD is denoted as $u(P)$ and defined as $u(P) = \Sigma_{S \in t(P)} u(P, S)$, where $t(P)$ is a set of sequences containing pattern P in the database TSD.*

Definition 2.5. *Top-k High Utility Traversal Patterns. A pattern P is called a top-k high utility traversal pattern if there are less than k sequences whose utilities are no less than*

$u(P)$. *The optimal minimum utility is denoted and defined as* $\sigma^* = \min\{u(P)|P \in \Omega\}$, *where* Ω *means the set of top-k high utility traversal patterns.*

Definition 2.6. *Local Information List of Pattern. A local information list of pattern* P *is a 3-tuples (utility, r-utility, sid) for each sequence* S_{sid} *containing pattern* P *with the same next element. The utility of a triple is the total utility of the pattern* P *in the sequence* S_{sid}, *i.e.,* $u(P, S_{sid})$. S_{sid} *is a sequence with its order sid in the database. The r-utility of a triple is defined as* $\Sigma_{f \in S_{sid} \wedge f \succ P} u(f, S_{sid})$, *where* $f \succ P$ *denotes that* f *is listed after* j *for every* $j \in P$. *The next element indicates an item which is adjacent to the last item of the current pattern* P *in the sequence* S_{sid}.

Besides, the local information list is marked by {pattern, next} which contains the next element of the pattern. The local information list of pattern is shown in Figure 1.

{pattern, next}

utility	r-utility	sid

FIGURE 1. Local information list of pattern

Definition 2.7. *Connection of Traversal Patterns. Let the connection of* Px *be the pattern that can be got by appending an item* y *to the pattern* Px, *where the item* y *must be adjacent to the last item* x *in a sequence.*

Definition 2.8. *Potential High Utility Traversal Patterns. A local potential high utility traversal pattern is that the sum of utility and r-utility in the local information lists is no less than the current threshold* σ^*. *If the sum is no more than the threshold, the pattern is called unpromising.*

3. Mining Top-k High Utility Traversal Patterns.

In this section, the algorithm for mining top-k high utility traversal patterns from software executing graph is proposed, in which three strategies are applied to improving the threshold quickly.

3.1. Strategy 1: Pre-insertion.

The pre-insertion strategy inserts the utilities of both the 1-sequences and s-sequences into the TKSList before the iterative process of mining.

A list TKSList is a fixed-size sorted list, which is used to maintain the top-k high utility sequential patterns dynamically, and a minimum utility σ^* is set to prune unpromising candidates. Initially, TKSList is empty and σ^* is 0. Once the k sequences are found, σ^* is raised to the least utility in the TKSList. After that, a candidate satisfying the current threshold is inserted into TKSList, then the candidate with least utility value in the list is pruned, and thus σ^* will be raised to the least utility in the updated list. In the meanwhile, a hash table T is used to record the real utility of every distinct item. All of these items are actually 1-sequences and they are inserted into TKSList.

3.2. Strategy 2: Sorting connection order.

The sorting strategy is used to order the potential high utility sequential patterns by its sum of utility and r-utility and raise the threshold shortly.

It follows from definitions that the local potential high utility sequences should be calculated quite early, which are more likely to improve the current threshold. Thus, all of the patterns are ordered by the sum of utility and r-utility in the local information lists. This strategy effectively identifies local potential high utility patterns which can be connected and prior to those low utility candidates.

3.3. **Strategy 3: Filtering unpromising patterns.** The strategy filters unpromising patterns in the process of connection by checking an upper bound of utility in local information lists of patterns.

Assume that the pattern P' is a connection of pattern P, and $t(P)$ is a set of sequences containing P. Because of $P \subseteq P'$, we can know that $t(P') \subseteq t(P)$. So we can find an upper bound of utility with respect to any connection of the pattern P.

Example 3.1. *Table 1 represents a database TSD,* $F = \{a, b, c, d, e, f, g, h\}$ *is a set of items appearing in TSD and* $k = 3$*. The pair* $(f, u(f, S))$ *represents the utility of function* f *in the sequence* S*. The SU is the utility of the sequence. Originally, TKSList is* $\{12, 11, 10\}$ *with the corresponding patterns* a*,* S_2*,* S_5 *by using Strategy 1 and* $\sigma^* = 10$*. The local information lists of the items are built to preserve available information. Secondly, the sum of utility and r-utility of the items in* F *with the same next element is calculated and the values are* $\{9, 21, 6, 14, 13, 4, 4, 3, 4, 3, 4, 1, 1, 1\}$ *respectively. Then, these lists are ordered to connect local potential high utility patterns. Finally, the ordered local information lists satisfying the threshold are showed in Figure 2. For instance,* $\{a, c\}$ *means the item of a is the highest potential high utility item with the next element c.*

TABLE 1. An example of TSD

S_{sid}	Sequences	SU
S_1	(a,2)(b,3)(f,3)(g,1)	9
S_2	(a,5)(c,3)(e,3)	11
S_3	(b,2)(f,4)(h,1)	7
S_4	(c,4)	4
S_5	(a,3)(c,2)(e,4)(f,1)	10
S_6	(a,2)(d,4)	6

{a, c}

utility	r-utility	sid
5	6	2
3	7	5

(a) the pattern {a} with next element c

{b, f}

utility	r-utility	sid
3	4	1
2	5	3

(b) the pattern {b} with next element f

{c, e}

utility	r-utility	sid
3	3	2
2	5	5

(c) the pattern {c} with next element e

FIGURE 2. Ordered local information lists satisfying the threshold

3.4. **The THTP-Miner algorithm.** The algorithm of mining top-k high utility traversal patterns denoted as THTP-Miner is described in Algorithm 1. The input parameters are software dynamic executing graphs, and the database TSD is obtained by traversing the software executing graphs. The first phase Pre_Insertion is called to raise the threshold by the utilities of both 1-sequences and s-sequences. The second phase is Connection process which can connect all potential high utility patterns iteratively.

In Pre_Insertion procedure, the TKSList is used to store the top-k high utility candidates dynamically. If the size of TKSList is less than k, the new potential high utility pattern is inserted directly into list. Another case is that the size is equal to k, and the threshold is set to the least utility of TKSList immediately. After that, a new candidate satisfying σ^* is inserted into list and the least utility candidate will be eliminated. Meanwhile, the threshold is updated.

The procedure Connection follows a pattern-growth method to mine the top-k high utility traversal patterns. These local information lists are ordered by the sum of utility

and r-utility. For every pattern P_i in Ordered_Local_Lists, if the sum of utility is no less than σ^*, we will call the Pre_Insertion to put the pattern into TKSList directly. If the pattern P_i is promising, the connection of P_i with its next element P_n is implemented to construct a new local information list. After the lists of patterns starting with P_i are obtained, the Connection process is called with the parameter lists iteratively.

Algorithm 1: The THTP-Miner algorithm

Input: software executing graphs SEG
Output: a set of top-k high utility traversal patterns
1. Generate TSD by traversing the Software Executing Graphs in the DFS way
2. itemUtilityMap ← calculate every different item and its total utilities in TSD
3. itemList ← store all different items in TSD
4. TKSList ← initialize to an empty list and set the threshold σ^* to 0
5. k ← the desired number of high utility traversal patterns
6. **for** every sequence S in TSD **do**
7. Call $Pre\_Insertion(S)$ to improve σ^* by the s-sequence
8. **for** every every item f in itemUtilityMap **do**
9. Call $Pre\_Insertion(f)$ to improve σ^* by the 1-sequence
10. **for** every item f in itemList **do**
11. New_Initial_List ← construct an initial Local_Information_List
12. Local_Information_Lists ← Local_Information_Lists ∪ New_Initial_List
13. Call $Connection(Local\_Information\_Lists)$ to connect potential patterns iteratively

Procedure: Pre_Insertion
Input: a sequential pattern S
Output: an ordered list TKSList containing top-k high utility patterns
14. **if** TKSList.size < k **then**
15. insert $u(S)$ into TKSList
16. **if** TKSList.size == k **then**
17. set the threshold σ^* to the least utility of patterns in TKSList
18. **if** $u(S) > \sigma^*$ **then**
19. delete the least utility of TKSList and insert the new utility $u(S)$ into TKSList
20. update σ^* to the least utility of the new TKSList

Procedure: Connection
Input: Local_Information_Lists
Output: a set of all high utility traversal patterns
21. Calculate the sum of utility and r-utility of Local_Information_Lists
22. Ordered_Local_Lists ← order these lists by the sum of utility and r-utility
23. **for** every list of P_i in Ordered_Local_Lists **do**
24. **if** its sum of utility $\geq \sigma^*$ **then**
25. Call $Pre\_Insertion(P_i)$ //by strategy 1
26. **if** its sum of utility and r-utility $\geq \sigma^*$ //by strategy 3 **then**
27. **for** every list of P_n in Local_Information_Lists **do**
28. **if** P_n is the next element of P_i **then**
29. $\{P_iP_n\}$ ← connect P_i and P_n
30. New_list ← construct a new Local_Information_List of $\{P_iP_n\}$
31. Lists ← Lists ∪ New_list
32. Call $Connection(Lists)$

4. **Experimental Evaluation.** In this section, the proposed method THTP-Miner is evaluated on a variety of datasets. We compare THTP-Miner with THTP+PS, THTP+PF, THTP+F and EUWPTM [12]. THTP+PS and THTP+PF are two approaches without filtering and sorting strategies respectively. THTP+F is a method only with strategy 3. Note that, in the experiments, we use optimal parameters for the utility-based web path traversal patterns mining algorithm (EUWPTM), in order to mine the same number of patterns as the case of THTP-Miner. All the algorithms were implemented in Java and these experiments were performed on 64 bit Windows 7 ultimate, Xeon CPU E5-2603 @1.80GHz, 8G Memory. Two datasets, Mushroom and BMS, were used in the experiments, which can be downloaded from SPMF [14].

4.1. **Runtime.** Figure 3 and Figure 4 show the total run time of each method with the k varied. The execution time of EUWPTM which needs to calculate projected databases iteratively is much worse than others, while THTP-Miner considers continuity characteristics of software executing traversal sequences, only when the utility of pattern connecting its next adjacency item exceeds the current threshold σ^*, we continue to traverse. Also, it can be observed that THTP-Miner can raise the threshold shortly than THTP+PF due to the strategy of sorting connection order, and thus the time is less than THTP+PF.

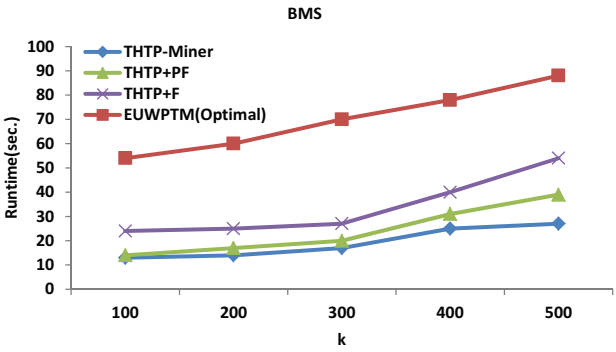

FIGURE 3. Runtime on BMS

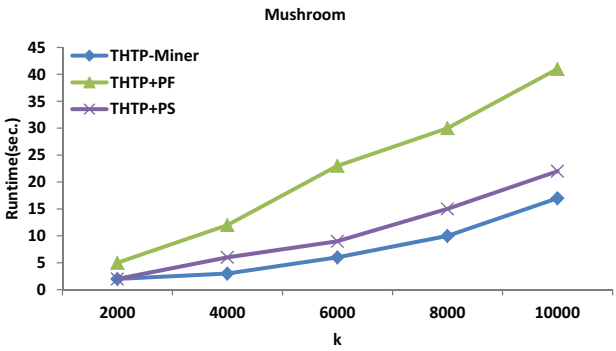

FIGURE 4. Runtime on Mushroom

4.2. **Memory usage.** Figure 5 and Figure 6 are the memory usages of these algorithms. Considering the performance results, EUWPTM consumes the most memory even though the threshold obtained by the corresponding k in top-k algorithm is optimal. It is caused by the fact that EUWPTM needs to store a great number of projected databases. The memory usage of THTP+F is also greatly large due to the absence of pre-insertion strategy and sorting strategy.

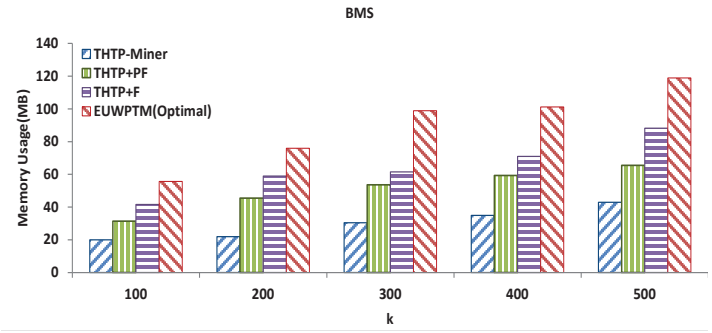

FIGURE 5. Memory usage on BMS

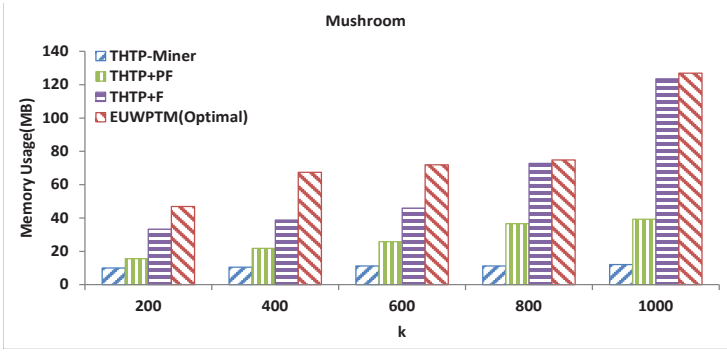

FIGURE 6. Memory usage on Mushroom

4.3. **Pruning ability.** Figure 7 and Figure 8 show the pruning ability of THTP-Miner. Because the threshold determined by the number of high utility patterns is optimal, the candidates generated by EUWPTM are the fewest. As shown in Figure 7, THTP+PF produces a smaller number of candidate patterns than THTP+F when the k is 100-300 on BMS. This is because THTP+F starts the mining from 0 while THTP+PF does not,

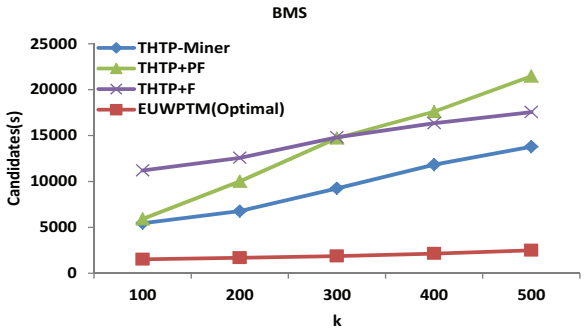

FIGURE 7. Candidates on BMS

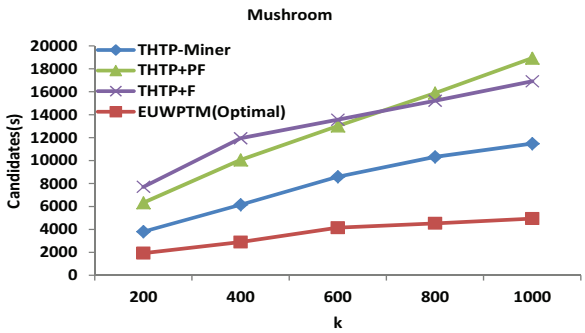

FIGURE 8. Candidates on Mushroom

and pre-insertion directly prunes certain unpromising patterns and raises the threshold σ^* shortly. Besides, we can observe that THTP-Miner can generate less candidates than THTP+PF and THTP+F. As a result, we can get the conclusion that THTP-Miner has stronger pruning ability than others.

It is worth noting that the accuracy rate of the algorithm THTP-Miner may be reduced by raising the threshold quickly in the mining process.

5. **Conclusions.** In this paper, we have proposed an efficient algorithm named THTP-Miner for mining top-k high utility traversal patterns which can be obtained by traversing software dynamic executing graphs in DFS way. The structure, local information list of pattern, is presented which can provide not only utility information about the pattern but also a next element for connecting promising patterns. Moreover, a pre-insertion strategy is proposed which can raise threshold quickly. Two strategies sorting connection order and filtering unpromising patterns are also developed to connect potential high utility patterns as early as possible so as to increase σ^* shortly. Experimental results indicated that the proposed strategies significantly reduced the considerable search space and decreased the number of generated candidates substantially. In the future, we are going to apply our algorithm in some more complex real software executing graphs.

Acknowledgement. This work is supported by the National Natural Science Foundation of China under Grant No. 61572420, No. 61472341 and the Natural Science Foundation of Hebei Province China under Grant No. F2013203324, No. F2014203152 and No. F2015203326. The authors are grateful to the valuable comments and suggestions of the reviewers.

REFERENCES

[1] J. Ma, D. Zeng and H. Zhao, Modeling the growth of complex software function dependency networks, *Information Systems Frontiers*, vol.14, no.2, pp.301-315, 2012.

[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] S. Chandra and R. A. Kan, Software security metric identification framework, *Proc. of the International Conference on Advances in Computing, Communication and Control*, pp.725-731, 2009.

[4] R. Agrawal and R. Srikant, Mining sequential patterns, *Proc. of IEEE International Conference on Data Engineering*, pp.3-14, 1995.

[5] J. Pei, J. Han, B. Mortazavi-Asl, Q. Chen, U. Dayal and M. C. Hsu, PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth, *Proc. of IEEE International Conference on Data Engineering*, pp.215-224, 2001.

[6] D.-Y. Chiu, Y.-H. Wu and A. L. P. Chen, Efficient frequent sequence mining by a dynamic strategy switching algorithm, *The VLDB Journal*, vol.18, no.1, pp.303-327, 2009.

[7] G. Pyun and U. Yun, Mining top-k frequent patterns with combination reducing techniques, *Appl. Intell.*, vol.41, no.1, pp.76-98, 2014.

[8] J. Yin, Z. Zheng, L. Cao, Y. Song and W. Wei, Efficiently mining top-k high utility sequential patterns, *ICDM*, pp.1259-1264, 2013.

[9] H. Ryang and U. Yun, Top-k high utility pattern mining with effective threshold raising strategies, *Knowledge-Based Systems*, pp.109-126, 2014.

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

[11] 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.

[12] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong et al., Efficient mining of utility-based web path traversal patterns, *The 11th International Conference on Advanced Communication Technology*, pp.2215-2218, 2009.

[13] C. F. Ahmed, S. K. Tanbeer and B.-S. Jeong, A Framework for mining high utility web access sequences, *IETE Technical Review*, vol.28, no.1, pp.3-16, 2011.

[14] *An Open-Source Data Mining Library*, http://www.philippe-fournier-viger.com/spmf/, 2008.