International Journal of Innovative
Computing, Information and Control
Volume 11, Number 2, April 2015

ICIC International ©2015 ISSN 1349-4198
pp. 541–553

GRAPH-MINE: A KEY BEHAVIOR PATH MINING ALGORITHM IN COMPLEX SOFTWARE EXECUTING NETWORK

Jiadong Ren[1,2], Weina Li[1,2], Yuzheng Wang[3] and Lianbo Zhou[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
jdren@ysu.edu.cn; wnli510@126.com; zlb-514@163.com

[3]International College
Beijing University of Posts and Telecommunications
No. 10, Xitucheng Road, Haidian District, Beijing 100876, P. R. China
yzwang@bupt.edu.cn

Received April 2014; revised August 2014

Abstract. *Security analysis of software has attracted researchers' great attention. And key paths in complex software executing network play an important role in analyzing software security. Mining key paths in complex software executing network has become a research hotspot. To address issues of path extension and low efficiency in mining key paths by previous algorithms, a novel approach called Graph-Mine is proposed to mine key paths in complex software executing network. In Graph-Mine algorithm, firstly complex software executing network graph SENG is defined, and functions and call relationship among functions are mapped to nodes and paths respectively. Secondly, Depth-First-Search strategy is adopted to transform complex software executing network graph SENG into software executing path SEP. Thirdly, Key-Path-Search is presented to mine key paths from software executing paths. During Key-Path-Search mining processes, according to continuity characteristics of software executing path, a new adjacency path extension strategy is designed to avoid inappropriate mined results and improve mining efficiency. Finally, prefix path and extended path are defined to mine key paths conveniently and efficiently by adjacency path extension strategy. Experimental results show that Graph-Mine is more efficient in finding key paths in complex software executing network.*

Keywords: Complex software network, Executing paths, Key paths, Adjacency path extension strategy

1. **Introduction.** More and more complex network characteristics are found in large software systems with the research on complex networks, and important functions and relationship among them are mapped to key nodes and paths respectively. If problems occur in key paths, software industry will suffer a great loss. In addition, software vulnerabilities or abnormalities may be found efficiently by analyzing key paths.

Complex network theory is used to study software executing processes, and many achievements have been obtained in this aspect. Valverde et al. [1] adopted complex network to analyze topology characteristics of software system. An undirected network was utilized to represent the structure of software systems, in which nodes represented classes and edges were inheritance and associated relationship among classes. Software systems expose small-world feature and follow scale-free degree distributions. Cai and Yin [2] put forward an evolving software mirror graph according to relationship among functions in dynamic executing processes. Software executing processes no longer expose

the small-world feature in the temporal sense. Further, degree distributions of software executing processes may follow a power law. However, they may also follow an exponential function or a piecewise power law. Ma et al. [3] modeled a software package as a network, with nodes representing functions in a package and edges representing dependencies among functions. He set out to develop a network growth model, explicitly imitating generally-advocated software development principals, such as divide-and-conquer, modularization, high intra-module cohesion, and low inter-module coupling. A new perspective was provided for future research on complex software network.

Graph mining is an important task in finding key paths in complex software executing network. Representative graph mining approaches include Apriori-based and FP-growth-based algorithms. Inokuchi et al. [4] proposed an Apriori-based algorithm called AGM to discover all frequent subgraphs. The algorithm works by generating candidate patterns and pruning infrequent patterns. However, a large number of candidate sets were generated and much time was consumed to test subgraph isomorphism. By extending Apriori, Inokuchi et al. [5] proposed an algorithm to mine frequent induced subgraphs. However, the approach also generated a large amount of candidates and wasted a lot of time to test graph isomorphism. In order to skip the candidate generation process, gSpan was presented by Yan and Han [6]. The method mapped each graph to a unique minimum DFS code and avoided generating candidate sets, but it also took much time to test graph isomorphism. For this reason, Shen and Yu [7] proposed an approach based on Markov Chain Monte Carlo sampling. An efficient neighboring pattern counting technique was proposed to greatly reduce time-consuming of testing subgraph isomorphism.

Because of graph isomorphism, mining frequent subgraph is much more difficult than mining frequent patterns from Graph sequences. Existing graph sequence mining approaches are applied to dynamic and static graphs. On one hand, researches have focused on finding time evolution law of graphs and taking changes of a graph over time into consideration. Borgwardt et al. [10] put forward a dynamic frequent subgraph mining algorithm to find all frequent patterns from a long graph sequence. Static pattern mining methods were introduced into dynamic graphs, especially taking the insertion and deletion of edges into account. Lahiri and Berger-Wolf [11] proposed an algorithm to find all frequent patterns that appeared periodically in dynamic social networks. The algorithm took imperfect periodicity into account and found periodic evolution law of social networks. A new measure, purity, was proposed for ranking mined subgraphs according to how perfectly periodic a subgraph is. Berlingerio et al. [12] put forward an algorithm GERM to mine all graph-evolution rules. All frequent patterns are mined from a long graph sequence. On the other hand, attention was paid to find frequent patterns in a graph, just considering order of nodes. Inokuchi and Washio [13] proposed an approach GTRACE to mine frequent patterns efficiently from graph sequences. It still needs substantial computation time to mine patterns from graph sequences containing long sequences. To improve efficiency, Inokuchi et al. [14] proposed GTRACE-RS to mine all frequent sequences from graph sequences based on a principle of reverse search. It was efficient and scalable for mining long graph sequence patterns and was several orders of magnitude faster than original GTRACE. Inokuchi and Washio [15] defined a subsequence class to find a complete set of frequent patterns efficiently from graph sequences. An algorithm FRISSs was put forward to efficiently mine frequent sequential patterns. Uno and Uno [16] proposed a graph mining algorithm to find all frequent sequential patterns in a graph sequence by enumerating all vertex subsets that are connected or cliques for a certain time period in a given graph sequence. A way of representing a graph sequence as the input format is defined. In this model, a graph sequence is represented by explicitly associating each edge with its time intervals during which it exists. Besides existing graph sequence

mining approaches, typical sequential pattern mining algorithms can also be applied to mine frequent sequential patterns from graph sequences. First, sequences are obtained by traversing a graph with DFS. Then a typical sequential pattern mining method called MEMISP [17] is utilized to mine frequent sequential patterns from sequences obtained above. For graph sequence mining or general sequence mining algorithms, frequent pattern is extended only with frequent items in its subsequent positions, and it is unfit for extensions of software executing paths. In this case, mined results contain some patterns which do not correspond to continuity characteristics of software executing paths.

When a key path is expanded, only next adjacency position of the path needs to be considered. If node in next adjacency position is not a key node, subsequent extension processes will be terminated. To mine key paths efficiently in complex software executing network graph, a novel approach Graph-Mine is put forward in this paper. The major contributions of this study can be summarized as follows.

Firstly, the model of complex software executing network graph is built, and important functions and call relationship among them are mapped to key nodes and paths respectively.

Secondly, software executing paths are obtained after traversing complex software executing network graph by DFS.

Thirdly, an algorithm Key-Path-Search is proposed to mine key paths from software executing paths obtained above.

Fourthly, index set is constructed to improve efficiency of mining key paths and memory utilization.

Finally, to improve efficiency of Graph-Mine, an adjacency path extension strategy is designed to extend key paths efficiently only with key nodes in its adjacent position, rather than key nodes in its subsequent positions.

The remaining of the paper is organized as follows. In Section 2, problems are defined. Algorithms are described in Section 3. In Section 4, algorithm instances are given. Section 5 is experiments. The paper is concluded in Section 6.

2. Descriptions and Definitions of Problems.

Complex software executing network can be expressed by a complex directed graph containing a lot of nodes and directed edges.

Definition 2.1. *SENG (Software Executing Network Graph). Software Executing Network Graph is a directed graph composed of nodes and directed edges. Nodes and edges represent functions and call relationships among functions respectively in software executing processes.*

Example 2.1. *The example of SENG can be seen as Figure 1. The nodes with the letters a, b, c, d, e, f and g mean the functions in softare source code. The edges with arrows are call relationships between each two nodes.*

Definition 2.2. *SEP (Software Executing Path). Software Executing Path is an access from one node to the other node in complex software executing network graph. It can be expressed as* $[e_1\text{->}e_2\ldots\text{->}e_n]$*, in which each element* e_k *means the k-th node in the SEP* ($k > 1$, *k is an integer) and there exists call relationship between* e_k *and* e_{k+1}.

The length of SEP p, expressed as $|p|$, is the total number of elements contained by p. If $|p| = k$, SEP p is a k-path. For instance, both [a->b->c] and [a->c->d] are 3-paths.

Definition 2.3. *Subpath. Path* $p = [a_1\text{->}a_2\ldots\text{->}a_n]$ *is a subpath of path* $p' = [b_1\text{->}b_2\ldots\text{->}b_m]$ *if there exists* $n \leq m$ *and* $a_i = b_i$, $a_{i+1} = b_{i+1}, \ldots, a_n = b_n$ ($i \geq 1$, *i is an integer).*

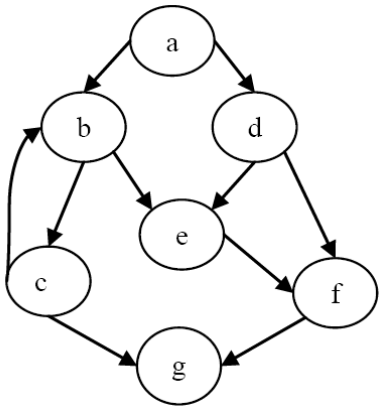

FIGURE 1. An example of complex software executing network graph

Path p' contains path p if p is a subpath of p'. For example, [b->m] is a subpath of [a->b->m->n].

Definition 2.4. *Support(p). P is a collection of all paths in software executing path database (SEP-DB). The support of path p, denoted by support(p), is the number of paths containing p divided by the total number of paths in SEP-DB and expressed as follows.*

$$Support(p) = \{p|p \subseteq P\} / |P| \tag{1}$$

Definition 2.5. *KPSE (Key Path of Software Executing). Minsup is a user-defined minimum support threshold. If support $(p) \geq$ minsup, path p is called a key path of software executing, which is called a key path for short in the following. Note that key nodes are key 1-paths.*

APES (Adjacent Path Extension Strategy). Considering that extension of software executing path is successive, when a key path is expanded, only node in next adjacency position of the path needs to be considered. If node in next adjacency position of the path is not a key node, extension processes will stop and will not continue to find key nodes in subsequent positions.

Definition 2.6. *Ext-Path (Extended Path). Given a key path of software executing* [KP] *and a key node x,* [KP$'$] *is an extended path if it can be formed by extending* [KP] *with x.*

Definition 2.7. *Pre-Path (Prefix Path). If an Ext-Path* [KP$'$] *is formed by expanding* [KP] *with a key node x,* [KP] *is a prefix path of* [KP$'$].

Definition 2.8. *Core. If an Ext-Path* [KP$'$] *can be formed by a Pre-Path* [KP] *and a key node x, x is called a core of* [KP$'$].

3. Graph-Mine: Key Path Mining in Complex Software Executing Networks.

3.1. **Framework of graph-mine.** Normally, functions and call relationships among them are extracted from software source code. Then the model to build complex software executing network is designed to map functions and call relationships into graph. Software executing paths will be extracted from the graph by DFS method after this. At last, key paths will be obtained by mining these software executing paths. The framework of the processes can be seen as Figure 2.

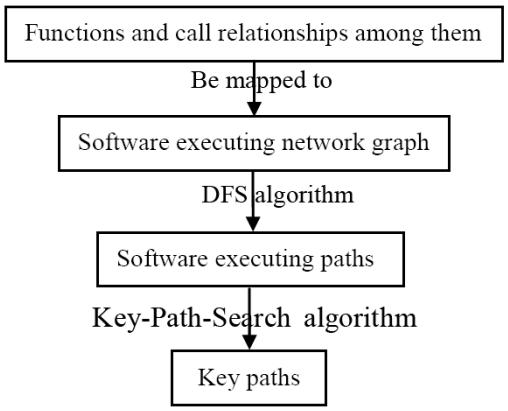

FIGURE 2. Processes of finding key paths

3.2. **Graph-Mine algorithm.** An algorithm Graph-Mine is proposed to mine key paths in complex software executing networks graph in this section. In the algorithm, firstly functions and call relationship among them are mapped to nodes and paths respectively in complex software executing network graph SENG. Then, Depth-First-Search strategy is adopted to transform SENG into software executing path SEP and store SEP into software executing path database SEP-DB. Finally, a novel algorithm Key-Path-Search is devised to mine key paths efficiently from SEP-DB. Graph-Mine algorithm is performed as follows.

Algorithm 3.1. Graph-Mine (SENG)

Input: Software executing network graph SENG.
Output: Software executing path database SEP-DB, key paths of software executing KPSE.

Begin:
(1) Call DFS-SENG (SENG) to get SEP-DB;
(2) Call Key-Path-Search (SEP-DB) to get KPSE;
(3) Return SEP-DB and KPSE;
End

Algorithm 3.1 outputs software executing path database and key paths. In the algorithm, firstly sub-procedure DFS-SENG (Algorithm 3.2) is called to transform complex software executing network graph SENG into software executing path SEP and store SEP into SEP-DB. Then sub-procedure Key-Path-Search (Algorithm 3.3) is called to mine key paths efficiently from SEP-DB.

Algorithm 3.2 outputs software executing path database in which paths correspond to call processes among functions. In the algorithm, software executing paths are found recursively from complex software executing network graph by DFS (lines 1-8). Then obtained paths are stored in SEP-DB (line 12).

TABLE 1. Software executing path database SEP-DB

Pid	software executing paths
P1	[a->b->c->g]
P2	[a->d->e->f->g]
P3	[a->b->e->f->g]
P4	[a->b->c->b->c->g]
P5	[a->d->f->g]
P6	[a->b->c->b->e->f->g]

Algorithm 3.2. DFS-SENG (SENG)

Input: SENG.
Output: SEP-DB.

```
Begin:
(1) If (SENG.Root!=Null), then
(2)     SEP=SEP+SENG.Root.Name;
(3)End if
(4) If (SENG.Root.child!=Null), then
(5)    For (every node of Child SENG.Root), do
(6)       SEP=SEP+SENG.Root.Name;
(7)       If (SENG.Root.child!=null), then
(8)          DFS-SENG (SENG.Root.child);
(9)       End if
(10)   End for
(11) else
(12)   SEP-DB.add (SEP);
(13) End if
(14) Return SEP-DB
End
```

Algorithm 3.3. Key-Path-Search (SEP-DB)

Input: Software executing path database SEP-DB.
Output: Set of key path of software executing KPSE.

```
Begin:
(1) Scan and store SEP-DB into MSEP-DB (memory SEP-DB), find all key nodes.
(2) For each key node x, do
(3)    Form key 1- path [KP]=<x> and add [KP] to set of KPSE;
(4)    Call CreateIndexSet(x,<>, MSEP-DB) to construct index set [KP]-idx;
(5)    Call Key-Path-Mine([KP], [KP]-idx) to get set of KPSE with index set [KP]-idx;
(6) End for
End
```

Example 3.1. *Figure 1 represents software executing network graph. It is converted into software executing paths by depth-first traversal, only part of the paths are given in Table 1 and Pid is path number.*

Algorithm 3.3 outputs set of key paths of software executing. Key-Path-Search is presented to mine key paths from path database obtained by DFS-SENG. In Key-Path-Search, adjacency path extension strategy is designed to extend paths successively with adjacent nodes. Software executing paths can be read into memory only by scanning database once. And index sets containing pointers and positions are used to tag positions and corresponding paths of a key path. Therefore, key paths can be extended efficiently. In Key-Path-Search mining algorithm, firstly software executing path database is read into main memory according to the size of memory and index sets are constructed for paths whose support satisfies the minsup (lines 1-4). Then set of key paths of software executing are mined recursively by index sets (line 5).

Algorithm 3.4. CreateIndexSet(x, [KP], Path-Set)

Input: Core x, [KP] is a Pre-Path, Path-Set is software executing path set for indexing.
Output: Index set [KP′]-idx.

```
Begin:
(1) For each path SEP in Path-Set, do
(2)    If (Path-Set==MSEP-DB), then
(3)      startPosition=0;
(4)    Else
(5)      startPosition=pos;
(6)    End if
(7)    Start from (startPosition+1)-th position in SEP
(8)       If (core-node x is first found in position pos of SEP)
(9)          Insert a (pos, ptr) pair to index set [KP′]-idx, where ptr points to SEP;
(10)     End if
(11) Return [KP′]-idx;
(12) End for
End
```

Algorithm 3.4 outputs index sets of key paths. CreateIndexSet constructs index database to reduce times and scopes of scanning database. In the algorithm, the value of startPosition is got according to the type of Path-Set (lines 1-5). Then core-node x is found and index set is constructed for Ext-Path [KP′] (lines 7-11).

Algorithm 3.5. Key-Path-Mine([KP], [KP]-idx)

Input: [KP], [KP]-idx
Output: Set of key paths of software executing KPSE.

```
Begin:
(1) For each software execute path SEP in [KP]-idx, do
(2)    Increase support count of potential core in the (startPosition+1)-th position
       of SEP by 1.
(3) End for
(4) Find set of cores x of [KP] from potential cores having enough support count.
(5) For each Core x of [KP], do
(6)   Construct Ext-Path [KP′] with Pre-Path [KP] and Core x, add [KP′] to the
      set of KPSE.
(7)   Call CreateIndexSet(x, [KP], [KP]-idx) to construct index set [KP′]-idx;
(8)   Call Key-Path-Mine([KP′], [KP′]-idx) to mine KPSE with index set [KP′]-idx;
(9) End for
(10) Output the set of KPSE.
End
```

Algorithm 3.5 outputs set of key paths of software executing by adjacency path extension strategy corresponding to call law among software functions. Positions and corresponding paths of key paths are tagged with index sets containing positions and pointers. In the algorithm, for a key path whose index set is constructed, nodes in next adjacency position (startPosition+1) of the path are counted to find cores satisfying the minsup (lines 1-4). Then key paths can be expanded efficiently with cores by adjacency path extension strategy (lines 5-10). Finally, sub-procedure CreateIndexSet and Key-Path-Mine are called recursively to mine longer key paths of software executing.

Above all, compared with previous algorithms, Graph-Mine has considered that extension of software executing path is continuous by adjacency path extension strategy, which extends a key path efficiently only with key nodes in its adjacent position, rather than

key nodes in its subsequent positions. Meanwhile, key paths can be applied to analyze software behavior and structure of software systems, locate software bugs and improve efficiency of software tests.

4. **Algorithm Instances.** Software is widely used in various domains, and a large number of call relations among functions will be produced. It is of great significance to analyze data of call relations among functions to get key paths. Because it will improve efficiency of detecting abnormal software by matching key paths with abnormal path database. In this instance, functions and all relations among them are normally represented as Figure 1 and minsup = 60%, software executing paths are obtained from complex software executing network graph by depth-first traversal. Then the paths are stored in SEP-DB as Table 1. Key paths of software executing are found by Key-Path-Search, which is described by the instance.

Step 1. SEP-DB is read into main memory and frequent software executing 1-paths are found. Nodes are counted to find all key nodes. MSEP-DB represents software executing path database in the memory. Key node a (support = 100%, for appearing in six paths P1, P2, P3, P4, P5, P6), b (support = 2/3), f (support = 2/3), g (support = 100%) are obtained. Key nodes are cores of Pre-Path = <>. Steps 2 and 3 are looped on each core to get all key paths.

Step 2. Output key path [KP] formed by current Pre-Path and Core x, and construct an index set [KP]-idx. An Ext-Path [KP] formed by current Pre-Path and core x is output, and index set [KP]-idx is constructed. Next, if a path contains x, a (pos, ptr) pair is allocated for it, pos is the first occurring position of x in the path and ptr is a pointer pointing to the path. Index set [KP]-idx is a collection of these (pos, ptr) pairs.

Take Core $x = a$ for example, its Pre-Path is <>. Frequent software executing 1-path [KP] = [a] is output and index set [a]-idx is constructed as shown in Figure 3-(1). For instance, pos is 1 for P1 = [a->b->c->g].

Step 3. Use index set [KP]-idx and MSEP-DB to find Cores with respect to Pre-Path = [KP]. Now ptr of every (pos, ptr) pair in [KP]-idx points to paths which contain [KP]. Any core appearing in position (pos+1) may be a potential core of [KP]. For every path in [KP]-idx, count of potential core in position (pos+1) is increased by one. A core meeting minimum support is found.

Continue to take [a]-idx for example. Pos of (pos, ptr) pointing to P1 is 1. Only nodes appearing in position (pos+1) in P1 need to be counted. Count of potential core b is increased by one. Similarly, nodes at 2 in P3, P4 and P6 are counted. After minimum support is verified, a core of Pre-Path = [a] is achieved. Continue to mine key paths with Pre-Path = [a] and core b in the following.

A key path of software executing [a->b] is generated and output by applying Step 2. If an SEP contains [KP], then a new (pos, ptr) is inserted into [KP]-idx ([a->b]-idx). While [a->b]-idx is constructed, only the most recent paths which constain [a]-idx needs to be checked, rather than entire MSEP-DB. Assume that (pos, ptr) pair in [a]-idx points to SEP. Search for core b with respect to Pre-Path = [a] is in position (pos+1) in SEP. Core b occurs at 2 in P1, P3, P4 and P6. So new index set [a->b]-idx is obtained as shown in Figure 3-(2). After generating [a->b] with [a->b]-idx, mining process is terminated.

Step 3 is used for [a->b] and MSEP-DB. No cores can be found to form extended paths. Therefore, mining processes are terminated and previous index set [a]-idx is popped. All subsequent find-then-index processes with respect to core a are finished now. Continue to mine key paths with key nodes f and g as above. Step 2 and Step 3 are used repeatedly till all key paths are found.

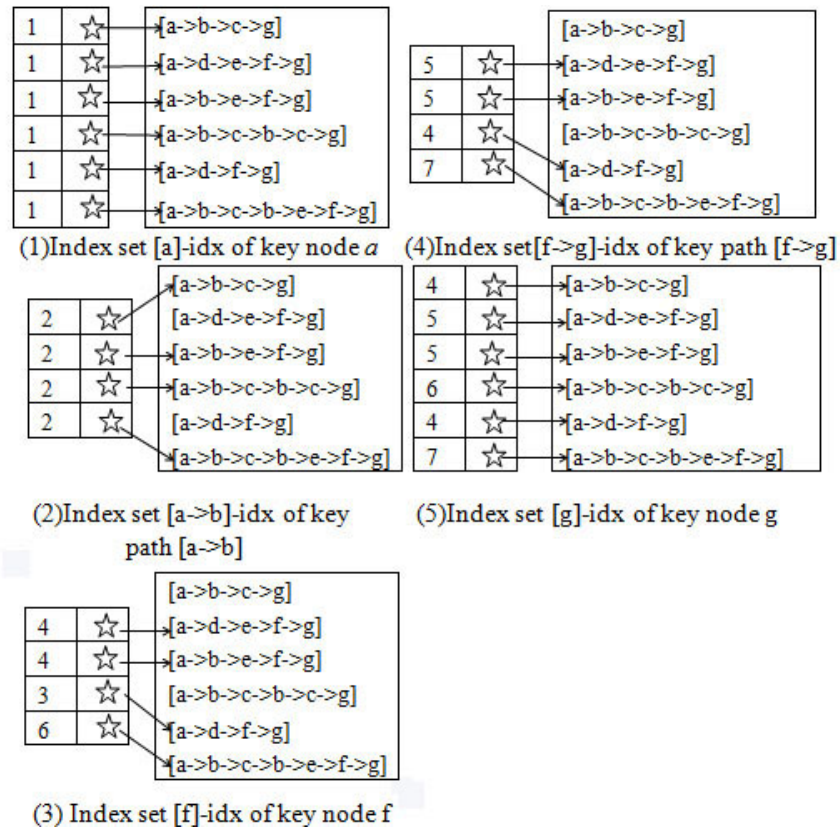

FIGURE 3. Some index sets and software executing path database in memory

In total, the mining results are meaningful. Firstly, abnormalities or vulnerabilities may appear frequently in software development, so these abnormalities or vulnerabilities can be found by analyzing key paths. Secondly, when software cannot work normally, key paths in the results can be firstly considered to maintain the software. Thirdly, software industry can be improved by reducing the occurring rate of abnormalities or vulnerabilities. What is more, the leaders can make right decisions according to the mining results.

5. **Experiments.** In this section, performance of Graph-Mine is tested. And it is compared with corresponding algorithms named by Graph-MEMISP-Mine, Graph-PrefixSpan-Mine and Graph-GSP-Mine, which are a typical sequential pattern mining approach used to mine frequent patterns from sequences obtained after traversing a graph. Experiment is conducted on Windows 7 system, CPU of AMD A8-3850 2.90 GHZ and 4.00G Memory. Algorithm is implemented in Java and compiled by NetBeans IDE 7.2.

5.1. **Experimental data sets and parameter setting.** Data sets in our experiments are from a real software Weka. Understand tool is adopted to generate a graph file from source code of software Weka, then Gephi tool is used to extract information of nodes and edges from the graph file. The number of nodes ranges from 300 to 1994 in the experiment. The number of edges varies from 800 to 4460. And minsup ranges from 0.2% to 1.5%.

5.2. **Result and performance of algorithm.** The number of nodes is increasing, minimum support is 0.6% and number of edges is 4000, both of executing time of Graph-Mine and Graph-MEMISP-Mine are increasing, and time consumption of Graph-Mine is significantly lower than that of Graph-MEMISP-Mine, Graph-PrefixSpan-Mine and Graph-GSP-Mine in Figure 4. The amount of edges is increasing, minimum support is 1% and

number of edges is 1200, both executing time of Graph-Mine and Graph-MEMISP-Mine are increasing, and time consumption of Graph-Mine is much lower than that of other three algorithms in Figure 5. Because the size of complex software executing network graphs is increasing and Graph-Mine has considered continuity characteristics of software executing path, only next adjacency position is taken into account to expand paths. If node in next adjacency position is not a key node, expansion processes will be terminated. The result shows high efficiency of Graph-Mine. However, Graph-MEMISP-Mine, Graph-PrefixSpan-Mine and Graph-GSP-Mine have taken all subsequent positions into account to extend paths and do not consider the pruning strategy.

Minimum support is increasing and the number of edges is 4460 in Figure 6, executing time of Graph-Mine and Graph-MEMISP-Mine is obviously decreasing under different minsup and different number of nodes in Figure 6. In Figure 7, minimum support is

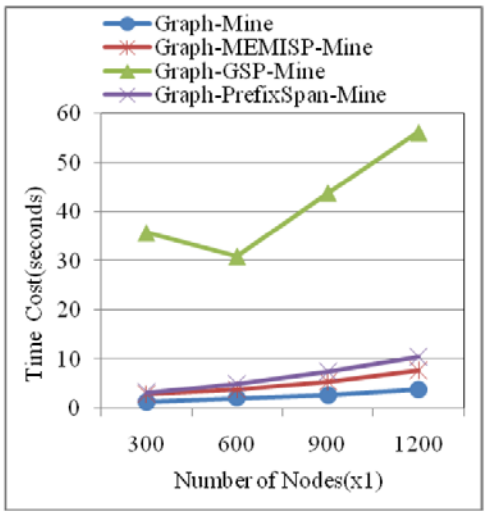

FIGURE 4. Executing time under different number of nodes

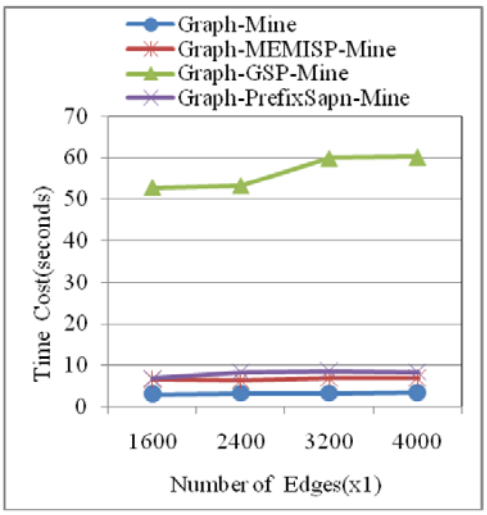

FIGURE 5. Executing time under different number of edges

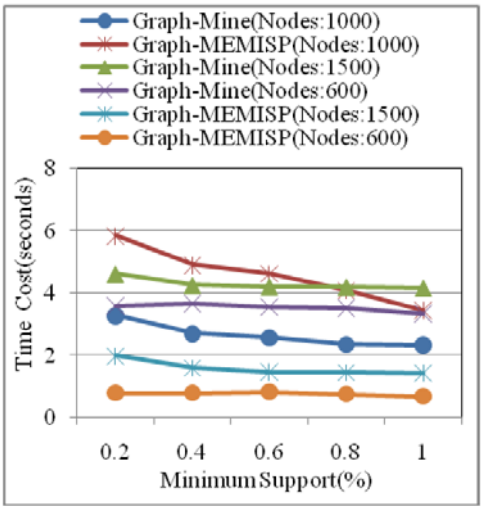

FIGURE 6. Executing time under different minsup and different number of nodes

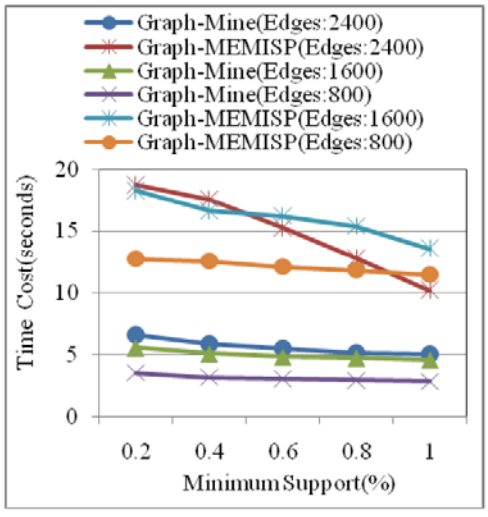

FIGURE 7. Executing time under different minsup and different number of edges

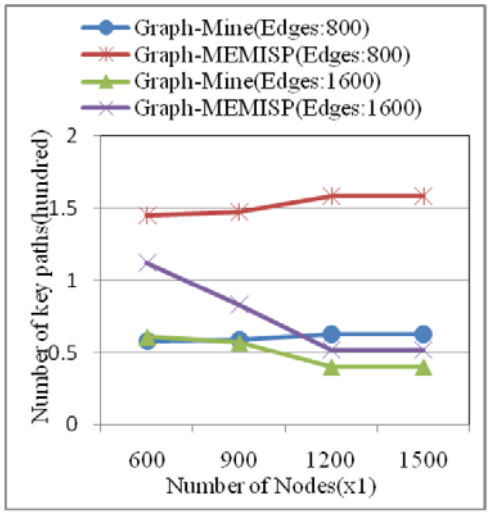

FIGURE 8. Number of key paths under different number of nodes and edges

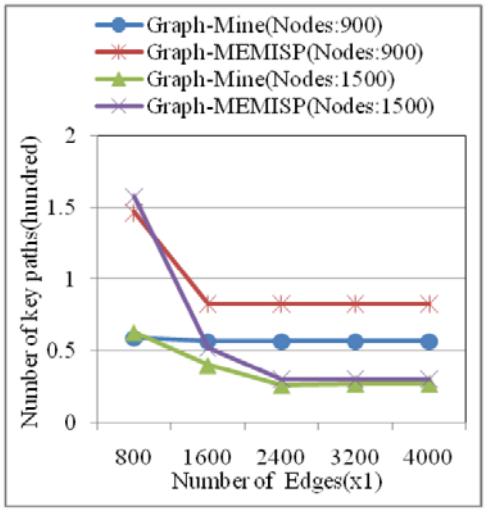

FIGURE 9. Number of key paths under different number of edges and nodes

increasing and the amount of nodes is 1994, time consumption of Graph-Mine and Graph-MEMISP-Mine are obviously decreasing under different minsup and different number of edges. Because the size of complex software executing network graphs is not changing when minimum support is increasing. Therefore, the quantity of frequent software executing paths is decreasing. Time cost of Graph-Mine is significantly lower than that of Graph-MEMISP-Mine under the same condition. Because Graph-Mine only needs to consider next adjacency position to extend paths by adjacency path extension strategy, which avoids a lot of inappropriate mining results and improves efficiency. The result validates Graph-Mine has high efficiency and good scalability. However, Graph-MEMISP-Mine does not take the superior prune strategy into consideration, and all subsequent positions must be considered to expand paths.

Minimum support is 1.5% and the number of nodes is increasing, the amount of key paths is decreasing gradually under different number of edges and the quantity of key paths generated by Graph-Mine is lower than that by Graph-MEMISP-Mine in Figure 8. In Figure 9, minimum support is 1.5% and the number of edges is increasing under different number of nodes, the count of key paths generated is decreasing gradually and key paths of Graph-Mine are significantly lower than those of Graph-MEMISP-Mine. The results indicate high accuracy and good scalability of Graph-Mine. Because this benefits from superior pruning method of Graph-Mine, which is applicable for extensions of software executing path. Graph-Mine has taken call characteristics of software executing into account to expand paths by adjacency path extension strategy. However, Graph-MEMISP-Mine does not take the pruning strategy into account to extend paths.

The number of edges is 4460 and minsup is increasing, the amount of key paths is decreasing gradually under different number of nodes in Figure 10. The number of nodes is 1994 and minsup is increasing under different number of edges in Figure 11, the number of key paths is decreasing little by little. Because the size of complex software executing network graphs is not changing when minimum support is increasing, the number of frequent software executing paths is decreasing. Key paths of Graph-Mine are less than those of Graph-MEMISP-Mine, because Graph-Mine adopts adjacency path extension strategy to prune unsuitable results, which validate high accuracy of Graph-Mine.

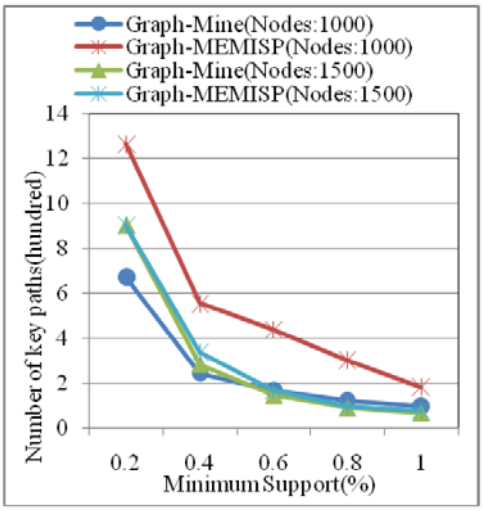

FIGURE 10. Number of key paths under different minsup and different number of nodes

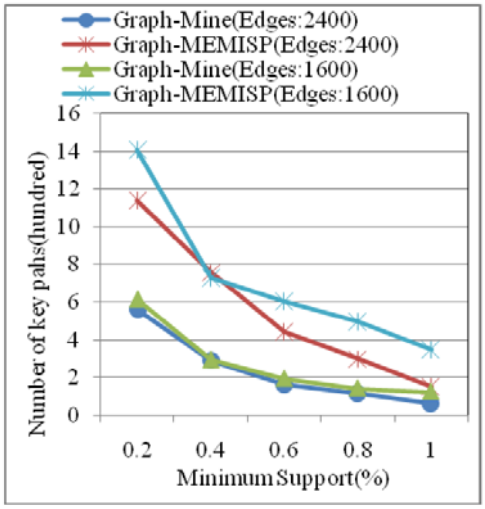

FIGURE 11. Number of key paths under different minsup and different number of edges

Above all, Graph-mine algorithm is efficient in executing time and the amount of key paths. The advantage of adjacency path extension strategy is fully displayed.

6. **Conclusions.** Considering that there exist issues of path extension and low efficiency in finding key paths by previous algorithms, a novel algorithm Graph-Mine is proposed to mine key paths in complex software executing networks. In the algorithm, complex software executing network graph SENG is defined, and Depth-First-Search algorithm is adopted to transform complex software executing network graph SENG into software executing paths SEP. Next, a novel algorithm Key-Path-Search is put forward to mine key paths of software executing from SEP. In Key-Path-Search, a novel adjacency path extension strategy is designed to filter out unsuitable results and improve efficiency, and Pre-Path and Ext-Path are defined to mine key paths conveniently and efficiently by adjacency path extension strategy. Instance analysis of Graph-Mine shows that the mining results are meaningful and key paths obtained can be used to improve efficiency of matching key paths with abnormal knowledge database. Experimental results indicate that Graph-Mine is more efficient in finding key paths in complex software executing networks.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61170190, and the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2012203062, No. F2013203324 and No. F20142031 52. The authors are also appreciated to the valuable comments and suggestions of the reviewers.

REFERENCES

[1] S. Valverde, R. F. Cancho and R. Solé, Scale free networks from optimal design, *Europhysics Letters*, vol.60, no.12, 2002.

[2] K.-Y. Cai and B.-B. Yin, Software executing processes as an evolving complex network, *Information Sciences*, vol.179, no.12, pp.1903-1928, 2009.

[3] 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.

[4] A. Inokuchi, T. Washio and H. Motoda, An apriori-based algorithm for mining frequent substructures from graph data, *Principles of Data Mining and Knowledge Discovery*, pp.13-23, 2000.

[5] A. Inokuchi, T. Washio and H. Motoda, Complete mining of frequent patterns from graphs: Mining graph data, *Machine Learning*, vol.50, no.3, pp.321-354, 2003.
[6] X. Yan and J. Han, gSpan: Graph-based substructure pattern mining, *Proc. of 2002 IEEE International Conference on Data Mining*, pp.721-724, 2002.
[7] E. Shen and T. Yu, Mining frequent graph patterns with differential privacy, *Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.545-553, 2013.
[8] M. Kuramochi and G. Karypis, Finding frequent patterns in a large sparse graph, *Data Mining and Knowledge Discovery*, vol.11, no.3, pp.243-271, 2005.
[9] M. Fiedler and C. Borgelt, Subgraph support in a single large graph, *The 7th IEEE International Conference on Data Mining Workshops*, pp.399-404, 2007.
[10] K. M. Borgwardt, H. P. Kriegel and P. Wackersreuther, Pattern mining in frequent dynamic subgraphs, *The 6th International Conference on Data Mining*, pp.818-822, 2006.
[11] M. Lahiri and T. Y. Berger-Wolf, Mining periodic behavior in dynamic social networks, *The 8th International Conference on Data Mining*, pp.373-382, 2008.
[12] M. Berlingerio, F. Bonchi, B. Bringmann et al., Mining graph evolution rules, *Machine Learning and Knowledge Discovery in Databases*, pp.115-130, 2009.
[13] A. Inokuchi and T. Washio, GTRACE: Mining frequent subsequences from graph sequences, *IEICE Transactions on Information and Systems*, vol.93, no.10, pp.2792-2804, 2010.
[14] A. Inokuchi, H. Ikuta and T. Washio, GTRACE-RS: Efficient graph sequence mining using reverse search, *IEICE Transactions on Information and Systems*, 2011.
[15] A. Inokuchi and T. Washio, FRISSMiner: Mining frequent graph sequence patterns induced by vertices, *IEICE Transactions on Information and Systems*, vol.95, no.6, pp.1590-1602, 2012.
[16] T. Uno and Y. Uno, Mining preserving structures in a graph sequence, *arXiv Preprint arXiv: 1206.6202*, 2012.
[17] M. Y. Lin and S. Y. Lee, Fast discovery of sequential patterns by memory indexing, *Inf. Sci. Eng.*, vol.21, no.1, pp.109-128, 2005.