International Journal of Innovative
Computing, Information and Control
Volume 11, Number 6, December 2015

 ISSN 1349-4198
pp. 1897–1911

# EFFICIENT SOFTWARE FAULT LOCALIZATION BY HIERARCHICAL INSTRUMENTATION AND MAXIMAL FREQUENT SUBGRAPH MINING

Jiadong Ren[1,2], Huifang Wang[1,2,*], Yue Ma[1,2], Yanling Li[1,2] and Jun Dong[1,2]

[1]College of Information Science and Engineering
Yanshan University
[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
No. 438, Hebei Ave., Qinhuangdao 066004, P. R. China
{jdren; dongjun}@ysu.edu.cn; *Corresponding author: 1215847706fang@sina.com
{my824808617; wintersmiles}@163.com

Received April 2015; revised August 2015

Abstract. *With large-scale increasing of software, locating software faults effectively is significant to improve software stability and robustness, and important for software debugging. This paper uses a selective hierarchical instrumentation method based on the granularity of functions and basic blocks to obtain software call traces. A software execution process is mapped as a directed graph. In this paper, a novel approach to mine maximal frequent subgraphs (MFSH-TreeMiner) from software dynamic call graphs is proposed. In the MFSH-TreeMiner, a maximal frequent subgraph hierarchical tree (MFSH-Tree) structure is constructed by accessing graph database only once. Combining executions set and executions complementary set, a novel metric is proposed for measuring the suspicious value of feature nodes in the passing and failing call graphs. Then feature nodes are sorted according to the descending order of the suspicious value to locate faults. Siemens benchmark test suite is used as the experimental subject to evaluate the performance of our approach; experimental results demonstrate that our approach is both efficient and effective for software fault localization.*

**Keywords:** Fault localization, Software dynamic call graph, Maximal frequent subgraph, Feature node

1. **Introduction.** With the prosperous development of software industry, software becomes more sophisticated. In order to guarantee software quality, a lot of effort resources are invested. Software testing accounts for 50%-75% of software development and maintenance [1]. Especially, fault localization is the most time-consuming and difficult process in software testing. Some software faults are non-crashing and difficult to be detected manually. Effective fault localization can significantly reduce manpower and time consumption during software debugging, and improve software quality.

Researchers have obtained a lot of achievements in software debugging and fault localization recently. J. Jones and M. Harrold [2] concluded that if a statement is more frequent in the failing executions than the passing executions, the probability of the statement leading to faults is larger. Based on the dynamic spectrum-based approach, R. Abreu et al. [3] presented a multiple-fault localization technique called BARINEL. A maximum likelihood estimation approach is introduced to measure the bug-relevance of predicates in BARINEL. These spectrum calculation methods mainly consider faults from statements and predicates, ignoring faults that may appear at definitions. Moreover, these methods may cause the statements having a higher ranking in many test cases. In order to

narrow down the search scope, G. Neelam et al. [4] developed a program slicing approach to locate software faults by combining delta debugging [5] with the forward and backward dynamic program slicing. However, since program slicing considers all statements affecting the output, there are a lot of redundant codes which do not need to be detected. With the purpose of solving some disadvantages of program slicing and spectrum calculation method, B. Hofer and F. Wotawa [6] combined program slicing technology and spectrum calculation method to propose SENDYS method. The SENDYS method improves the accuracy of fault localization. As coverage information cannot identify statements whose executions affect the output, Y. Lei et al. [7] presented a statistical fault localization approach to address the issue. This approach uses an approximate dynamic backward slicing approach to take into account static slices and dynamic slices. Their experiment results show that the proposed approach is better than SFL. M. Renieris and S. Reiss [8] drew the conclusion that the successful path which is the most similar to the failing path is more beneficial for fault localization than the successful path chosen randomly. Hence, the neighbor model calculating the similarity of execution traces is proposed to locate faults. Nevertheless, it does not distinguish the executions whose statements are same but being executed in different order. Therefore, L. Guo et al. [9] introduced a control flow method by measuring the difference between execution runs. This method is more effective than the nearest neighbor model [8] on the detection of branch faults, but the results are still insufficient in the detection of statement faults. C. Sun and S. C. Khoo [10] presented a predicated bug signature method and a discriminative itemsets generator to discover succinct predicated bug signatures on data predicates and control flow information. To improve the predictive ability of bug signatures, Z. Zuo et al. [11] applied a hierarchical instrumentation technique to predicate bug signatures and proposed HIMPS algorithm to find top-$k$ bug signatures significantly. The HIMPS algorithm behaves well on large-scale programs especially.

A software execution process can be mapped as a directed graph. Analyzing software execution graph can find faults early and reduce software maintenance cost. C. Liu et al. [12] constructed a classification framework by using closeGraph [13]. The classification framework can find non-crashing bugs efficiently. However, the closeGraph is not suitable for large-scale software. Thus, F. Eichinger et al. [14, 15] applied reduction techniques to reduce software call graphs and proposed a software fault localization framework based on closeGraph. Then they combine information entropy and structure score to improve bug localization accurately. Since some frequent subgraphs may be irrelevant to bugs, H. Cheng et al. [16] put forward a top-$k$ discriminative graph mining algorithm (Top-K LEAP) by extending LEAP algorithm [17] to generate candidate discriminative graphs. These candidate discriminative graphs are used to identify bugs. However, the Top-K LEAP does not consider weight; S. Parsa et al. [18] presented a discriminative graph mining algorithm among edge-weighted graphs. In addition, they optimize the reduction strategy to find the potential fault characteristics subgraphs. Despite all this, some less frequent subgraphs may be obtained by mining discriminative subgraph. In order to locate complex faults, X. Wang and Y. Liu [19] put forward a hierarchical multiple predicate switching (HMPS) method by identifying critical predicates. The method contributes to the search for critical predicate by using instrumentation method and switching combination strategies.

Function call executions or statement call executions are used in previous work. However, if execution traces are only function call executions, all statements in functions need to be checked artificially for finding faults. If execution traces are only statement call executions, graph size will be large. To address this issue, we present a hierarchical instrumentation method to obtain function call traces and basic block call traces. R. Vijayalakshmi et al. [20] put forward the FP-GraphMiner algorithm to find frequent graphs

in network graphs. By constructing an FP-Graph which stores structural information of edges in all graphs, the FP-GraphMiner can obtain frequent graphs. Since some conditions are not considered in FP-GraphMiner, some frequent graphs cannot be obtained. Therefore, we propose an MFSH-TreeMiner algorithm based on FP-GraphMiner to find all maximal frequent graphs. Then feature nodes are found as fault signatures by using MFSH-TreeMiner in basic block call executions. Pinpoint [21] calculated the suspicious value of program entity by Jaccard which only considers faults occurring in the failing executions. However, faults may not appear in the passing executions. For purpose of locating faults accurately and quickly, we define a measure to calculate the suspicious value of feature nodes based on these two cases.

In summary the following contributions have been made in this paper:

- In order to obtain call traces on the granularity of functions or basic blocks, we propose a hierarchical instrumentation method. As graph is more compact and scalable than trace, call traces are reduced to form software dynamic call graphs for identifying faults easily.
- We construct a maximal frequent subgraph hierarchical tree (MFSH-Tree) structure by visiting software dynamic call graphs only once. An efficient maximal frequent subgraph mining (MFSH-TreeMiner) algorithm is presented to find all maximal frequent graphs by traversing MFSH-Tree from a leaf node to the root node.
- Since Jaccard which is used to calculate the suspicious value of program entity only considers the executions set, we designed a measure to calculate the suspicious value of feature nodes. This measure takes account of the executions set and the complementary executions set.

The remaining of the paper is organized as follows. Some definitions are given in Section 2. Section 3 introduces a framework of fault localization with maximal frequent subgraph mining and a measure for calculating the suspicious value. Experiments and analysis in Section 4 show the performance of our approach. Section 5 summarizes the paper.

2. **Preliminary Concepts.** A software dynamic call graph $s$ related to a call trace, is a 2-tuple $s = (V, E)$, where $V$ is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. Figure 1 is a sample of graph database consisting of four simple graphs.
*The vertices and edges represent basic blocks or functions and call relationships among them respectively in a software dynamic call graph.*

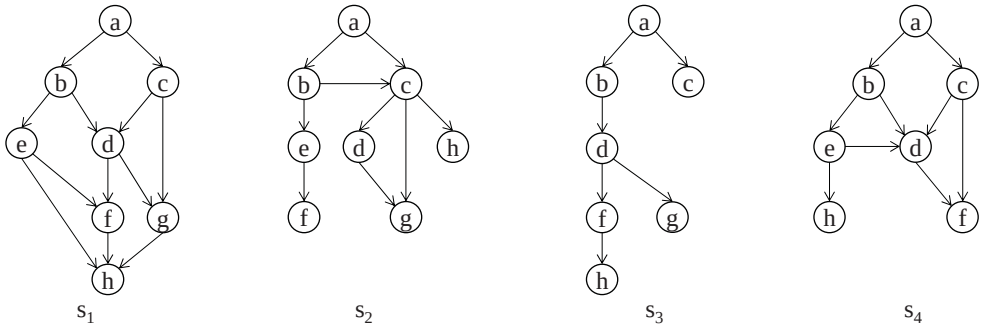



FIGURE 1. A sample of graph database

Given a graph database $S = \{s_1, s_2, \cdots, s_n\}$ and a user-specified minimum support $\sigma$, a subgraph is frequent if its support is no less than $\sigma$. $|S|$ is the number of graphs in $S$.

For a given support, let $F = \{f_1, f_2, \cdots\}$ be a set of all frequent subgraphs in $S$. If a subgraph $f_i \in F$ is a maximal frequent subgraph, there is no subgraph $f_j \in F$ to satisfy $f_i \subseteq f_j$.

The EdgeCode of an edge $e$ denoted as $EC(e)$ is a $|S|$ length string. Each character corresponds to a graph in $S$. If the character is 1, $e$ exists in the corresponding graph. On the contrary, $e$ does not exist in the corresponding graph. As shown in Figure 1, $EC(ab) = 1111$, $EC(bd) = 1011$.

The frequency of an edge $e$ denoted as $fre(e)$ or $fre(EC(e))$ is the number of 1 in $EC(e)$, also is the number of graph that contains $e$ in $S$. As shown in Figure 1, $fre(ab) = 4$, $fre(bd) = 3$, $fre(1011) = 3$.

A group denoted as $g = \{(EC(e_1), < e_1, e_2, \cdots >) | EC(e_1) = EC(e_2) = \cdots\}$ contains all edges with the same EdgeCode. The EdgeCode of $g$ is denoted as $EC(g) = EC(e_1)$. The frequency of $g$ is denoted as $fre(g) = fre(e_1)$.

A cluster denoted as $c = \{(fre(c), < g_1, g_2, \cdots >) | fre(g_1) = fre(g_2) = \cdots\}$ contains all groups with the same frequency. The frequency of $c$ is denoted as $fre(c) = fre(g_1)$.

A level $l$ includes clusters, $l = \{(Level, < c_1, c_2, \cdots >)\}$. $Level$ is the position of the level.

Maximal Frequent Subgraph Hierarchical Tree for short MFSH-Tree, is constructed by linking groups in different levels. Nodes are groups in levels. Let $l_i$, $l_j$ be two levels with $Level(l_i) > Level(l_j)$, where $l_i$ is the higher level of $l_j$. If $g_1 \cap g_{2(g_1 \in l_i,\, g_2 \in l_j)} = g_1$ is satisfied, $g_2$ is the parent group of $g_1$, $g_1$ is the child group of $g_2$. Any group has at most one parent group. If a group has a parent group but does not have a child group, it is a leaf node.

## 3. **A Framework of Software Fault Localization.**

This framework includes three critical phases, transforming software call traces to software dynamic call graphs, mining maximal frequent subgraphs as faulty regions, and calculating suspicious value. Software dynamic call graphs are built on the granularity of function and basic block respectively.

The framework of software fault localization is described in detail as follows.

Step 1: Software function call traces are collected during software executing. Every function call trace is assigned a label (passing, failing), which is determined by comparing and analyzing trace structure similarity. These function call traces are reduced to construct software dynamic call graphs.

Step 2: For a given support, the maximal frequent subgraph mining algorithm is used to find frequent called functions.

Step 3: Basic block call traces are acquired. These basic blocks appear in frequent called functions. And a label (passing, failing) is assigned to each call trace. These call traces are used to build dynamic call graphs.

Step 4: For a given support, frequent executed feature nodes are found with the maximal frequent subgraph mining algorithm in these dynamic call graphs.

Step 5: A measure to calculate the suspicious value of feature nodes is proposed to help developers locate faults quickly.

In Step 1, to collect software function call traces, some codes need to be added to the beginning of each function to form instrumented software. And the software execution results are not affected. In Step 4, in order to acquire basic block call traces, some codes are added to the basic blocks of frequent call functions to form instrumented software.

If the scale of software is small, some codes are directly inserted into the basic block of all functions to form an instrumented software in Step 3, skipping Steps 1 and 2.

### 3.1. **Building software dynamic call graphs.**

In order to build software dynamic call graphs, software call traces need to be obtained during software executing. We instrument functions or basic blocks of software artificially and run the instrumented software to collect software call traces. Every software call trace is assigned a label (passing, failing). Algorithm 1 presents the process of building software dynamic call graphs.

**Algorithm 1 Building Software Dynamic Call Graphs**

**Input:** a software, test cases
**Output:** dynamic call graphs $S$

```
1: instrument functions or basic blocks of the software;
2: Run the software to collect software call traces T;
3: S = ∅;  //a collection of software dynamic call graphs
4: for each trace t ∈ T do
5:     assign a label (passing, failing) to t;
6:     transform t to a software dynamic call graph s;
7:     S = S ∪ s;
8: end for
9: return S;
```

3.2. **Mining maximal frequent subgraph.** As accessing disk needs abundant time, scanning the graph database repeatedly increases the time complexity of the algorithm. Many proposed frequent subgraph mining algorithms scan the graph database more than once. Our algorithm aims to mine all maximal frequent subgraphs with a given support by scanning the graph database just only once.

Since each edge is distinct for a software dynamic call graph, an edge list representation is more efficient than a vertex adjacency matrix representation for the graph. Each edge in a graph is regarded as the 2-tuple $< B, E >$, where $B$ is the beginning node, and $E$ is the end node. As shown in Figure 1, $s_1 = \{ab, ac, bd, be, cd, cg, df, dg, ef, eh, fh, gh\}$.

A graph database is represented as a distinct edge list $EL$. Each edge is expressed as a two-tuple $< e, EC(e) >$ in the list, where $e$ represents the distinct edge, and $EC(e)$ represents the EdgeCode of $e$. The list is sorted in descending order according to the EdgeCode of edge. In the following details, edges that satisfy $fre(edge) \geq \sigma \cdot |S|$ are selected from the sorted list.

Mining maximal frequent subgraph includes two major phases, constructing MFSH-Tree and mining maximal frequent subgraph in MFSH-Tree.

3.2.1. *Constructing MFSH-Tree.* This process has two stages, forming different hierarchical levels and linking groups in different hierarchical levels to construct MFSH-Tree.

**Forming different hierarchical levels.** First, the edges with the same EdgeCode are combined into a group. These groups with the same frequency are grouped into a cluster. Each cluster constitutes a level. All levels form initial hierarchical levels. Second, groups in the same cluster are going to do AND operation according to the EdgeCode of group. Third, all new groups of the level with the same EdgeCode are formed into a group. These groups with the same frequency are grouped into a cluster. These clusters and the initial cluster of the level constitute a new level. Finally, all new levels form different hierarchical levels. The procedure of forming different hierarchical levels is performed in Algorithm 2.

Algorithm 2 maps graph database into Edge List $EL$ in line 1 to line 4. In line 5, $EL$ is sorted in descending order according to the EdgeCode of edge. The process of forming initial hierarchical levels is described in line 6 to line 13. In line 15 to line 17, groups in the same cluster do AND operation. The number of groups which are chosen to do AND operation is from $C_k^2$ to $C_k^j$, where $k$ represents the number of the groups in the cluster, $j$ satisfies that the frequency of all new groups that $C_k^{j+1}$ forms is less than $\sigma \cdot |S|$, $j \leqslant k$. Finally, forming different hierarchical levels is shown in line 18 to line 24. In order to link groups in different hierarchical levels conveniently, these clusters of each level in different hierarchical levels are ranked by the frequency of clusters in ascending order.

**Algorithm 2 Forming Different Hierarchical Levels**

**Input:** graph database $S$, support $\sigma$
**Output:** different hierarchical levels

1: $EL = \emptyset$; //Edge List
2: **for** each distinct edge $e_i \in S$ **do**
3: insert $< e_i, EC(e_i) >$ into $EL$;
4: **end for**
5: descending sort $EL$ by EdgeCode;
6: **for** each $g_i$ **do**
7: $g_i = \{(EC(e_1), < e_1, e_2, \cdots >) | EC(e_1) = EC(e_2) = \cdots\}$;
8: **end for**
9: $l = \{(0, < root >) | Level = 0\}$;
10: **for** each $c_i$ **do**
11: $c_i = \{(fre(c_i), < g_1, g_2, \cdots >) | fre(g_1) = fre(g_2) = \cdots\}$;
12: $l_i = \{(Level, < c_i >) | Level = i\}$;
13: **end for**
14: **for** each $l_i$ **do**
15: **for** each $c_j$ in $l_i$ && $fre(c_j) \geqslant \sigma \cdot |S|$ **do**
16: Groups in $c_j$ do AND operation according to the EdgeCode of group. The number of groups which are chosen to do AND operation in $c_j$ is from $C_k^2$ to $C_k^j$, $k$ represents the number of the groups in $c_j$, $j$ satisfies that the frequency of all new groups that $C_k^{j+1}$ forms is less than $\sigma \cdot |S|$, $j \leqslant k$;
17: **end for**
18: **for** each $g_{new}$ **do**
19: $g_{new} = \{(EC(g_1), < g_1, g_2, \cdots >) | EC(g_1) = EC(g_2) = \cdots\}$;
20: **end for**
21: **for** each $c_{new}$ **do**
22: $c_{new} = \{(fre(c_{new}), < g_1, g_2, \cdots >) | fre(g_1) = fre(g_2) = \cdots\}$;
23: insert $c_{new}$ into $l_i$;
24: **end for**
25: **end for**
26: **return** different hierarchical levels;

**Example 3.1.** *Initial hierarchical levels with* $\sigma = 50\%$ *is shown in Figure 2. Groups in* $Level = 3$ *does not need to do AND operation. Groups* $(1110, < dg >)$, $(1101, < be, cd >)$, $(1011, < bd, df >)$ *of* $c_1$ *in* $Level = 2$ *must do AND operation to generate groups* $(1100, < dg, be, cd >)$, $(1010, < dg, bd, df >)$, $(1001, < be, cd, bd, df >)$. *Groups* $(1100, < dg, be, cd >)$, $(1010, < dg, bd, df >)$, $(1001, < be, cd, bd, df >)$ *are formed into a cluster* $c_2$, $fre(c_2) = 2$. $c_1$ *and* $c_2$ *constitute a new level,* $l = \{2, (c_1, c_2) | Level = 2\}$. *Figure 3 shows different hierarchical levels with* $\sigma = 50\%$.

**Linking groups in different hierarchical levels.** Groups are linked by starting from the highest level. If a group is the parent group of another group, all edges in the group are marked as visited and stored in a visited edge collection called $VEC$. If all edges in a group are visited, we need to find the parent group of the group. These groups in different hierarchical levels are linked to form MFSH-Tree. All maximal frequent subgraphs can be mined in MFSH-Tree. Algorithm 3 represents the process of constructing MFSH-Tree.

If $i = 1$, the groups in $Level = 1$ can be linked to the root in line 5 of Algorithm 3. The process of finding the parent group of $g_1$ is shown in line 9 to line 22. The group

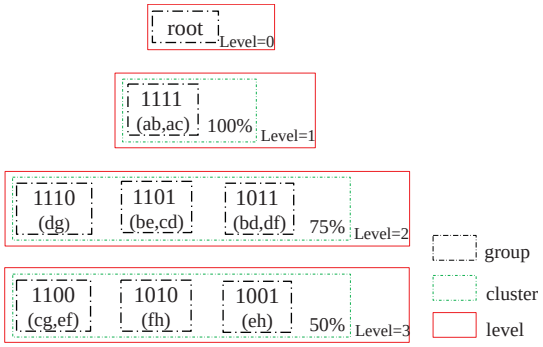



FIGURE 2. Initial hierarchical levels with $\sigma = 50\%$

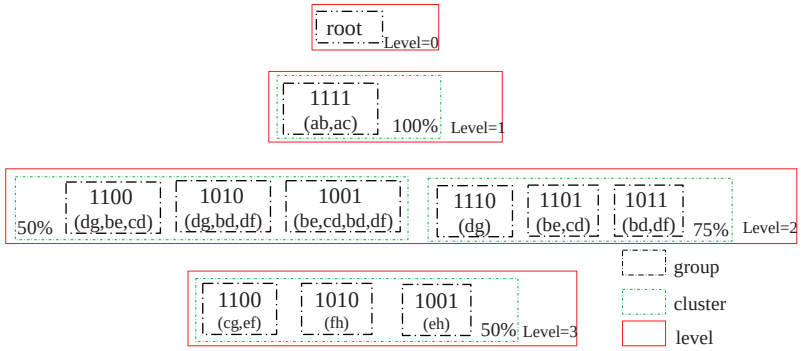



FIGURE 3. Different hierarchical levels with $\sigma = 50\%$

$g_2$ is selected from a cluster that has minimum frequency in Level $l_j$ $(j = i - 1)$ until $g_1 \cap g_{2(g_1 \in l_i,\, g_2 \in l_j)} = g_1$ is satisfied. If $g_1 \cap g_{2(g_1 \in l_i,\, g_2 \in l_j)} \neq g_1$ for all groups in $l_j$, the groups in the next higher level are examined to find the parent group of $g_1$. In line 11, if $j = 1$ and the parent group of $g_1$ is not found, $g_1$ should be linked to the root.

**Example 3.2.** *Different hierarchical levels with* $\sigma = 50\%$ *is presented in Figure 3. First we search the parent group of the group* $g_1 = \{(1100, < cg, ef >)\}$ *in Level* $= 3$ *and the cluster* $c_3$ *that satisfies* $fre(c_3) = 2$. *Since we find that* $g_2 = \{(1100, < dg, be, cd >)\}$ *is the parent group of* $g_1$, *where* $g_2$ *is in Level* $= 2$ *and the cluster* $c_4$ *that satisfies* $fre(c_4) = 2$, $g_1$ *is linked to* $g_2$. *All edges of* $g_2$ *are inserted into* $VEC$. *Because all edges of* $g_3 = \{(1110, < dg >)\}$ *have existed in* $VEC$, *where* $g_3$ *is in Level* $= 2$ *and the cluster* $c_5$ *that satisfies* $fre(c_5) = 3$, *we do not need to find the parent group of* $g_3$. *MFSH-Tree with* $\sigma = 50\%$ *is shown in Figure 4.*

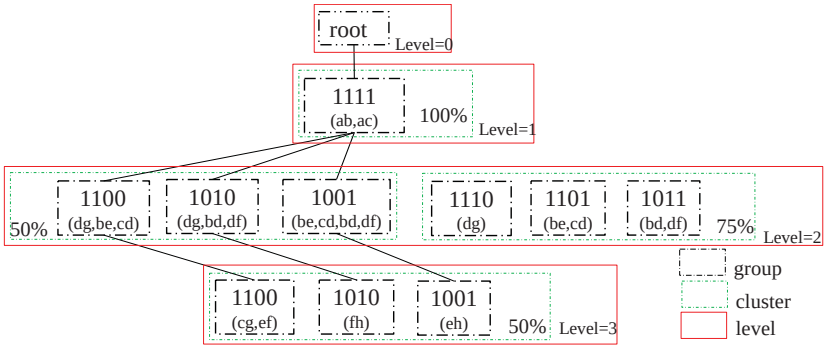



FIGURE 4. MFSH-Tree with $\sigma = 50\%$

**Algorithm 3 Constructing MFSH-Tree**

**Input:** different hierarchical levels, support $\sigma$
**Output:** MFSH-Tree

```
1: X = Maxinum Level;
2: VEC = ∅;  //visited edge collection
3: for each c1 in Level = i, X ⩾ i ⩾ 1 do
4:     for each g1 in c1 && ∀e_{e∈g1} ⊈ VEC do
5:         if i = 1 then
6:             g1.parent = root;
7:             break;
8:         else
9:             for each c2 in Level = j, i − 1 ⩾ j ⩾ 1 do
10:                if j = 1 then
11:                    g1.parent = root;
12:                    break;
13:                else
14:                    for each g2 in c2 do
15:                        if g1 ⋂ g2 = g1 then
16:                            g1.parent = g2;
17:                            insert all edges of g2 into VEC;
18:                            break;
19:                        end if
20:                    end for
21:                end if
22:            end for
23:        end if
24:    end for
25: end for
26: return MFSH-Tree;
```

3.2.2. *MFSH-TreeMiner: Maximal frequent subgraph mining in MFSH-Tree.* Since MFSH-Tree is constructed by the given support, all maximal frequent subgraphs are traversed from a leaf node to the root node in MFSH-Tree via finding the parent group of a group. A maximal frequent subgraph can be obtained from the collection that contains all edges of the groups in a traversal path. And the number of traversal paths is the number of maximal frequent subgraphs. MFSH-TreeMiner is presented in Algorithm 4.

**Example 3.3.** *As an example is in Figure 4, all maximal frequent subgraphs with* $\sigma = 50\%$ *are shown in Figure 5.*

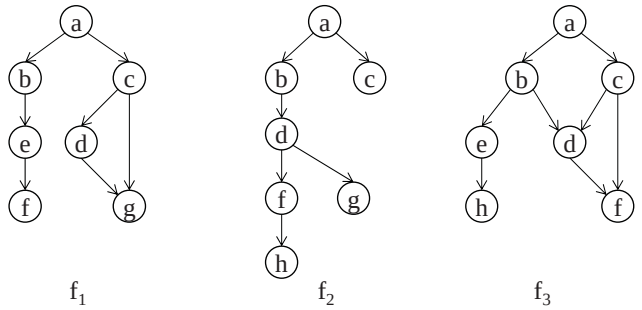



FIGURE 5. All maximal frequent subgraphs with $\sigma = 50\%$

**Algorithm 4 MFSH-TreeMiner**

**Input:** MFSH-Tree, support $\sigma$
**Output:** all maximal frequent subgraphs $F$

```
1: N = {n_1, n_2, ···};   //leaf nodes set
2: F = ∅;   //all maximal frequent subgraphs set
3: for each n_i in N do
4:     f = ∅;   //a maximal frequent subgraph
5:     insert all edges of n_i into f;
6:     while n_i = root do
7:         g = n_i.parent;
8:         n_i = g;
9:         insert all edges of n_i into f;
10:    end while
11:    insert f into F;
12: end for
13: return F;
```

In fact, for software fault localization in the passing and failing executions, the support is at most the percentage of fail executions and all executions. Obviously, when the support is lower, the number of feature nodes is more.

3.3. **Calculating suspicious value for fault localization.** Intuitively, a fault appears frequently in the failing executions but rarely in the passing executions. To help developers to locate faults effectively, a measure is proposed to compute the suspicious value of feature nodes. For software dynamic call graph sets, $totalpassed$ and $totalfailed$ are defined, corresponding to the passing and failing executions, whose correctness is determined by comparing and analyzing trace structure similarity. Given a feature node $n$ in the software, $passed(n)$ represents the number of $n$ existing in the passing executions; $failed(n)$ represents the number of $n$ existing in the failing executions; $un\_passed(n)$ represents the number of $n$ not existing in the passing executions; $un\_failed(n)$ represents the number of $n$ not existing in the failing executions.

Pinpoint [21] uses Jaccard coefficient to calculate the suspicious value of program entity, given in Equation (1).

$$Jaccard(n) = \frac{failed(n)}{failed(n) + un\_failed(n) + passed(n)}. \tag{1}$$

If a feature node contains faults, its suspicious value is related to the complementary set of executions. Based on complementary set of executions, a coefficient is defined, given in Equation (2).

$$Un\_Jaccard(n) = \begin{cases} 1 & un\_failed(n) = 0, \\ \dfrac{un\_passed(n)}{un\_passed(n) + un\_failed(n) + passed(n)} & other. \end{cases} \tag{2}$$

Combining executions set and complementary set of executions, a measure called *Suspicious* is designed to calculate the suspicious value of each feature node according to the execution results, given in Equation (3). Feature nodes are sorted in descending order according to the suspicious value. Then programmers can find the faults by combining with code. The process of finding feature node that contains faults is presented in Algorithm 5.

$$Suspicious(n) = \frac{Jaccard(n) + Un\_Jaccard(n)}{2}. \quad (3)$$

**Algorithm 5 Finding Feature Node that Contains Faults**

**Input:** all maximal frequent subgraphs $F$
**Output:** feature node $n$ that contains faults

```
N = ∅;  //feature nodes set
for each f_i in F do
    if n_{n∈f_i} ⊈ N then
        insert n_{n∈f_i} into N
    end if
end for
for each n_i in N do
    Suspicious(n);
end for
descending sort N by Suspicious(n);
find n that contains faults;
return n;
```

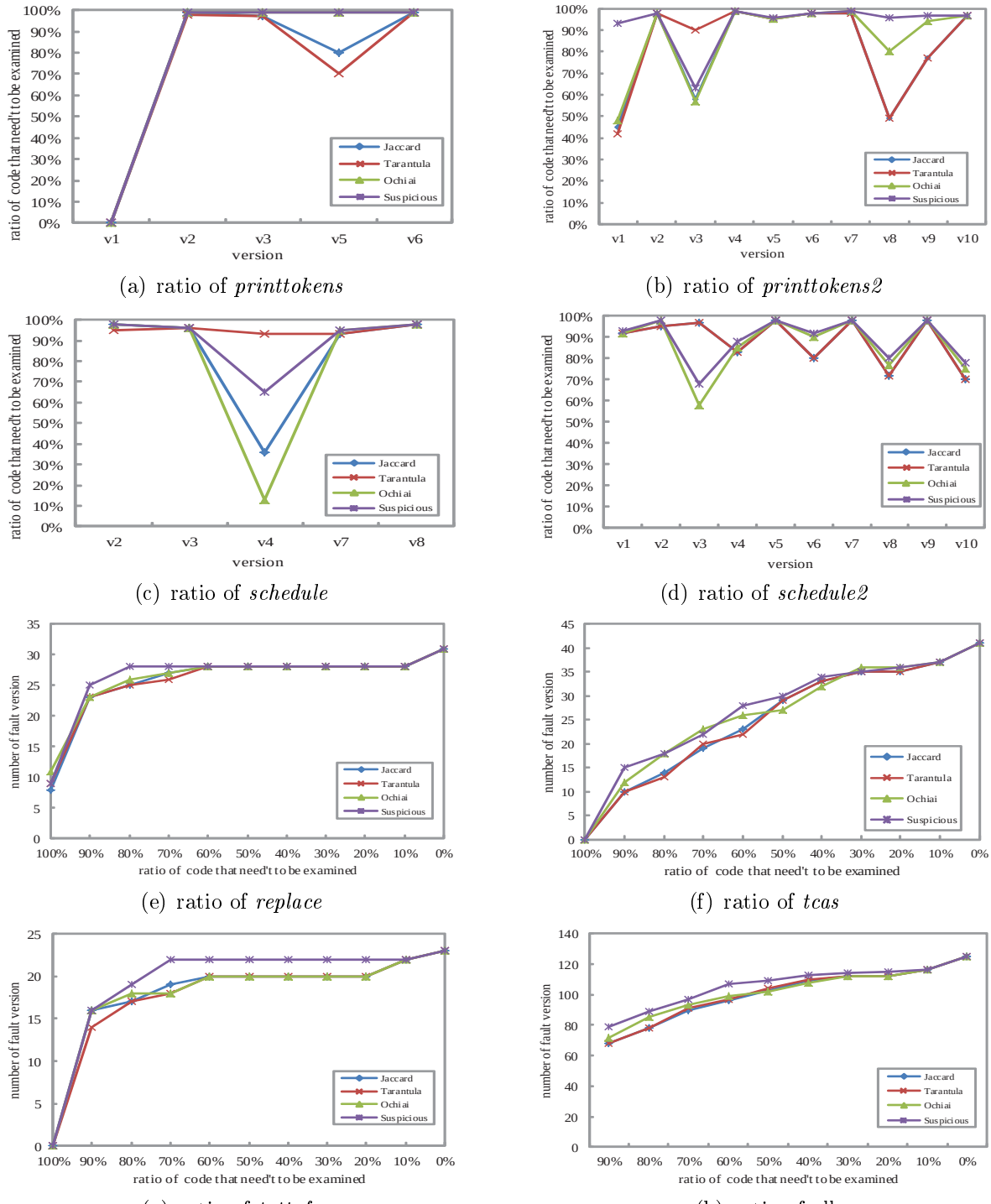
4. **Experiment.** In this section, since *Jaccard*, *Tarantula* and *Ochiai* are efficient statistical measures to statistical fault localization, we compare the performance of the proposed *Suspicious* with them. These algorithms are implemented in Java. Experiments are conducted on 64 bit Windows 7 system, Xeon CPU E5-2603 @1.80GHz, 8G Memory.

4.1. **Experimental data sets and parameter setting.** We use Siemens benchmark test suite as our experimental data set, which is widely used as the experimental subject in the field of software research. Siemens contains seven programs and each program has some bug versions which are seeded artificially. Siemens are described in detail, shown in Table 1. Because faults in some versions appear in header file, we ignore them. Versions 4 and 6 of *printtokens* are not considered. Since $totalfailed = 0$, version 1, 5, 6, 9 of *schedule2* and version 32 of *replace* are ignored in our experiment. We use 125 versions in our experiment finally. Since the scale of seven programs in Siemens is small, some codes are directly added to all basic blocks to form instrumented software. The given support is close to the percentage of failing executions and all executions in our experiments.

TABLE 1. Siemens benchmark test suite

| Program | Faulty versions | LOC | Basic blocks | Test cases | Description |
|---|---|---|---|---|---|
| *printtokens* | 7 | 565 | 107 | 4130 | lexical analyzer |
| *printtokens2* | 10 | 510 | 106 | 4115 | lexical analyzer |
| *replace* | 32 | 563 | 124 | 5542 | pattern recognition |
| *schedule* | 9 | 412 | 55 | 2650 | priority scheduler |
| *schedule2* | 10 | 307 | 60 | 2710 | priority scheduler |
| *tcas* | 41 | 173 | 21 | 1608 | altitude separation |
| *totinfo* | 23 | 406 | 45 | 1052 | information measure |

4.2. **Result and performance analysis.** Three performance metrics are used. The first metric is the ratio of codes which need not be examined until finding fault in the whole executable codes. If the ratio is greater, the fault localization is more effective. The second metric is the suspicious value index of feature node that contains faults. If the index is smaller, it refers *Suspicious* is effective. The third metric is the average ratio of codes which need not be examined in seven programs.




(a) ratio of *printtokens*

(b) ratio of *printtokens2*

(c) ratio of *schedule*

(d) ratio of *schedule2*

(e) ratio of *replace*

(f) ratio of *tcas*

(g) ratio of *totinfo*

(h) ratio of all

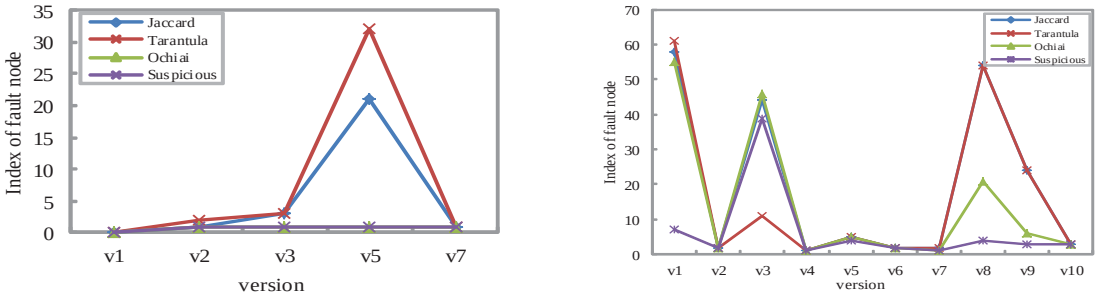
FIGURE 6. The ratio of codes that need not be examined on Siemens

Each figure in Figure 6 shows the ratio of codes which need not be examined respectively. Since the number of versions in *printtokens*, *printtokens2*, *schedule* and *schedule2* is less, the performance of first metric is displayed by recording the ratio of codes which need not be examined in each version in Figures 6(a), 6(b), 6(c) and 6(d). While *replace*, *tcas* and *totinfo* have more versions, we demonstrate the version number of subsection statistics according to the ratio of codes which need not be examined in Figures 6(e), 6(f) and 6(g). The purpose of subsection statistics is to facilitate the comparison. Every ten percentage points form a subsection. $[99\%, 100\%]$ is the first section; it is impossible that the ratio is 100%. However, if the ratio is larger than 99%, faults are found by checking the code less than 1%. It has a good practical significance. In order to compare the efficiency, the numbers of the front subsections are merged. Figure 6(h) implements the version number of all the subsection statistics according to the ratio of codes which need not be examined in seven programs.

From Figure 6(a) to Figure 6(d), it is clear that *Suspicious* in half versions is better than *Jaccard*, *Tarantula* and *Ochiai*, while in other versions *Suspicious* is better than at least one method of *Jaccard*, *Tarantula* and *Ochiai*. Most faults can be located. However, because support is greater, faults in some versions may not be found, such as *v1* in *printtoken*. To solve the problem, the support can be decreased. In Figure 6(e), if all faults except that the ratio is 0% are checked out, *Suspicious* need not check the code size of 80%, while *Jaccard*, *Tarantula* and *Ochiai* need not check 60% code size. Likewise, *Suspicious* need not check the code size of 70%, while *Jaccard*, *Tarantula* and *Ochiai* need not check 60% code size in Figure 6(g). In Figure 6(f), we can find that *Suspicious* is slightly better than *Jaccard*, *Tarantula* and *Ochiai*. In Figure 6(h), when the ratio is 90%, *Suspicious* can find 79 faults of the 125 ones, while *Jaccard* and *Tarantula* only 68 faults, *Ochiai* only 72 faults. By observing the trend throughout the Figure 6(h), *Suspicious* is better than *Jaccard*, *Tarantula* and *Ochiai* with the increasing ratio. So it proves that *Suspicious* is better than *Jaccard*, *Tarantula* and *Ochiai* in locating software faults.

The number of versions is recorded which can be found faults in each index, and the experimental results are shown in Figure 7. The abscissa is the sorted index number of feature node, and the vertical axis is the number of versions. Since *replace*, *tcas* and *totinfo* have more versions, they are not displayed in this paper. Figure 7(a) to Figure 7(d) show the performance of *Suspicious* on *printtokens*, *printtokens2*, *schedule* and *schedule2*. In addition, Figure 7(e) presents the version number of each index whose feature node contains faults in seven programs. It is discovered that the index of half feature nodes is smaller than *Jaccard*, *Tarantula* and *Ochiai* from Figure 7(a) to Figure 7(d) by *Suspicious*, while *Suspicious* in other versions is better than at least one method of *Jaccard*, *Tarantula* and *Ochiai*. In Figure 7(e), 37 faults are found when the index is 1. The size of faults found is greater than 20 when the index is 2. When the index is 3, 5, 8 and 11, the average faults found with *Suspicious* is 5. While the index is greater than 6 except 8 and 11, the faults found are not more than 3. Because the given support is greater, 8 faults are not found. If the index is less than 10, *Suspicious* can find 94 faults of the 125 ones. It indicates that most faults can be located as early as possible using *Suspicious* to calculate the suspicious value.

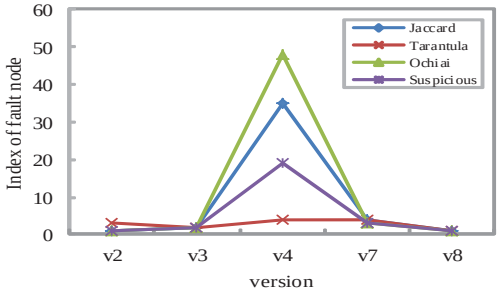
In Figure 8, it presents the average ratio of code which need not be examined in seven programs. It displays that the average ratio of *Suspicious* is larger than *Jaccard*, *Tarantula*, *Ochiai* in *printtoken2*, *replace*, *schedule2*, *tacs* and *totinfo* obviously. *Suspicious* is better than *Jaccard*, *Tarantula*, but equal to *Ochiai* in *printtoken*. *Suspicious* is better than *Jaccard*, *Ochiai*, but worse than *Tarantula* in *schedule*. As a whole, it refers that code which need not be examined is more to find faults using *Suspicious*. Therefore, *Suspicious* can locate faults more effectively, and it can be applied to improve software quality.



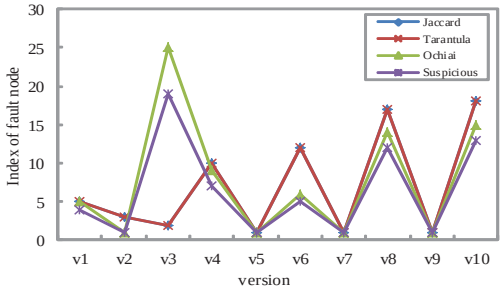

(a) index of *printtokens*



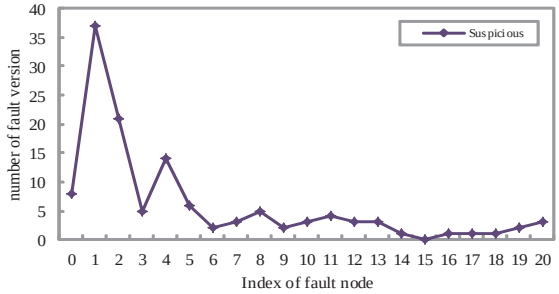

(b) index of *printtokens2*




(c) index of *schedule*




(d) index of *schedule2*




(e) index of all

FIGURE 7. The index of feature node containing faults on Siemens

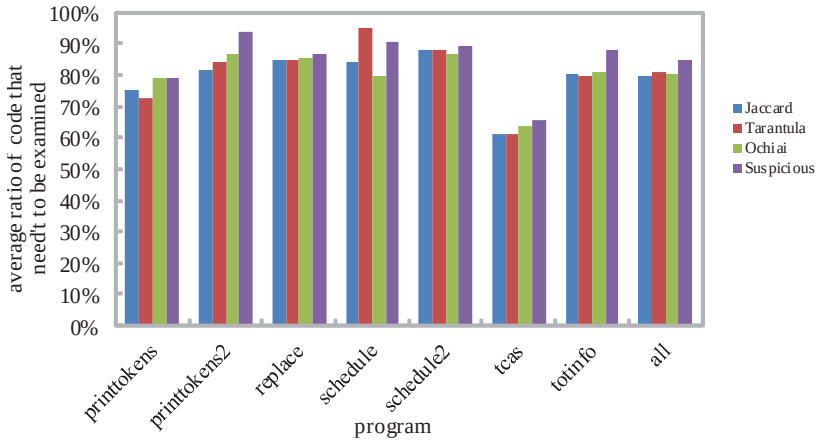



FIGURE 8. Average ratio of codes that need not be examined on Siemens

5. **Conclusions.** In this paper a simplified framework is proposed for fault localization. This framework includes three major phases. (1) Software call traces are obtained on the granularity of functions or basic blocks using a hierarchical instrumentation technique. Then each software call trace is transformed to a dynamic call graph and classified the

passing and failing call trace by analyzing trace structure similarity. The selective instrumentation can enhance the efficiency of software fault localization. (2) A maximal frequent subgraph hierarchical tree (MFSH-Tree) structure is devised. MFSH-TreeMiner algorithm is proposed to mine maximal frequent call subgraphs using MFSH-Tree. And MFSH-TreeMiner algorithm can find frequently executed basic blocks as feature nodes. (3) Taking account of both executions set and complementary set of executions, a measure based on Jaccard is designed to calculate the suspicious value of feature nodes. In order to locate faults quickly, feature nodes are sorted in descending order according to the suspicious value. In the end, experiments on Siemens benchmark test suite show that our fault localization approach is efficient. Therefore, our approach can help developers reduce time to locate faults efficiently for large-scale software.

**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 China under Grant No. F2013203324, No. F2014203152 and No. F2015203326.

## REFERENCES

[1] B. Hailpern and P. Santhanam, Software debugging, testing, and verification, *IBM Systems Journal*, pp.4-12, 2002.

[2] J. Jones and M. Harrold, Empirical evaluation of the Tarantula automatic fault-localization technique, *Proc. of the 20th IEEE/ACM Conference on Automated Software Engineering*, pp.273-282, 2005.

[3] R. Abreu, P. Zoeteweij and A. V. Gemund, Spectrum-based multiple fault localization, *The 24th IEEE/ACM International Conference on Automated Software Engineering*, pp.88-99, 2009.

[4] G. Neelam, H. F. He, X. Y. Zhang and R. Gupta, Locating faulty code using failure-inducing chops, *Proc. of IEEE/ACM International Conference on Automated Software Engineering*, pp.263-272, 2005.

[5] A. Zeller, Isolating cause-effect chains from computer programs, *Proc. of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol.27, no.6, pp.1-10, 2002.

[6] B. Hofer and F. Wotawa, Spectrum enhanced dynamic slicing for better fault localization, *Proc. of the European Conference on Artificial Intelligence*, pp.420-425, 2012.

[7] Y. Lei, X. Mao, Z. Dai and C. Wang, Effective statistical fault localization using program slices, *Proc. of the Computer Software and Applications Conference*, pp.1-10, 2012.

[8] M. Renieris and S. Reiss, Fault localization with nearest neighbor queries, *Proc. of IEEE International Conference on Automated Software Engineering*, pp.30-39, 2003.

[9] L. Guo, A. Roychoudhury and T. Wang, Accurately choosing execution runs for software fault localization, *Proc. of the 15th International Conference on Compiler Construction*, pp.80-95, 2006.

[10] C. Sun and S. C. Khoo, Mining succinct predicated bug signatures, *Proc. of the 9th Joint Meeting on Foundations of Software Engineering*, pp.576-586, 2013.

[11] Z. Zuo, S.-C. Khoo and C. Sun, Efficient predicated bug signature mining via hierarchical instrumentation, *Proc. of the 2014 International Symposium on Software Testing and Analysis*, pp.215-224, 2014.

[12] C. Liu, X. Yan, H. Yu, J. Han and P. S. Yu, Mining behavior graphs for "Backtrace" of noncrashing bugs, *Proc. of the 5th International Conference on Data Mining*, pp.286-297, 2005.

[13] X. Yan and J. Han, CloseGraph: Mining closed frequent graph patterns, *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.286-295, 2003.

[14] F. Eichinger, K. Böhm and M. Huber, Improved software fault detection with graph mining, *Proc. of the 6th International Workshop on Mining and Learning with Graphs*, 2008.

[15] F. Eichinger, K. Böhm and M. Huber, Mining edge-weighted call graphs to localise software bugs, *Proc. of the Machine Learning and Knowledge Discovery in Databases*, pp.333-348, 2008.

[16] H. Cheng, D. Lo, Y. Zhou, X. Wang and X. Yan, Identifying bug signatures using discriminative graph mining, *Proc. of the 2009 International Symposium on Software Testing and Analysis*, pp.141-152, 2009.

[17] X. Yan, H. Cheng, J. Han and P. S. Yu, Mining significant graph patterns by leap search, *Proc. of the 2008 ACM SIGMOD International Conference on Management of data*, pp.433-444, 2008.

[18] S. Parsa, S. A. Naree and N. E. Koopaei, Software fault localization via mining execution graphs, *International Conference on Computational Science and Its Applications*, pp.610-623, 2011.

[19] X. Wang and Y. Liu, Automated fault localization via hierarchical multiple predicate switching, *Journal of Systems and Software*, pp.69-81, 2015.

[20] R. Vijayalakshmi, R. Nadarajan, J. F. Roddick, M. Thilaga and P. Nirmala, FP-GraphMiner: A fast frequent pattern mining algorithm for network graphs, *Journal of Graph Algorithms and Applications*, vol.15, no.6, pp.753-776, 2011.

[21] M. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, Pinpoint: Problem determination in large, dynamic Internet services, *Proc. of the 2002 International Conference on Dependable Systems and Networks*, pp.595-604, 2002.