International Journal of Innovative Computing, Information and Control
Volume **12**, Number **3**, June **2016**

ICIC International ©2016 ISSN 1349-4198
pp. 847–857

MFGP-MINER: MAXIMAL FREQUENT GRAPH PATTERN MINING FOR FAULT LOCALIZATION

Jiadong Ren[1,2], Huifang Wang[1,2,*], Yue Ma[1,2], Hongdou He[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@163.com; hehondou@yeah.net

Received September 2015; revised January 2016

Abstract. *With the increased workload and difficulty in software maintenance, the researches in automatic debug and software fault localization are more significant to improve software quality. This paper presents a simple framework of software fault localization. Firstly, software execution sequences are collected on the granularity level of basic blocks during the software testing phase. These software execution sequences are mapped as directed software execution graphs. Next, Dynamic BitCode (DBC) data structure is constructed by scanning the graph database just once. In order to discover feature nodes with software faults, this paper proposes MFGP-Miner (maximal frequent graph pattern mining) algorithm to mine maximal frequent graph patterns based on Dynamic BitCode (DBC) data structure. Finally, taking account of the executions set and the executions complementary set, a measure based on Ochiai is designed to calculate the suspicious value of feature nodes. These feature nodes are ranked to help programmers to find faults in descending order according to the suspicious value. Siemens benchmark test suite is used in our experiments, and experimental results display that our approach is both efficient and effective for locating software faults.*

Keywords: Software fault localization, Software execution graph, Maximal frequent graph pattern, Dynamic BitCode

1. Introduction.

As we know, the most time is spent on software testing and debugging in software development and maintenance. Software fault localization is the most difficult and time-consuming task in the process of software debugging [1]. Therefore, in order to increase software stability and reduce the amount of time required to debug manually, many researchers have studied automatic and effective techniques for fault localization.

Recently a lot of researches on software fault localization help programmers to locate faults such as model-based methods, statistical spectrum analysis and program slicing. L. Mariani et al. [2] presented a static analysis technique SEIM to extract interaction models. In SEIM, a refinement strategy is included to identify infeasible elements from statically derived models for analyzing service-based applications. With the purpose of locating faults, Z. Zhang et al. [3] introduced a CP model by abstracting program executions as edge profiles. CP captures suspicious propagation of program states and calculates suspiciousness scores of these program states to identify faults. Since CP considers the structural relationships among statements, it is more effective than coverage-based fault localization. However, in order to analyze data dependence, G. K. Baah et al. [4] presented a model called probabilistic program dependence graph (PPDG). The RankCP algorithm is proposed to rank probabilistic statements for fault localization in PPDG. Whereas analyzing data dependence is a time-consuming process, D. Gong et al. [5] proposed a two-step test-suite reduction approach by taking into account coverage information and concrete execution paths. As this approach removes test cases having little effect on fault localization, it is more effective than traditional techniques. In addition, D. Gong et al. [6] put forward an SDPM model to obtain behavior state information. SDPM analyzes the impact of control flow dependence between statements, and distinguishes the state dependencies of program elements in passed and failed test cases. SDPM is an effective model for fault localization even though few test cases are available. By combining spectrum fault localization (SFL) with slicing-hitting-set-computation (SHSC), B. Hofer and F. Wotawa [7] introduced a SENDYS method. Thus, SENDYS enhances the ability of fault localization and solves these disadvantages of SFL and SHSC. In order to locate faults, S. K. Sahoo et al. [8] proposed an automated bug diagnosis technique by combining program invariants, delta debugging and dynamic program slicing. Two filtering heuristic methods are presented to reduce false positives and the number of candidate bug signatures. Wong et al. [9] proposed the DStar(D*) technique based on the Kulczynski coefficient for locating faults. DStar(D*) contributes to identifying multiple fault locations automatically without regard to the prior knowledge of program structure and program semantics. Statistical fault localization approaches need to collect program executions from end-user. However, it is unrealistic that end-users run fully instrumented programs. Thus, Z. Zuo et al. [10] put forward an iterative statistical bug isolation approach based on systematic hierarchical instrumentation (HI). Coarse-grained pruning and ranking measures are proposed in this approach. The HI technique prunes some unnecessary instrumentation and decreases the performance cost of end-users. In order to locate complex faults, X. Wang and Y. Liu [11] presented a hierarchical multiple predicate switching (HMPS) method based on instrumentation method and switching combination strategies. By utilizing spectrum-based fault localization techniques, HMPS narrows the scope for identifying critical predicates.

Software executions can be transformed to call graphs. Analyzing software execution graph can find faults early and reduce software maintenance cost. H. Cheng et al. [12] transformed program execution paths to software behavior graphs. In correct and false software behavior graphs, the Top-K LEAP algorithm is proposed to extract the most discriminative subgraphs as bug signatures based on LEAP algorithm [13]. Nevertheless, Top-K LEAP does not consider the influence from the occurrences number of subgraph on fault localization. Thus, S. Parsa et al. [14] presented a discriminative function F-Scored to calculate discriminative value of subgraphs on weighted graph instead of un-weighted graph. F-Scored has higher precision and recall than RAPID and Top-K LEAP. Despite all this, some less frequent subgraphs may be obtained by mining discriminative subgraph. For solving this issue, F. Eichinger et al. [15, 16] proposed a software fault localization framework using closeGraph [17]. This framework generates software call graphs by reduction techniques and combines information entropy and structure score to improve the accuracy of fault localization. However, above methods only apply to single-threaded programs rather than multithreaded programs. In order to locate faults in multithreaded programs, F. Eichinger [18] introduced a general method for defect localization by reducing program executions to call graphs. They calculate the defective probability of methods. Those methods are ranked according to the probability for presenting to developers. Tree also can represent software execution process. Therefore, Narouei et al. [19] developed a heuristic method DLLMiner based on DLL dependency tree. DLLMiner can extract coarse-grain behavior features by mining closed frequent subtrees. These behavior features are regarded as effective fault signatures.

Previous works use closeGraph algorithm to locate faults. However, the closeGraph is not suitable for large-scale software. In the FP-GraphMiner algorithm [20], *BitCode* is proposed to mine frequent subgraphs by scanning graph databases once. However, *BitCode* of an edge is a fixed size (equal to number of graphs in graph databases). It leads to expending more memory for storing *BitCode* of edges and the time for computing the frequency of edges and subgraph. Thus, we introduce *Dynamic BitCode* data structure for storing edges with an unfixed size by scanning graph databases once in the whole mining process. This paper computes the frequency which is obtained by only computing the number of bits 1 in *Dynamic BitCode*. We put forward an MFGP-Miner algorithm based on *Dynamic BitCode* data structure. Advantages of this algorithm are based on the intersection between two subgraphs for fast computing the support. Ochiai similarity coefficient [21] is used to calculate the suspicious value which only supposes that faults appear at the failing executions. Nevertheless, the passing executions may not appear faults. By taking into account these two cases, we define a measure to calculate the suspicious value of feature nodes for locating faults accurately and quickly.

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

- In order to analyze program executions, we obtain software execution sequences at the granularity level of basic blocks. We adopt a graph representation for locating faults easily that avoids repeated substructures of software execution sequences.
- Dynamic BitCode data structure is devised to store graph information by visiting graph database only once. We construct a potential maximal frequent graph pattern tree (PMFGP-Tree) structure based on Dynamic BitCode data structure. By traversing PMFGP-Tree, an efficient maximal frequent graph pattern mining (MFGP-Miner) algorithm is presented to find all maximal frequent graph patterns.
- As Ochiai only considers the executions set to calculate suspicious values, a measure is designed by taking account of the execution set and complementary execution set.

The remaining of the paper is organized as follows. Section 2 describes some definitions and formulas. Section 3 presents a framework of fault localization with maximal frequent graph pattern mining and a measure for calculating the suspicious value. Experiments and analysis are shown in Section 4. Section 5 summarizes the paper.

2. **Preliminary Concepts.**

A software execution graph g is denoted as $g = (V, E)$ corresponding to a software execution sequences, 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 represent program entities during program are executed. The edges represent call relationships among program entities respectively. If a program entity e_i calls the

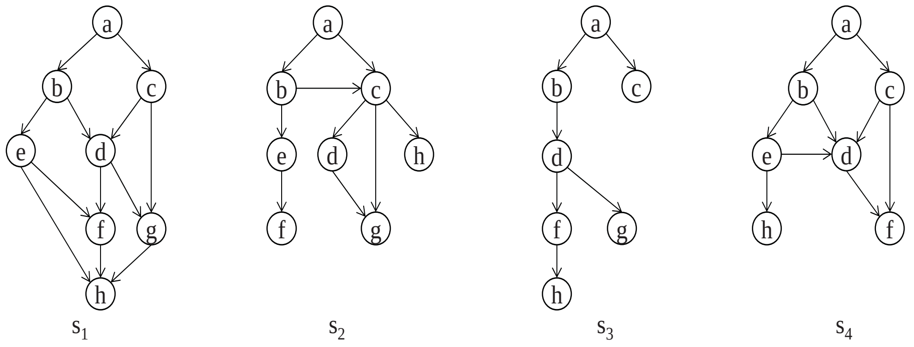


FIGURE 1. A sample of graph database

program entity e_j, there is a directed edge from e_i to e_j. All edges are unique in a software execution graph. Program entities contain functions, basic blocks and statements.

A maximal frequent graph pattern can be defined as follows. Let D be a graph database and $minSup$ is a minimum support threshold. The support of a graph pattern g, denoted as $\sigma(g)$, is the percentage of graphs in D containing g and D. g is a frequent graph pattern if $\sigma(g) \geqslant minSup$. Formally, a graph g is called a maximal frequent graph pattern if it is frequent, and it does not have any frequent graph pattern g_1 such that $g \subseteq g_1$.

As described above, $BitCode$ of each graph pattern or edge has the fixed size $|D|$, so it takes up more memory and time. In fact, if $BitCode$ of a graph pattern or an edge contains many bits 0, it can be simplified to reduce space and time. Therefore, *Dynamic BitCode* data structure is proposed to solve the problem.

Definition 2.1. *Dynamic BitCode (DBC). DBC of an edge e is* $DBC(e) = \{index, BitCode\}$, *where index is the position of the first nonzero bit in BitCode, and BitCode is a bit string in BitCode after removing 0 from first position to index and from the position where the last bit is 1 in BitCode to last position. DBC of a graph pattern likes this.*

Definition 2.2. *Frequency of Edge. Frequency of an edge e denoted as* $fre(e)$ *is the number of 1 in BitCode of DBC. Similarly, frequency of a graph pattern can be defined.*

The DBCPattern structure combines a DBC structure with a representation of graph information. Each DBCPattern consists of two parts: subgraph information and a DBC. A DBC represents the positions of the subgraph which appears in the graph database.

Pattern-Extension (DBC_1, DBC_2): we start to do AND operation between DBC_1 and DBC_2 from the index which is the greater index value in two index values of them. If the result is 0, *index* is increased by one until attaining the first nonzero value. We then start to do AND operation for each bit of $BitCode$ from this index until the result of the rest bits is 0. A new DBC is obtained.

Example 2.1. *As shown in Figure 1,* $BitCode(bd) = 1011$, $BitCode(cf) = 0001$, $DBC(bd) = \{0, 1011\}$, $DBC(cf) = \{3, 1\}$, $fre(bd) = 3$. *Assume that we do AND operation between* $DBC(bd) = \{0, 1011\}$ *and* $DBC(cf) = \{3, 1\}$. *Because index* $1 > 0$, *it implies index = 1. If index = 1, we have* $0\&1 = 0$, *so index = 2. In the same way,* $1\&0 = 0$, $1\&1 = 1$, *so index = 3. Next, new DBC is* $\{3, 1\}$.

Definition 2.3. *Potential Maximal Frequent Graph Pattern Tree (PMFGP-Tree). The potential frequent graph patterns are stored in PMFGP-Tree, which is a multi-tree. The node of the tree contains three parts, edge list, frequency and node-link, where all edges in edge list constitute a graph pattern, the frequency is the frequency of the graph pattern, node-link is a pointer to its child node. The root node of the tree is defined by* ϕ.

TP and TF are defined as the passing and failing executions respectively. Given a feature node n in software, $P(n)$ and $F(n)$ represent the number of n existing in the passing and failing executions respectively; $NP(n)$ and $NF(n)$ represent the number of n not existing in the passing and failing executions respectively.

Ochiai is used to calculate the suspicious value for fault localization [21], given in Equation (1).

$$Ochiai(n) = \frac{F(n)}{\sqrt{(F(n) + NF(n)) \cdot (F(n) + P(n))}}. \tag{1}$$

Since the passing executions may not execute feature node that leads to a fault, a coefficient is defined to compute the suspicious value of feature nodes, given in Equation

(2).

$$Un\_Ochiai(n) = \begin{cases} 1 & NF(n) = 0, \\ \dfrac{NP(n)}{\sqrt{(P(n) + NP(n)) \cdot (F(n) + P(n))}} & \text{other.} \end{cases} \tag{2}$$

A measure called *Suspicious* is devised to calculate the suspicious value of feature nodes by combining execution sets with execution complementary set, given in Equation (3).

$$Suspicious(n) = \frac{Ochiai(n) + Un\_Ochiai(n)}{2}. \tag{3}$$

3. A Framework of Locating Software Fault.

This framework of software fault localization which is described in Algorithm 1 has three phases mainly. First, software execution sequences are collected during software executing. These software execution sequences have a wealth of information for locating software faults. Every software execution sequence is assigned a label (passing, failing), which is determined by comparing and analyzing sequence structure similarity. As software execution sequences are too long, they are reduced to construct software execution graphs. Next, for a given support, the maximal frequent graph pattern mining (MFGP-Miner) algorithm is used to find the features which may contain faults. Finally, a measure to calculate the suspicious value of feature nodes is proposed to help developers to locate faults quickly.

Algorithm 1 A Software Fault Localization Framework

```
Input: software execution sequences $S$, support $\sigma$, graph database $D$
Output: Feature node $n$ that contains faults
 1: $D = \phi$
 2: for each trace $s \in S$ do
 3:     assign a label (passing, failing) to $s$;
 4:     transform $s$ to a software execution graph $g$;
 5:     $D = D \cup g$;
 6: end for
 7: Call MFGP - Miner($D$, $\sigma$);
 8: calculate the suspicious value of all feature nodes;
 9: descending sort feature nodes by suspicious value;
10: find feature node $n$ that contains faults;
11: return $n$;
```

Since each edge is distinct for a software execution graph, a graph database is represented as edge list EL. Each edge is expressed as $< e, DBC(e) >$ in EL. EL is sorted in descending order according to the DBC of edge. Edges that satisfy $fre(edge) \geq \sigma \cdot |D|$ are selected from EL as follows.

Algorithm 2 shows the pseudo code of MFGP-Miner algorithm in the framework. The algorithm first scans the database D only once, and D is represented as edge list EL where each edge is expressed as $< e, DBC(e) >$ (lines 2-6). The list is sorted in descending order according to the DBC of edge to reduce the steps in the extension phase (line 7). Then the algorithm finds frequent patterns with an edge and then stores them in PMFGP-Tree as child nodes of the root (line 8). Then, according to the child nodes of the root, DBC-Pattern-Extension algorithm (Algorithm 3) is called to construct PMFGP-Tree (lines 9-11). All frequent graph patterns can be obtained by traversing PMFGP-Tree (lines 10-13).

Algorithm 2 MFGP-Miner Algorithm

```
Input: graph database $D$, support $\sigma$
Output: All maximal frequent graph patterns $F$
 1: $F = \phi$, PMFGP-Tree.root = $\phi$, $DEL = \phi$;
 2: for each distinct edge $e_i \in D$ do
 3:     if $fre(e_i) \geqslant \sigma \cdot |D|$ then
 4:         insert $< e_i, DBC(e_i) >$ into $EL$;
 5:     end if
 6: end for
 7: descending sort $EL$ by frequency;
 8: add $EL$ to child node of PMFGP-Tree.root;
 9: Call DBC-Pattern-Extension($subNode$, $\sigma$);
10: if PMFGP-Tree.root.getChildren() != null then
11:     Call TraverseTree(PMFGP-Tree, PMFGP-Tree.root, $F$);
12: end if
13: return $F$;
```

Algorithm 3 DBC-Pattern-Extension(root, minSup)

```
Input: node $N$, support $\sigma$
Output: A set of maximal frequent graph patterns root
 1: Let $nodelist$ = child node of $N$;
 2: for each $n_i$ in $nodelist$ do
 3:     for each $n_j$ in $nodelist$, $i+1 \leqslant j \leqslant |nodelist|$ do
 4:         if $fre(n_{ij}$ = Pattern-Extension($n_i$,$n_j$))$\geqslant \sigma \cdot |S|$ then
 5:             add $n_{ij}$ to child node of $n_i$;
 6:         end if
 7:     end for
 8:     Call DBC-Pattern-Extension($n_i$, $\sigma$);
 9:     if $n_i$.getChildren() != null then
10:         $n_i.label$ = non-max;  // $n_i$ is not the maximal frequent graph pattern
11:     end if
12: end for
```

Algorithm 3 shows that DBC-Pattern-Extension algorithm is called by the MFGP-Miner algorithm. If the frequency of extended frequent graph pattern using Pattern-Extension approach is more than $\sigma \cdot |D|$, the extended pattern is a child node of n_i. The process executes recursively (line 12) until no maximal frequent graph patterns are generated. If the node has children, it will be not a maximal frequent graph pattern, so the label of the node is set to non-max.

The process of traversing PMFGP-Tree is presented in Algorithm 4. PMFGP-Tree is traversed by DFS strategy to find all maximal frequent graph patterns. If the label of the node is non-max, we will call the TraverseTree algorithm recursively; otherwise, the node is a maximal frequent graph pattern and is added to F.

The MFGP-Miner algorithm uses DBC data structure and graph information to mine maximal frequent graph patterns. The MFGP-Miner algorithm is split into two main phases: (1) the graph database is converted into DBCPattern structure, where each DBC-Pattern stores the positions of frequent graph patterns appearing in the graph database; (2) frequent graph patterns are generated and verified, and graphs which are not satisfied

Algorithm 4 TraverseTree(PMFGP-Tree, root, F)

```
Input: PMFGP-Tree, node $N$, $F$
Output: All maximal frequent graph patterns set $F$
1: if $N$.getChildren() != null then
2:     for each child $c_i \in N$.getChildren() do
3:         if $c_i.label$ != non-max then
4:             add $c_i$ to $F$;
5:         end if
6:         Call TraverseTree(PMFGP-Tree, $c_i$, $F$);
7:     end for
8: end if
9: return $F$;
```

frequency threshold are pruned early. The MFGP-Miner algorithm scans the graph database only once and calculates the frequency based on the *DBC* to generate new patterns. Because of using the compacted structure, the MFGP-Miner algorithm is more efficient to mine maximal frequent graph patterns in terms of runtime and memory usage.

All feature nodes can be obtained in maximal frequent graph patterns. We calculate the suspicious value of feature nodes by using *AllOchiai* and rank these feature nodes in descending order according to the suspicious value. Software developers can find faults by analyzing source code and these feature nodes.

4. Experiment.

In this section, our proposed *Suspicious* compared the performance with *Ochiai*, *Jaccard* and *Tarantula* which are efficient statistical measures to statistical fault location. Java is employed to implement these algorithms. We run all experiments on 64 bit Windows 7 system, Xeon CPU E5-2603 @1.80GHz, 8G Memory.

4.1. Experimental data sets and parameter setting.

Since a lot of methods use Siemens benchmark test suite as experimental subject in previous researches of fault localization, we also apply Siemens benchmark test suite as our experimental data set. Siemens contains seven programs and each program has a correct version, a number of wrong version and test cases. Table 1 provides an overview about relevant information of programs in Siemens benchmark test suite. As some predefined faults appear in header file and some faults are not detected by the available test cases in some version, we ignore them. Thus, versions 4, 6 of *printtokens* version 1, 5, 6, 9 of *schedule2* and version 32 of *replace* are not considered. We employ 125 versions in our experiments at last. In our experiments, the support is close to the percentage of failing executions and all executions.

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

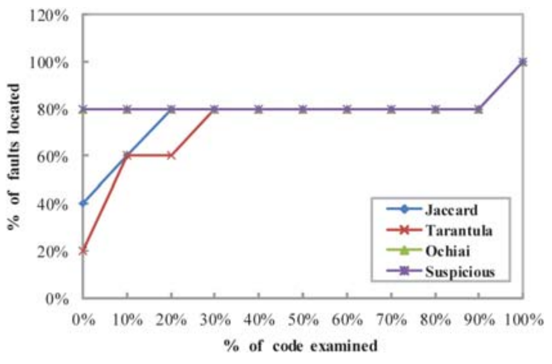
4.2. **Result and performance analysis.** We use two metrics to evaluate the performance of our approach. The first metric is the percentage of codes which is examined until finding fault in the whole executable codes. If the percentage of faults which are located is more, it means that the approach is clearly more effective for software fault localization. The second metric is the average percentage of codes which need not be examined in seven programs. If the average percentage of code examined is less, it refers that *Suspicious* is more efficient to locate faults. Description and analysis are shown as follows.

Each figure in Figure 2 compares the fault localization efficiency of four approaches for seven programs respectively. The abscissa represents the percentage of code that is examined. The ordinate represents the percentage of faults which are located among amount of code. In order to facilitate the comparison, we demonstrate the percentage of faults using subsection statistics. Every ten percent constitute a subsection. [0%, 1%] is the first subsection. It is impossible that the percentage is 0%, whereas if the percentage is less than 1%, faults can be quickly found. Thus, [0%, 1%] has a good practical significance. The purpose of combining the numbers of front subsections is to compare the efficiency clearly. Figure 2(h) implements the percentage of faults for all the subsection statistics according to the percentage of codes which is examined in seven programs.

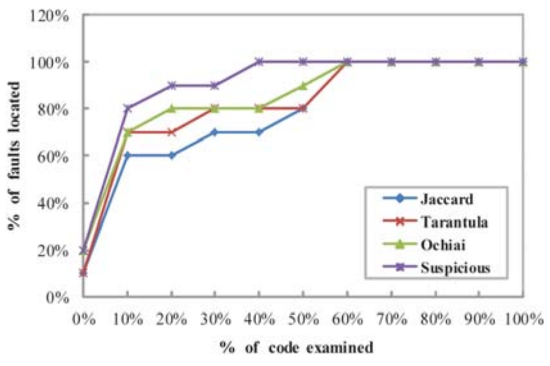
In our experiments, almost all faults can be localized. Since the given support is greater, we cannot find some faults in some versions unless we check all code. In order to solve the above problem, we can reduce the support. We can discover that the results of *Suspicious* are superior to *Ochiai*, *Jaccard* and *Tarantula* in Figures 2(b), 2(e), 2(f), 2(g). For example, we only examine the code size of 40% to find 80% faults in Figure 2(f). As shown in Figure 2(g), if all faults are checked out unless few faults need to examine all code, *Suspicious* only need to check the code size of 40%, while *Ochiai*, *Jaccard* and *Tarantula* require to check 90% code size. In Figure 2(d), we can find that *Suspicious* is slightly better than *Ochiai*, *Jaccard* and *Tarantula*. We use a more comprehensive view to present the performance of our approach in Figure 2(h). By observing and analyzing the trend of Figure 2(h), we can draw a conclusion that *Suspicious* is better than *Ochiai*, *Jaccard* and *Tarantula* with the increasing percentage. Figure 2 proves that *Suspicious* is better than *Ochiai*, *Jaccard* and *Tarantula* in locating software fault. That is because *Suspicious* considers that feature node containing faults may not be executed in passing executions; nevertheless *Ochiai*, *Jaccard* and *Tarantula* ignore this case.

Table 2 shows the average percentage of code that must be examined in all versions of every program by using different fault localization approaches. This table presents that the performance of *Suspicious* is better than *Ochiai*, *Jaccard* and *Tarantula* in *printtoken2*, *replace*, *schedule2*, *tacs*, and *totinfo* obviously. *Suspicious* is larger than *Jaccard*, *Tarantula*, but as good as *Ochiai* in *printtoken*. And *Suspicious* is superior to *Jaccard*, *Ochiai*, but slightly poorer than *Tarantula* in *schedule*. In general, it refers that *Suspicious* performs better than *Ochiai*, *Tarantula* and *Jaccard* to find faults. Therefore, it shows that using *Suspicious* can improve the capability of fault localization, and it can be applied to improve software quality.

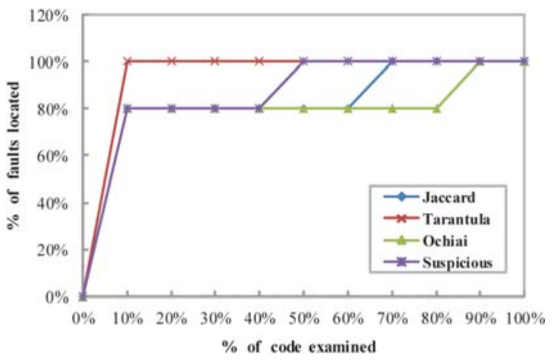
5. **Conclusion.** In this paper, we propose a software fault localization framework. This framework contains three main sections. (1) Software execution sequences are obtained at the granularity level of basic blocks and categorized as the passing and failing execution sequences by analyzing and comparing sequence structure similarity. Each software execution sequence is reduced to a software execution graph for compact representation of program execution. (2) A potential maximal frequent graph pattern tree (PMFGP-Tree)



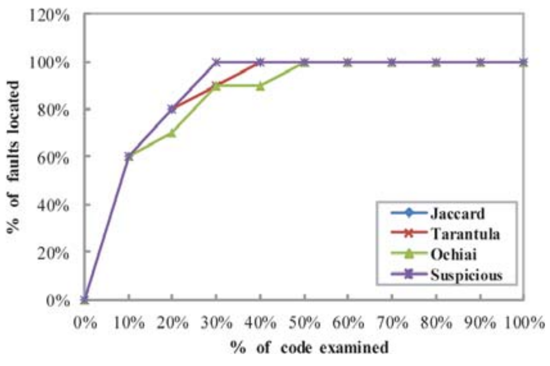
(a) The percentage of *printtokens*



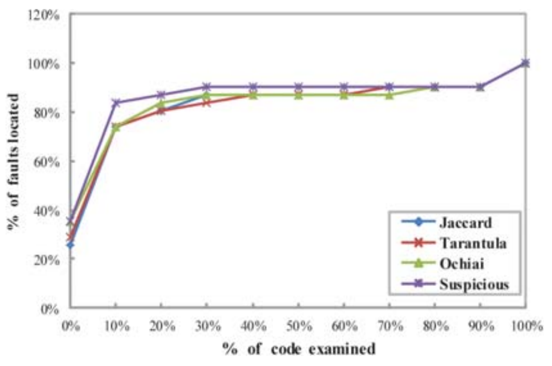
(b) The percentage of *printtokens2*



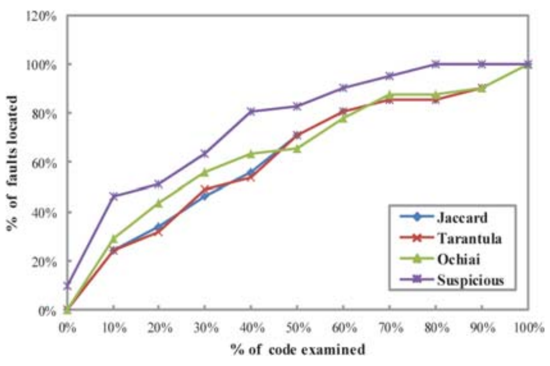
(c) The percentage of *schedule*



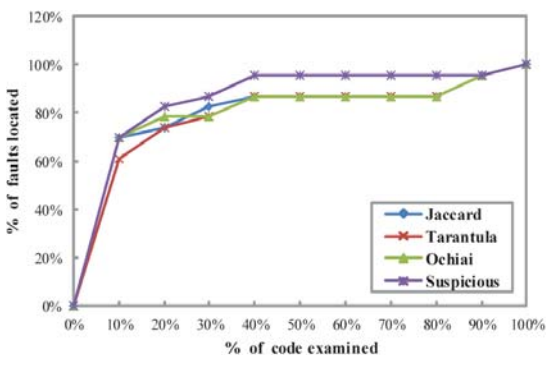
(d) The percentage of *schedule2*



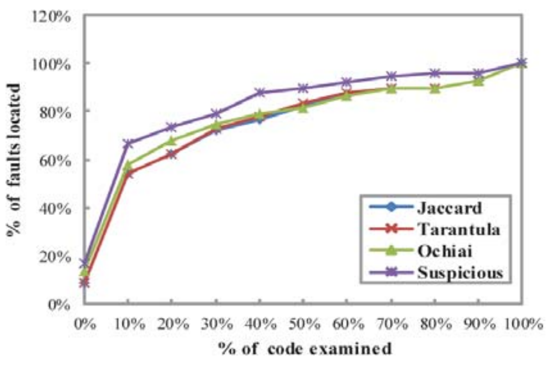
(e) The percentage of *replace*



(f) The percentage of *tcas*



(g) The percentage of *totinfo*



(h) The percentage of all

FIGURE 2. The percentage of codes examined on Siemens

Table 2. The average percentage of code checked on Siemens

Program	Suspicious	Ochiai	Jaccard	Tarantula
printtokens	20.80%	20.80%	25.00%	27.20%
printtokens2	7.60%	13.50%	18.60%	15.70%
replace	13.77%	14.26%	15.00%	15.29%
schedule	11.00%	20.00%	15.80%	5.00%
schedule2	11.00%	12.60%	11.20%	11.20%
tcas	26.12%	36.10%	38.54%	38.93%
totinfo	12.78%	18.91%	19.65%	20.43%
all	14.72%	19.45%	20.54%	19.11%

is devised by employing *Dynamic BitCode* data structure which is generated by accessing graph database only once. The MFGP-Miner algorithm is proposed to mine maximal frequent graph patterns based on PMFGP-Tree. Basic blocks which are frequently executed are found as feature nodes using MFGP-Miner. (3) Based on *Ochiai*, a measure is designed to calculate the suspicious value of feature nodes by taking into consideration both executions set and complementary set of executions. These feature nodes are sorted in descending order according to the suspicious value for locating faults rapidly and accurately. At last, experimental results demonstrate that our approach is efficient on Siemens benchmark test suite. Therefore, our approach can facilitate software developers to locate faults efficiently and reduce their debugging time 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. Authors are grateful to receive the helpful comments and suggestions of the reviewers.

REFERENCES

[1] I. Vessey, Expertise in debugging computer programs, *Information Systems Working Papers Series*, pp.459-494, 1985.

[2] L. Mariani, M. Pezze, O. Riganelli and M. Santoro, SEIM: Static inference of interaction models, *Proc. of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, pp.22-28, 2010.

[3] Z. Zhang, K. Wing, T. Tse, J. Bo and X. Wang, Capturing propagation of infected program states, *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.43-52, 2009.

[4] G. K. Baah, A. Podguiski and M. J. Harrold, The probabilistic program dependence graph and its application to fault diagnosis, *IEEE Trans. Software Engineering*, pp.528-545, 2010.

[5] D. Gong, T. Wang, X. Su and P. Ma, A test-suite reduction approach to improving fault-localization effectiveness, *Computer Languages, Systems & Structures*, pp.95-108, 2013.

[6] D. Gong, X. Su and T. Wang, State dependency probabilistic model for fault localization, *Information and Software Technology*, pp.430-445, 2015.

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

[8] S. K. Sahoo, J. Criswell, C. Geigle and V. Adve, Using likely invariants for automated software fault localization, *ACM SIGARCH Computer Architecture News*, pp.139-151, 2013.

[9] W. E. Wong, V. Debroy, Y. Li and R. Gao, Software fault localization using DStar (D*), *Proc. of the IEEE 6th International Conference on Software Security and Reliability*, pp.21-30, 2012.

[10] Z. Zuo and S. C. Khoo, Iterative statistical bug isolation via hierarchical instrumentation, *Technial Report TRV7/14*, School of Computing, National University of Singapore, 2014.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] F. Eichinger, V. Pankratius and K. Böhm, Data mining for defects in multicore applications: An entropy-based call-graph technique, *Concurrency and Computation: Practice and Experience*, pp.1-20, 2014.

[19] M. Narouei, M. Ahmadi and G. Giacinto, DLLMiner: Structural mining for malware detection, *Security and Communication Networks*, 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] D. Hao, L. Zhang, Y. Pan et al., On similarity-awareness in testing-based fault localization, *Automated Software Engineering*, pp.207-249, 2008.