Software Fault Localization via Mining Program Dynamic Execution Graph

Kai ZHANG
Ke YANG
Lianbo ZHOU
Jiadong REN

1 College of Information Science and Engineering, Yanshan University, Qinhuangdao 066004, China
2 The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province, Qinhuangdao 066004, China

Abstract

This paper is concerned with the problem of locating the code area related to software potential fault quickly and accurately in software testing period. A new method Sig BB based on graph model is proposed for mining the suspicious fault nodes from the passing and failing execution graphs. Representing each execution of a program as a graph, the graphs are divided into the passing and failing sets. By extracting the most representative passing and failing graphs based on these sets, the discriminative sub-graph is mined between the two representative graphs. First, Sig BB searches the max common graph, and then gets the opposite nodes set. The discriminative sub-graph is obtained by organizing and extending the set finally. Since the detected code scale is associated with the sorting of suspicious nodes, a suspicious metric strategy is also designed to sort the nodes in the discriminative sub-graph. Experimental results indicate that our method is both effective and efficient for software fault localization.

Keywords: Software Fault Localization; Discriminative Sub-graph; Sub-graph Isomorphism; Software Dynamic Execution

1 Introduction

Software faults are mostly non-crashing and compiled in the software testing phase. The fault contains all kind of types, such as missing of initialization, incorrect variable assignment, logic design fault and so on. Such software faults are not easy to be detected. If you analyze and locate them artificially, this process will be a hard work for developers, and the result of localization is not accurate. So a lot of studies begin to apply statistics-based methods in fault localization, people want to establish a heuristic model through a lot of test cases which describes the feature of spectrum for program dynamic behavior and calculates the suspicious code fragments associated with an error automatically [6, 7].

*Project supported by the National Natural Science Foundation of China (No. 61170190) and the Natural Science Foundation of Hebei Province P. R. China (No. F2012203062, No. F2013203324 and No. F2014203152).
*Corresponding author.
Email address: zhk198929@163.com (Jiadong REN).
Some researchers began to use graph structure demonstrating software dynamic execution behavior in the software testing process. In the graph, the node can be expressed as a function, a basic block or a predicate, the edges as the calling relationships among the code units. Though extracting and contrasting correct and incorrect dynamic execution graphs, a discriminative subgraph is mined between them in order to find the fault location. A few researchers mine frequent subgraphs as suspicious code region [2, 9]. But the frequent subgraphs often do not provide effective information to find the difference between pass and fail execution graphs, thus they can not express the discriminative subgraph related program errors properly. Actually the frequent subgraph based on statistical technology has not a obvious relationship with discriminative subgraph of errors, but certain faulty types can lead that some subgraphs emerge frequently, the appropriate use of frequent subgraph is critical.

Applications of graph mining to locate software bugs have been explored by a lot of researchers, the program execution paths are represented as the network graph from which the reliability of the software is analyzed and assessed [5]. Some firstly append the edge weight between the basic unit node that is the times of calls, then analyze and optimize suspicious subgraphs [1]. In order to find the discriminative subgraph among frequent subgraphs, some researchers start adding information gain measures into above approaches and use efficient search algorithm to find feature sub-graph in frequent subgraphs [3]. However, this feature subgraph is based on mining frequent subgraphs, the result subgraphs sometimes are not associated with errors strongly.

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

- A discriminative subgraph mining algorithm is proposed to find the difference between pass and fail execution graphs and the graph model with a basic block granularity is established for the dynamic execution of software.
- The node weights on graph model are designed and the sorting of suspected nodes is optimized though appending artificially weights on them.

The rest of the paper is organized as follows. Section 2 describes the related work; Section 3 gives some definitions; Feature subgraph mining algorithms and node weights computing method are discussed in Section 4; Experiment and analysis are presented in Section 5; Section 6 concludes our study.

2 Related Work

Methods based on statistical techniques focused on the analysis of program characteristics spectrum in code block units. Researchers proposed predicate count-based spectrum [6, 7], program invariants hit-based spectrum [4, 10] and method calls sequence hit-based spectrum [8] using different program spectrum models and calculation equations. Program statements covering count in a large number of test cases is as a basis for this statistical technology, and then sort the various objects covered on the degree of suspicious though the program spectrum model. This theoretical principles of technology are as following rules: the more a covered program statement appears in the pass test cases, then the less likelihood it contains errors; the less a covered program statement appears in the fail test cases, then the more likelihood it contains errors; a covered program statement appears in the fail a lot, rarely in the pass, then it indicates the possibility of error is high. This technique relies on good test cases to cover all possible program execution paths, and
spends a lot of time to judge the detected object whether to be covered in the course of each test for the followed spectrum calculation.

Graph mining-based methods were proposed for fault localization recently, the execution sequence of software test case is exchanged into the edge-weight graphs at first, comprehensive measurements are considered including the statistical counting and the structure strategy, then suspicious functions are discovered as the candidate of the fault function [3]. Top-K LEAP Search [11] provides a frequent subgraph approach from the pass and fail test cases. The strength of differences is calculated for subgraphs via information gain [1]. Although this method can reduce some frequent subgraphs as the discriminative subgraphs successfully, the obtained sub-graphs are incomplete because some unfrequent subgraphs may also indicate certain discriminative subgraph. The possible feature subgraphs are considered more comprehensively [9], statistical technique is used to evaluate f-score rating of subgraphs which eventually separated the discriminative subgraphs from previous subgraphs. This method is abandoned frequent subgraph measure, trying to find feature subgraphs at the greatest extent.

3 Preliminary Concepts

3.1 Basic definitions

The unit of granularity can be set to statement, predicate, basic block, function in software dynamic execution graph. Statements and predicates are strong expression for more specific information, although this makes the localizing more precisely on certain types of errors, it requires developers to detect a lot of specific and isolated code information. The code size in a function block is uncertain, although it tells developers wrong location identified in a wide range, the region of errors will be found uneasily in a mass function blocks actually. And sometimes function-based dynamic graphs have not much difference on the right and the wrong test cases, which is not conducive to mine the fail feature subgraph. In this paper, we introduce the basic block as the unit of a software dynamic execution graph. The basic definitions are as follows.

Definition 1 (Software Dynamic Execution Graph). A software dynamic execution graph \( G_i \) related to a test case, is a quadruple \( G_i = (V, E) \). \( V = v_1, v_2, ..., v_n \) is the set of vertices, where \( v_i \) \((1 \leq i \leq n)\) is a basic block node. And \( E = e_1, e_2, ..., e_{nm} \) is the set of directed edges where \( e_k = (v_i, v_j) \) \((1 \leq i, j \leq n)\) is a directed edge between two nodes.

Definition 2 (Subgraph Isomorphism). For two software dynamic execution graphs \( G_1 = (V_1, E_1), G_2 = (V_2, E_2) \), if there exists an injective function \( f : \forall v_i, v_j \in V_1 \text{ and } \langle v_i, v_j \rangle \in E_1 \Rightarrow \langle f(v_i), f(v_j) \rangle \in E_2 \); (2) the labels of \( \langle v_i, v_j \rangle \) and \( \langle f(v_i), f(v_j) \rangle \) are identical. So \( G_1 \) and \( G_2 \) are called isomorphism.

Definition 3 (Max Isomorphic Subgraph). Given a software dynamic execution graph dataset \( D = G_1, G_2, ..., G_n \) and a graph \( G_0 \), \( T(G_x, G_y) \) is a function for finding the isomorphic subgraph between the \( G_x \) and \( G_y \) graphs, \( Node(T) \) is the node count of a isomorphic subgraph. \( MIS = \max\text{Node}(T(G_0, G_i))(G_i \in D, 1 \leq i \leq n) \) is as the max isomorphic subgraph of \( G_0 \) referred to the graph dataset \( D \).
4 Graph Mining-based Software Fault Localization Localizing

The program execution sequence is extracted to create the graph structure model firstly, and then we select the correct and incorrect execution graphs for comparison, from which the discriminative subgraph is mined. In the end we sort the suspicious nodes belonging to the discriminative subgraph.

4.1 Building graph structure model

In order to obtain the dynamic execution graph, it needs to record covered state of a basic block during the program execution process. We make the instrumentation procedures for SIR benchmark programs artificially and collect the execution sequences of basic block and test results. The obtained execution sequences are converted into dynamic execution graphs with node weights. The value of node weight is determined by the level priority traversal of the tree model, so the node weight and its calling sort in a graph present a positive correlation relationship. The sooner a basic block is called, the greater its weight is.

**Definition 4 (Depth of Dynamic Execution Graph).** The longest path size from a program execution entry point $S$ to the remaining nodes is as the depth $H$ of the dynamic execution graph.

**Definition 5 (Node Weight).** The distance of a node is $h$ from the program execution entry point, the node weight is $1-h/H$. On the software dynamic execution graph, sometimes a node is as a part of multi-paths. At this situation the node weight is $\max(1-h_i/H) (1 \leq i \leq n)$, $n$ is the total path number through the node.

**Definition 6 (Weighted Execution Graph).** The software dynamic execution graph with weight is expressed as the graph structure $G(V,E,L,W)$, where $L$ is the set of node labels, $W$ is a function that assigns node to weight.

4.2 The node weight calculating strategy

When the suspicious nodes set is mined, a proper sorting on them will be an effective clue to help developers locate faults quickly. A method to calculate the node weight is designed in a program execution graph. If a node weight is larger, the suspicion of the node containing an error is stronger. The whole calculating process is followed as Fig. 1.

A sample program code summary is as below, it is an example in use for describing the calculating process of node weight.

```c
1 #include <stdio.h>
2 int main(int argc, char *argv[])
3 {
4     scanf("year:%d",&year);
5     for(year=1000;year<2010;year++)
6         {
7             printf("B0");
8             if(leapyear(year))
9                 printf("B1");
10             if(year%5==0)
11                 printf("B2");
12             printf("%d is a leap year
```
The sample program is designed for the sake of obtaining the whole leap-years in 1000-2010 periods, and then they are classified according to whether a leap year is divisible by 5. In the traversal process, different years have different execution paths shown in Table 1 below.

Table 1: Execution path sequences for different input data

<table>
<thead>
<tr>
<th>No</th>
<th>Year</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>P1={B0,B5,B1,B2}</td>
</tr>
<tr>
<td>2</td>
<td>2005</td>
<td>P2={B0,B6,B10,B4}</td>
</tr>
<tr>
<td>3</td>
<td>2008</td>
<td>P3={B0,B6,B7,B8,B1,B3}</td>
</tr>
</tbody>
</table>

Basic block B1 has two paths through the node, its weight is identified by the comparison of the calculated results. Depth is 6 calculated from the execution path graph in Fig. 2, for the path P1, Weight_P1 (B1) = 1-3 / (6 +1) = 0.5714; then in the path P3, Weight_P3 (B1) = 1-5 / (6 + 1) = 0.2857, so the Weight (B1) = Max Weight_P1 (B1), Weight_P3 (B1) = 0.5714. Similarly, the B6 weight is maximum value among the P2 and P3 paths, due to the B6 depth is the same 2 on the two paths, then the weight is Weight (B6) = 1-2 / (6 +1) = 0.7143. The calculation values are as Table 2.
4.3 Selecting the fail and pass behavior graphs

Software dynamic execution graphs are constructed from execution sequences, these graphs are divided into two categories based on the pass and fail results. Given software dynamic execution graph collection \( G = G_1, G_2, \ldots, G_n \), and \( C = \text{fail, pass} \) as the result labels. \( G \) is divided into gfail and gpas collection, in order to mine the discriminative subgraph for the fail execution graph, at first we select the correct and incorrect graph objects to conduct comparative analysis. There are a lot of fail graphs in the gfail collection, \( f_g \) is the fail comparison graph model selected, it is concerned with the coverage of the program execution.

\[
G_{cov} = \{cov(G_1), cov(G_2), \ldots, cov(G_n)\} \\
f_g = G_i \ (cov(G_i) = \min\{G_{cov}\}, i \in [1, n])
\]

The fail execution graph with minimum coverage is mined in gfail collection, according to the principle, when the test result is fail, it must contain the suspicious code block that causes error in the execution path.

The correct graph is selected via traversing the gpas graph collection, in order to find the similar graph with the fail graph \( f_g \). But in the actual process, the different types of errors
can lead to large differences in the degree of similarity. Some reach 100% in similarity for the two graphs. The others have relatively low similarity overall. To this end, the $k$ coefficient of Algorithm 1 makes similarity constraints. And we select the pass graph model with high similarity by sorting. The mining process of pass graph model $pg$ has been described in Algorithm 1.

In order to introduce the algorithm more clearly, some notations are listed as follows:

1. $k$: constraint coefficient of similarity
2. $F$: a function for maximum common subgraph in two graphs $G_1$ and $G_2$
3. $G_{iso}$: record maximum common subgraph for two graphs $G_1$ and $G_2$
4. $G_{sim}<x, sim>$: record index number and the corresponding similarity
5. $g^*$: record certain graph structure of a graph collection
6. $sort$: sort $G_{sim}$ set by the size of similarity

Algorithm 1

Mining the pass graph model for comparative analysis ($g_{pass}$, $fg$, $k$, $F$)

Input: $g_{pass}$, $fg$, $k$

Output: $pg$

0 $G_{iso}=\emptyset$, $G_{sim}<x, sim>=\emptyset$
1 for $i$ from 1 to $n$
2 $g^*=g_{pass}(i)$
3 $G_{iso}=F(g^*, fg)$
4 $G_{sim}<x, sim>=G_{sim}<x, sim> \cup <i, |G_{iso}|/|fg|>$
5 end for
6 $sort(G_{sim}<x, sim>)$
7 for $j$ from 1 to $n$
8 if $G_{sim}(j).sim<k$
9 $pg=g_{pass}(G_{sim}(j).x)$
10 break
11 end if
12 end for
13 return $pg$

Line 0 of the algorithm is the initialization operations, including initializing the initial node count of the maximum common subgraph $G_{iso}$ and the record data set $G_{sim}(x, sim)$ related to graph index number and the corresponding similarity. Both of $G_{iso}$ and $G_{sim}(x, sim)$ are initialized to be empty set. The similarity calculation of two graphs is given in line 1 to line 5. When there is no node in $g_{pass}$ graph set, the calculation process stops. In line 6, the result of calculation is sorted descending by similarity. The process of mining $pg$ is showed in line 7 to line 13. The index value of $pg$ is found by comparing similarity with the threshold $k$. Then, the $pg$ is got through $g_{pass}$ set.

The mining progress of $pg$ is relatively time-consuming, but with the traditional techniques based on statistical terms it is acceptable, because this method needs less times in graph traversal. The pass graph model is a higher similarity with fail graph model, and certainly there are differences which are needed to calculate feature subgraph between two ones.
The discriminative subgraph is relatively easy to be mined, the main task is to find difference between the pass and fail graph models. This difference is based on the basis of similarity. Giso is the maximum common subgraph as a result of comparative analysis for the two ones.

\[ Gother = \mathcal{C}_{fg} Giso \]  

Nother is a collection of nodes from Gother, which is part of the discriminative subgraph nodes collection, as well as the node collection Niso that belongs Giso also contains feature nodes. Because these nodes may be the starting point of differences for the correct and error graph models. Detail mining process has been described in Algorithm 2.

Some notations are listed as follows to introduce the algorithm more clearly:

1. F: a function for maximum common subgraph in two graphs G1 and G2
2. U: a function for node complement set of G2 in G1
3. Weight: sort the final set of suspicious nodes by node weight
4. Giso: record maximum common subgraph for two graphs G1 and G2
5. Nother: the feature node set outside the maximum common subgraph
6. Niso: the feature node set in Giso
7. Nsus: the set of total feature nodes
8. Node.parent: the parent node of a node

**Algorithm 2** Mining the collection of discriminative subgraph nodes \((fg, pg, F, U, \text{Weight})\)

**Input:** \(fg, pg\)  
**Output:** Nsus

```
0 Giso=\emptyset, Nother=\emptyset, Niso=\emptyset, Nsus=\emptyset
1 Giso=F(fg, pg)
2 Nother=U(fg, Giso)
3 for i from 1 to n
4 if Nother(i).parent \in Giso
5 Niso = Niso \cup Nother(i).parent
6 end if
7 end for
8 Nsus = Nother \cup Niso
9 Weight(Nsus)
10 return Nsus
```

Line 0 of the algorithm is the initialization operations, including initializing the Giso, Nother, Niso, Nsus. They record different nodes sets as the above notations listed and are initialized as empty set. In line 1, the maximum common subgraph nodes set Giso of two graphs is recorded. Thereby the feature node set Nother outside the maximum common subgraph is mined in line 2. The feature node set in Giso is given in line 3 to line 7. When the parent node of a node belonging to Nother set is contained in Giso set, the parent node is combined with Niso set. In line 8, the set of total feature nodes Nsus is got through merging the both Nother and Niso sets. In line 9, the nodes of Nsus set are sorted by the level of suspicion and the set Nsus is returned.
5 Preliminary Experiments

5.1 Establishing experiments

This section describes the process on the experiment that tests the effectiveness of our method. Experimental data set is the famous Siemens benchmark test suite in the field of software testing. These data sets are designed by the Siemens Corporation researchers to optimize the coverage of software testing. There are seven programs in the dataset, and each program has a number of erroneous versions which are seeded manually. It is appropriate for software fault localization, many localization technologies have used it as experimental subjects.

Tarantula is comparative with our software fault localization method, Tarantula is a well-done method based on statistical techniques. The statistical count of the execution statements for each test case is recorded. And then it calculates the suspicious degree of each statement according to the execution results. Finally, it sorts the suspicious statements, and finds the wrong location. Experimental results prove that it is an excellent method in fault location. The metrics are these ones: the traditional ratio of program that is examined and localization index value that is defined newly, they can evaluate the practicality of these methods.

5.2 Performance metrics

Experimental evaluation metric is the code scale detected for locating an error. It refers to the percentage of the examined code in the whole executable code from the start point to the end one that is the location of the real error code. In addition, in order to examine the effect of sorting suspicious nodes, we use localization index value to describe the real node containing an error. It refers to the ability in finding the count of fault program versions for suspicious nodes of different index values. If the most program errors can be located in the front of suspicious node index, and it proves the sorting strategy is effective. We make the additional statistics and calculations to get the evaluation data in various test programs.

5.3 Experimental results

In order to evaluate our proposed method, the famous software fault localization method Tarantula is added in experiment. It is a statistical-based method to inspect the count of covering about executable statements and locate the suspicious code area. The ratio of program that is examined, the experimental results are shown in Figs. 3 and 4. The ordinate is as the fault version amount, the abscissa as the ratio of checking code. In Siemens test program experiment, the fault version number is 132. When the ratio is 20%, the Sig BB method can find 95 errors out the 132 ones, while Tarantula only 83 errors. And we scan the trend throughout the figure, Sig BB method is better than Tarantula with the increasing ratio. When all errors are checked out, Sig BB needs to check the code size of 55%, while Tarantula needs to check 84% code size.
The real program Grep is selected in experiment; it is a total of 21 error versions. Fig. 4 demonstrates that Sig BB method can find the entire errors by checking the 7% code, while the Tarantula can discover 12 errors. Tarantula needs to check the code size of 24% when all the errors found. So it can be drawn that the Sig BB method is also better than Tarantula on locating real program errors.

For the sorting strategy of suspicious nodes, we record the number of errors that can be found in each suspicious node index, the experimental results shown in Figs. 5 and 6. The abscissa is the sorted index number of suspicious node; the vertical axis is the amount of program errors. Fig. 5 is about the Siemens charts, the size of errors found is greater than 16 in node index 2, 3, and 4, index 1 and index from 5 to 9 are average in 10 errors approximately, while the greater than 10 node indexes are not more than 3 ones. It can be seen that the index value containing the error code is small; most of the errors are found in the front of the sort. Likewise for Grep statistics, Fig. 6 show the errors found are 16 in the indexes that are smaller than 6, the other 5 fault versions are mined in the node index 8, 10 and 11. These indicate that most errors can be located as soon as possible and the examined code size used to fault localization becomes small.
6 Conclusions

In this paper, we propose a new graph mining approach for software fault localization, which can mine a suspicious node set about program errors from the graph model. Our software fault localizing is described mainly in the following aspects: (1) we build the graph model from software execution path sequence, and each node weight in graph is calculated expressing the suspected degree of the node; (2) we extract the pass and fail execution graphs for comparison analysis based on graph structure similarity, then mine the discriminative graph and find the feature nodes set; (3) we sort the feature nodes by the suspicious degree in order to help developers locate fault code area quickly. Through a series of experiments in Siemens and Grep programs, the results show that our fault localization method and the node sorting strategy are valid. With the increasing scale of software program, the data generated in the process of software testing will become very large, our method can locate the software fault at a small unit of basic blocks, and this helps developers reduce work time for large-scale software program at the most extent.

References


