Machine Learning Project Final Report:  
Bug Localization using Classification for Behavior Graph

Fuyao Zhao  
School of Computer Science  
Carnegie Mellon University  
fuyaoz@cs.cmu.edu

Abstract

Bug localization is a widely studied problem in program analysis. And in several studies, machine learning technique such as graph classification is introduced to aid the analysis of the software. These methods usually generate two sets of graphs of different runs. Knowing whether they have correct outputs, graph classifier can be built and its performance can be evaluated. If generate checkpoints for each program, partial graphs in each checkpoint could be built. By examining the classification performance boost in each checkpoint, a set of bug-like functions could be reported to help programmer identify and fix them. In this project, above idea is implemented. In addition, weighted subgraph model of Behavior Graph is built, and a technique to reduce the number of checkpoints and increase the result consistency is proposed, therefore both the classification performance and the localization performance will be improved.

1 Introduction

Debugging is a very painful process for each programmer. And along with the growing of software size nowadays, it become harder and harder to find what is bug in programs. Many approaches have been proposed to deal with computer aided debugging and testing, and some researches [6, 5, 2, 1] integrate machine technique to identify bug lines. Yuriy and Michael [2] uses dynamic invariant detection to generation properties for program, and build fault invariant classifier. Hong, David and Yang [1] mine the top-k discriminative graphs to find bug information. [2] is also a graph mining approach focus on backtracking the noncrashing bugs such as logic error, the framework of which can be basically divided into two parts:

1. Graph classification:
   (a) Generate graphs from different executions of program.  
   (b) Mine the graph set, find closed frequent subgraphs as features of program in addition to edges.  
   (c) Assign values to this features for each graphs  
   (d) Build SVM classifier and do cross validation

2. Bug localization:
   (a) For each function, we assign two checkpoints, and for each checkpoints, we can generate a set of partial graphs can be generated represent different stages of the program runs, in which each partial graph corresponds to the test case running up to the checkpoint.
(b) For each set of partial graphs, we build graph classifier as part 1, and evaluate its performance
(c) By looking the classification performance boost after each function is executed, we report a set of possible bug functions.

In the rest parts of the report, Section 2 discuss the method used in each step in detail, including the weighted graph model for CFG and checkpoints reduction. Section 3 gives a series of experiment, analyzing which method is better and how to choose parameters. We conclude our study in Section 4.

2 Method

2.1 Graph Generation

[2] uses behavior graph to describe the runs. A behavior graph is showed in Figure 1.a, which contains two parts: call flow graph (CFG) in Figure 1.b and transition graph in Figure 1.c. Functions are represented as nodes, solid arrows are calls and dash arrows are transitions.

![Figure 1: Behavior Graph, Call Flow Graph and Transition Graph.](image)

However, we should notice that transition edges are not independent of call edges. For example, if in Figure 1.b, the call edge (1, 3) is not exist, then the transition edge (4, 3) in Figure 1.c would also not exist. So call graph might be sometimes more useful than behavior graph since it contains majority information of the program and will generate less features, which means, we may need less training examples. Though we use behavior graph in the project, an empirical comparison between this two kinds of graphs is made in Section 3, which will show the behavior graph has better classification result, but we get less features by using CFG.

2.2 Subgraph Extraction

Let $G' \subseteq G$ denote that $G'$ is a subgraph of $G$, $D$ be the dataset of graph, and $\text{support}(g)$ denote the frequent of $g$ appear in $D$. A subgraph $g$ is frequent if $\text{support}(g) > \text{threshold}$. And a subgraph is closed if there is no supergraph $g'$ which $\text{support}(g') = \text{support}(g)$. We also restrict that and the subgraphs should be connected graphs. By naïve depth first search, we can get a set of closed frequent subgraphs. Figure 2 shows a dataset of behavior graph, and Figure 3 shows two subgraphs, where the first is closed frequent subgraph and the second is not.
2.3 Graph Classification

In [5], all the edges are used as features. If a graph has a particular edge, it will have a corresponding feature value 1, otherwise 0. And features of subgraph is defined in similar way.

In this project, a weighted graph model is used. For each edge \( e = (u, v) \) in graph \( g \), its weight:

\[
    w(e) = \frac{c_e}{\sum_{e' \in g} w(e')},
\]

where \( c_e \) is define as times that \( u \) call \( v \). The weight of a subgraphs \( g' \) of \( g \) is defined as:

\[
    w(g') = \sum_{e \in g'} w(e).
\]

This weight make sense because it can be thought as a measurement about how important is \( g' \) in \( g \). Section 3 will give a comparison of these two model.

To evaluate the performance of classification, solely use classification accuracy is not enough since we have few incorrect runs. We use recall and precision, and combine them into F-Score, which is defined as:

\[
    \text{recall} = \frac{\# \text{ successfully classified incorrect runs}}{\# \text{ incorrect runs}}
\]
\[
\text{precision} = \frac{\# \text{successfully classified incorrect runs}}{\# \text{all runs classified as incorrect}}
\]

\[
\text{F-Score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]

2.4 Checkpoint Generation

In [5], each function \( F_i \) has two checkpoints: \( B_i^\text{in} \) and \( B_i^\text{out} \), correspond to the entrance and the exit of \( F_i \). And let \( P_i^\text{in} \) be the classification performance in \( B_i^\text{in} \), and \( P_i^\text{out} \) is the performance in \( B_i^\text{out} \). A set of graphs can be generated at each checkpoint, which represent the test cases running up to the checkpoints respectively.

Therefore, given a graph of \( n \) nodes, we should generate \( 2n \) graph sets and do graph classification on each set. In this project, a more efficient method is proposed. Let \( F_j \) be the first function called by \( F_i \), and \( F_k \) be the last function called by \( F_i \), we define

\[
P_i^\text{in} = P_j^\text{in}, \quad P_i^\text{out} = P_k^\text{out}
\]

We define the node of function which will not call others the \textit{ender}, notice the ender may have multiple \textit{caller}. So, we actually only need generate a few checkpoints, which can be uniquely identified as pairs of (caller, ender). Then, by walking on the behavior graph of an incorrect wrong, we can assign classification score for each checkpoints. The advantage of doing this is not just for reducing the number of checkpoints, in addition, we can also reduce the result inconsistent as much as possible, e.g we will never have \( P_i^\text{in} < P_i^\text{out} \), where \( F_i \) call \( F_j \), but similar things could happen in original method due to the inaccuracy of the classifier. And in original method, \( P_i^\text{in} - P_i^\text{out} \) will always be 0 where \( F_i \) is an ender which might also be buggy. Figure 4 shows how to generate checkpoints from the program using behavior graph representation.

![Figure 4](image_url)

Figure 4: Checkpoints generated from Figure 1. The 4 checkpoints could be generated by depth-first traversal of behavior graph: when we walk down to ender, we generate subgraph contain all visited nodes.

2.5 Bug like function dectection

Now, we define a functions \( F_i \) is bug like if \( P_i^\text{out} - P_i^\text{in} > \theta \). It can be interpreted as: if the classification performance has been boost a lot (exceed \( \theta \)) after execution the function, then it is bug like. There should be a set of functions which is bug like, and if we choose a incorrect run, then they be lined up to form a backtracing according to the graph.

It should be noticed that this is the most difficult part of the project. Because this part is highly depend on the graph classification part. If the performance of the classifier is not as high as we wanted, then the result of bug localization may be nonsense. In addition, the number of test example for each checkpoint will be different since there may be some nodes exist in some graphs but not exist in other graphs, therefore, it is possible that for some checkpoints which should have high classification performance, they actually don’t have such performance due to the lack of training data. And it is possible that latter checkpoints have lower performance than that of the previous
checkpoints. However, we have already reduce such inconsistent as much as possible by reducing the number of checkpoints.

3 Experiment

We use replace program in Simens Programs as test data. It is a regular expression matching and replacing program, and is widely used as program analysis. It has a correct source file, and 32 versions contain different bugs. It also has 5542 input file. So we first run the standard program and get a set of correct output. Then run other version, and compare its outputs with standard ones. The detail of the data set is show in Table 1

<table>
<thead>
<tr>
<th></th>
<th>correct runs</th>
<th>incorrect runs</th>
<th>correct graph</th>
<th>incorrect graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>version 1</td>
<td>5478</td>
<td>64</td>
<td>300</td>
<td>24</td>
</tr>
<tr>
<td>version 2</td>
<td>5507</td>
<td>35</td>
<td>238</td>
<td>10</td>
</tr>
<tr>
<td>version 3</td>
<td>5414</td>
<td>128</td>
<td>300</td>
<td>12</td>
</tr>
<tr>
<td>version 4</td>
<td>5401</td>
<td>141</td>
<td>300</td>
<td>17</td>
</tr>
<tr>
<td>version 5</td>
<td>5280</td>
<td>262</td>
<td>302</td>
<td>39</td>
</tr>
<tr>
<td>version 6</td>
<td>5459</td>
<td>83</td>
<td>299</td>
<td>5</td>
</tr>
</tbody>
</table>

We use gprof [3] to generate the execution information of each run. For example, we use following command to compiler a program

gcc replace .c -o replace -pg

Run the program

`./ replace -?' 'a&' < input1`

Generate run information

gprof -b replace gmon.out

In the classification stage, we use SVM light [4] with linear kernel. 5-fold cross-validation is used to evaluate the performance of classifiers.

3.1 Threshold Choose

The threshold for subgraph extraction is important parameters, low threshold means more features we use, but we may also need more examples, high threshold means less features, but the classifier may also not perform well if too little features provided. Table 2 shows the experiment result for different threshold on version 4.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>F-Score</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.17</td>
<td>0.17</td>
<td>104.97s</td>
</tr>
<tr>
<td>0.33</td>
<td>0.18</td>
<td>19.73s</td>
</tr>
<tr>
<td>0.50</td>
<td>0.20</td>
<td>5.88s</td>
</tr>
<tr>
<td>0.67</td>
<td>0.44</td>
<td>1.84s</td>
</tr>
<tr>
<td>0.83</td>
<td>0.42</td>
<td>1.31s</td>
</tr>
<tr>
<td>0.95</td>
<td>0.42</td>
<td>1.13s</td>
</tr>
</tbody>
</table>

3.2 Call Flow Graph v.s. Behavior Graph

Table 3 shows the comparison between Behavior Graph and CFG (Threshold = 0.65). We can see that Behavior graph have slight advantage over CFG, especially for the case that incorrect runs is
scarce. However, we should notice that CFG may also useful in some classify application since it need less features.

Table 3: Behavior Graph v.s. CFG. The result of version 2 is pessimistic, since all the incorrect graphs are also in the set of correct graphs.

<table>
<thead>
<tr>
<th>Behavior Graph</th>
<th>Call Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-Score Features</td>
<td>F-Score Features</td>
</tr>
<tr>
<td>version 1</td>
<td>0.44</td>
</tr>
<tr>
<td>version 2</td>
<td>0.00</td>
</tr>
<tr>
<td>version 3</td>
<td>0.37</td>
</tr>
<tr>
<td>version 4</td>
<td>0.49</td>
</tr>
<tr>
<td>version 5</td>
<td>0.31</td>
</tr>
<tr>
<td>version 6</td>
<td>0.17</td>
</tr>
</tbody>
</table>

3.3 Weighted Model vs Unweighted Model

From Table 5 we can see that weighted model have higher F-Score compare to unweighted model (graphs are all weighted). Further more, weighted model converges much faster in SVM.

Table 4: Weighted Model v.s. Unweighted Model.

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Unweighted</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-Score Time</td>
<td>F-Score Time</td>
</tr>
<tr>
<td>version 1</td>
<td>0.44</td>
</tr>
<tr>
<td>version 2</td>
<td>0.00</td>
</tr>
<tr>
<td>version 3</td>
<td>0.37</td>
</tr>
<tr>
<td>version 4</td>
<td>0.49</td>
</tr>
<tr>
<td>version 5</td>
<td>0.31</td>
</tr>
<tr>
<td>version 6</td>
<td>0.17</td>
</tr>
</tbody>
</table>

3.4 Checkpoint Reduction

Checkpoint reduction trick can reduce the number of checkpoints we should examine. Since the the classification might be a very time consuming process, so less checkpoints will save times to finish the whole process.

Table 5: Total checkpoints with checkpoint reduction v.s. without checkpoint reduction

<table>
<thead>
<tr>
<th>With CR</th>
<th>Without CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>version 1</td>
<td>16</td>
</tr>
<tr>
<td>version 2</td>
<td>15</td>
</tr>
<tr>
<td>version 3</td>
<td>16</td>
</tr>
<tr>
<td>version 4</td>
<td>16</td>
</tr>
<tr>
<td>version 5</td>
<td>16</td>
</tr>
<tr>
<td>version 6</td>
<td>16</td>
</tr>
</tbody>
</table>

3.5 Bug localization

Figure 5 shows a incorrect run of version 4, in which bug appear in line 494 of the program:

```plaintext
if ((m >= 0) && (/= lastm BUG! += i != m))
```

And all non-enders have been marked their \([P_{in}, P_{out}]\). If we choose \(\theta = 0.10\), then the bug trace procedure is as follow:
• Find all the function which $P_{out} - P_{in} > 0.10$

• Backtrace these function in the graph

Then we can find two sequence of function to be suspicious

• main → getpat → matpat

• main → change → subline → amatch → patsize → in_pat_set

Thus, the scope for finding the bug is successfully narrowed down so that it is more easier for programmer to find the bug of program.

However, the result of our experiment is not perfect because of the limitation of our graph classifier, therefore you can see for some function $P_{out} < P_{in}$. The example here is a very good case in all the experiments, in some other cases, the performance boost for different checkpoints may not make sense because even the the final F-Score is so low.

Figure 5: Trace bug at an incorrect run.
4 Conclusion

In the project, we first generate behavior graphs from program runs, and compare the output to the standard output to give the graphs label. Then we try to classify the graphs using SVM with linear kernel under different models. We’ve shown that weighted model is superior than unweighted model because it reflect how important the subgraph is for its supergraph. And it is shown that the behavior graph performs better than CFG in most cases, but it do not have decided advantage, so CFG may also useful since we will need much less training examples.

The most important part of this project is to take advantage of the graph classification performance boost to locate bugs. We’ve shown how to reduce the number of checkpoints, and how to successfully backtrack the bug like functions. However, due to the low performance of our graph classifier, the bug location algorithm do not perform well on all program runs.

To improve the classification performance, one work could be done is to use more sophisticated graph mining technique [7]. Since number of negative examples is so low in our dataset, then it is possible that other measurement of classification is better than us, such as picking precision under highest recall [5]. In addition, using other kernel for SVM may help us get better result.

References


[5] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. Mining behavior graphs for backtrace of noncrashing bugs. In ICSE ’10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2005.


[7] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S. Yu. Mining significant graph patterns by leap search. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD ’08, pages 433–444, New York, NY, USA, 2008. ACM.