Improved Software Fault Detection with Graph Mining

Frank Eichinger EICHINGER@IPD.UKA.DE
Klemens Böhm BOEHM@IPD.UKA.DE
Matthias Huber HUBERM@IPD.UKA.DE
Institute for Program Structures and Data Organisation (IPD), Universität Karlsruhe (TH), Germany

Keywords: graph mining, software fault detection, weighted graph mining, call graphs

Abstract

This work addresses the problem of discovering bugs in software development. We investigate the utilisation of call graphs of program executions and graph mining algorithms to approach this problem. We propose a novel reduction technique for call graphs which introduces edge weights. Then, we present an analysis technique for such weighted call graphs based on graph mining and on traditional feature selection. Our new approach finds bugs which could not be detected so far. With regard to bugs which can already be localised, our technique also doubles the precision of finding them.

1. Motivation

Software quality is a big concern in industry. Almost any software displays at least some minor bugs after being released, incurring significant costs. A group of bugs which is particularly hard to handle is *noncrashing bugs*, e.g., logical failures which do not lead to a crash but to faulty results. Such bugs are in general hard to find, as no stack trace of the failure is available.

2. Related Work

Research in the field of software reliability has been extensive, and various techniques have been developed for locating bugs. One direction of research is static analysis, where properties of the source code or the version history are analysed. Another direction is dynamic analysis, which requires the execution of the program. These techniques can be further distinguished in data centric (analysing values of variables) and control flow oriented (analysing branches taken).

Many static techniques require a large bug and version history database, which is not always available. Nagappan et al. (2006), e.g., use standard metrics from software engineering and mine the source code with regression models.

Dynamic techniques usually work with just the current version of the software and require test oracles, which decide if an execution is *correct* or *failing*. An example of a dynamic data flow focused approach is the work of Liblit et al. (2003). It instruments the source code to gain program invariants. These are used as features which are analysed with regression techniques. This allows to discover potentially buggy pieces of code. Such instrumentation based approaches usually suffer from a poor runtime behaviour or from missed bugs, if the software is not fully instrumented.

Dynamic control flow based techniques often rely on *call graphs* of program executions. In such graphs, a node represents a method and an edge a method call. Figure 1(a) gives an abstract example. Recent work operating on call graphs by Liu et al. (2005) and di Fatta et al. (2006) employ *graph mining* techniques for bug localisation. These studies discover structural patterns in call graphs, which are characteristic for failing executions. A major challenge of graph mining

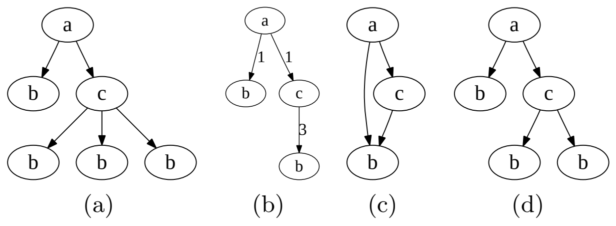

Figure 1. Variants of reduced call graphs.

Appearing in the *6th International Workshop on Mining and Learning with Graphs*, Helsinki, Finland, 2008.

techniques is that such algorithms do not scale for large graphs. Call graphs become relatively huge, as substructures typically are repeated many times. Therefore, reduction techniques need to be applied first. In the mentioned literature, this leads to a loss of information. Namely, call frequencies are lost, which are important for detecting certain groups of bugs.

3. Call Graph Reduction

The last section has mentioned two approaches of reducing software call graphs. The first one from Liu et al. (2005) does total reduction – every method occurs just once within the graph. See Figure 1(c) for a totally reduced version of Figure 1(a). This leads to small graphs, which allows for graph-mining-based bug localisation even with larger software projects. On the other side, much information about the program execution is lost, e.g., frequencies of the execution of methods and information on different structural patterns within the graphs. The approach from di Fatta et al. (2006) omits substructures which are called more than twice in a row (see Figure 1(d)). Thus, it keeps more information than the other one, with the risk of generating very large graphs. In consequence, graph mining algorithms might not work on these graphs.

In our approach, we try to overcome the shortcomings of both approaches and keep most of the information available. We reduce substructures executed several times in a row by deleting all but the first one and inserting the call frequencies as edge weights (see Figure 1(b)). This allows for a detailed analysis of the call frequencies. If, for example, a bug is hidden in a loop condition, this might lead to hundreds of iterations of the loop, compared to just a few in correct program executions. Note that, with both previous graph reduction techniques, the graph of the correct and failing execution is reduced to exactly the same structure in this case. In our approach, the edge weights would be significantly different. This raises the need for weighted graph mining algorithms. In the following section, we present a technique for analysing differences in edge weights subsequent to traditional graph mining.

4. The Mining Framework

We will now describe our framework to derive a ranking of potentially buggy methods from call graphs. This ranking can be given to a software developer who can do a code review of the suspected methods. At first, frequent subgraph mining is applied to the reduced call graphs. The resulting frequent subgraphs are then processed with two different approaches: the *conventional scoring approach* and our *entropy based approach*. As some bugs result in different call frequencies while others result in different substructures, we combine both scores, which leads to a final ranking. Figure 2 is an overview of this framework – the following paragraphs describe the individual steps.

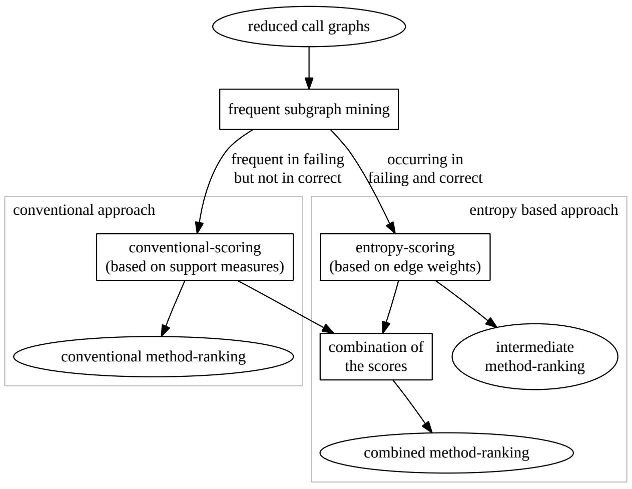
reduced call graphs
frequent subgraph mining
frequent in failing but not in correct
occurring in failing and correct
conventional approach
entropy based approach
conventional-scoring (based on support measures)
entropy-scoring (based on edge weights)
conventional method-ranking
combination of the scores
intermediate method-ranking
combined method-ranking

Figure 2. The ranking framework.

After having reduced the call graphs gained from correct and failing program executions, we do frequent subgraph mining with the *CloseGraph* algorithm (Yan & Han, 2003), ignoring the edge weights for now.

In the **conventional approach** (di Fatta et al., 2006), we just consider the discovered subgraphs which are frequent within the set of failing executions, but not frequent in the set of correct ones. In order to gain a scoring of the methods, we calculate for every method the probability P_c for containing a bug. Please see the literature for further details.

In our **entropy based approach**, we focus on frequent subgraphs occurring in both classes of program executions, the class of correct and the class of failing ones. Our goal is to find out which edge weights are most significant to discriminate the two classes of executions. To this end, we first assemble a feature table. This table contains all edge weights in all subgraphs discovered by CloseGraph in the columns[1] and all program executions (represented by the call graphs) in the rows. The following table serves as an example:

	SG_1 $a{\to}b$	SG_1 $a{\to}c$	SG_2 $a{\to}b$	…	*Class*
$Graph_1$	2	1	6	…	*failing*
$Graph_2$	0	0	4	…	*correct*
…	…	…	…	…	…

[1]More precisely, we also differentiate between edges occurring at different positions within one subgraph.

The first column corresponds to the first subgraph (SG_1) and the edge from a to b, the second column to the same subgraph but the edge from a to c, the third column to the second subgraph (SG_2) etc. In the last column, the class *correct* or *failing* is displayed.

Once the table is filled, we employ a standard feature selection algorithm to score the columns of the table and thus the effect of the different edges. We use an entropy based algorithm which calculates the information gain for every column. At this point, the ranking not only contains suspected methods, but suspected method calls (edges). Thus, not just the information in which method a bug is hidden is included, but also where exactly within this method. In order to compare and to combine our results with those of di Fatta et al. (2006), we just keep the starting methods from our edge list and omit the methods called. As our list might contain duplicates, we eliminate them and keep the maximum score for every method. This leads to our score P_e for methods and an intermediate ranking ordered by the likelihood of containing a bug.

Finally, we combine our edge weight based results gained so far with the ones from di Fatta et al. (2006). This results in an overall likelihood P of containing a bug for every method, based on the average of the normalised values for P_c and P_e.

5. Experimental Results

For the evaluation of our technique, we instrumented a Java diff tool (Darwin, 2004) with nine different bugs. The bugs we are using represent the same types of bugs which are used in the evaluation of Liu et al. (2005) and di Fatta et al. (2006). We executed every version corresponding to an instrumented bug exactly 100 times with different input data. We then classified the results as correct or failing executions with a test oracle based on a bug free reference version. Based on this data, we carried out three experiments (see Figure 2): **(1)** The *conventional method-ranking* following the approach from di Fatta et al. (2006), including its graph reduction technique, **(2)** the *intermediate method-ranking*, using our reduction technique together with the entropy based scoring and **(3)** the *combined method-ranking* which combines (1) and (2).

In order to evaluate the precision of the results, it needs to be checked in which line (out of 25 methods in our program) of the ranking the first instrumented bug is found. A software developer would have to pay attention to all lines till the bug is found. The following table displays the results of the three experiments for all nine versions ('25' refers to bugs not discovered):

Exp. \ Bug	1	2	3	4	5	6	7	8	9
1. Conventional	3	25	1	4	6	4	3	3	1
2. Intermediate	3	3	1	1	1	3	3	1	25
3. Combined	1	3	1	2	2	1	2	1	3

Bug 2 is not found by the structural technique (1) as it occurs within a loop and affects call frequencies only: it is found with edge weight analysis (2). Bug 9 in turn does not affect any call frequencies and can only be found by structural analysis (1). The combined approach (3) proves to be key to locate both bugs. Looking at the other bugs, our intermediate approach (2) already performs better than the conventional one (1). The combination (3) can slightly improve this. Leaving aside Bugs 2 and 9, our combined approach more than doubles the precision of di Fatta et al. (2006).

6. Conclusion

In this work we have presented a dynamic control flow centred approach for the localisation of noncrashing bugs. It uses call graphs which are reduced by a novel technique. This is done by introducing edge weights representing call frequencies. As none of the recently developed graph mining algorithms allows for the analysis of weighted graphs, we have developed a respective technique. This involves frequent subgraph mining and scoring of numerical edge weights using an entropy based algorithm. Our experiments show that a combination of structural and numerical mining techniques leads to significantly improved bug localisations.

References

Darwin, I. F. (2004). *Java Cookbook*. O'Reilly.

di Fatta, G., Leue, S., & Stegantova, E. (2006). Discriminative Pattern Mining in Software Fault Detection. *Proc. of the Int. Workshop on Software Quality Assurance (SOQUA)*.

Liblit, B., Aiken, A., Zheng, A. X., & Jordan, M. I. (2003). Bug Isolation via Remote Program Sampling. *ACM SIGPLAN Notices*, *38*, 141–154.

Liu, C., Yan, X., Yu, H., Han, J., & Yu, P. S. (2005). Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. *Proc. of the Int. Conf. on Data Mining (SDM)*.

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining Metrics to Predict Component Failures. *Proc. of the Int. Conf. on Software Engineering (ICSE)*.

Yan, X., & Han, J. (2003). CloseGraph: Mining Closed Frequent Graph Patterns. *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining (KDD)*.