Indian Journal of Science and Technology, Vol 8(35), DOI: 10.17485/ijst/2015/v8i35/86772, December 2015
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

An Enhanced Approach for Software Bug Localization using Map Reduce Technique based Apriori (MRTBA) Algorithm

A. Adhiselvam[1*], E. Kirubakaran[2] and R. Sukumar[3]

[1]Department of MCA, S.T.E.T. Women's College, Mannargudi, Tiruvarur - 614 016, Tamil Nadu, India; adhiselvam@yahoo.com
[2]STTP (System), Bharat Heavy Electrical Limited, Trichy - 620014, Tamil Nadu, India; ekiru@bheltry.co.in
[3]Department of CSE, Sethu Institute of Technology, Kariapatti, Virudhunagar - 626115, Tamil Nadu, India; rsukumar@sethu.ac.in

Abstract

Background/Objectives: Software bugs are generally the faults or errors that occur in the code that may leads to incorrect results. It is therefore necessary to find software bugs to increase the quality of software and for making the software to meet the user requirements. **Methods/Statistical Analysis:** The testing enables assessment of software which ensures the system whether it meets the system requirements. Graph mining is an approach to find software bugs and furthermore testing involves more computational complexity and cost therefore tools are developed for testing the software. The challenging task is to locate and fix bugs automatically. Bug localization using high performance map reducing process is very successful in the recent research. **Findings:** This proposed technique called Map Reduce Technique Based Apriori (MRTBA) Algorithm based on Graph mining in where edge weights are used to takes the program as the input and computes method calls of the program. Here each node represents method and each edge represents a method call. The algorithm aims for detection of faulty nodes. It uses Hash Map for reducing edge weights. JUnit test cases are performed at last for detecting bugs by giving different inputs. The approach mines all nodes not only at same level but also the nodes present at different levels. **Conclusion/Improvements:** In this work MRTBA, a dynamic control flow centered approach for bug localization has been presented. In future this technique can be extended with the proposal of steps involves in connecting same nodes at different levels. It also further enhances to find out indirect recursions for both the trivial and non-trivial map reduction in software bug localization.

Keywords: Apriori, Call Graph, Graph Mining, Map Reduce, Software Bug Localization, Subgraph

1. Introduction

Today's software industries are competing for software quality which depends upon sound software testing phase. No software is released bug free if tested extensively. Bug localization is the process of identify set of statements that causes the failure of a program. Now-a-days, Software reliability is becoming a top concern in modern industries. Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or art of determining the location of a bug in a program. It is hard to locate the bugs in complex programs. Bug localization can be performed by analysing of call graph. Call graph can be either static or dynamic. Static call graph describes every possible runs of a program. Static call graph can be generated from

*Author for correspondence

source code. Dynamic call graph describes the record of an execution of a program. It considers only one run of a program. Without any further treatment, a call graph is a rooted ordered tree.

Graph mining is a stream in data mining. Several algorithms had been proposed for frequent graph mining. Its difficulty lies on computation as the call graphs are not scale. Therefore necessary reduction techniques call graph is a debugging tool for finding out software bugs. It generally relates two functions. It can be used to find what calls a particular function or the functions called by a particular function for finding the buggy portions on which the system fails. The call graphs are large as the subgraphs may get repeated many times involving more computational overhead. So it has to be reduced. Several approaches had been proposed previously for call graph reduction. The system involves generation of reduced call graphs for the source code. Dynamic control flow based techniques rely on call graphs on program executions. They involve structural patterns in call graph which are characteristic for failing executions.

Large bug and version history database are needed for many of the static techniques and that are not always available. Standard metrics from software engineering are used by[1] for mining the source code with regression models. Dynamic techniques require test oracle for deciding whether an execution is correct or failing one and also it work for current version of the software. The work of[2] can be said as an example for focused approach on dynamic data. It gains program invariants from the source code. Various regression techniques are used for analysing these features. Such things are very useful to find buggy pieces of code. Such techniques suffer from missed bugs or from poor run time behaviour when the software is not fully instrumented.

In the existing approach, the substructures which comes more than one in arrow will get deleted resulting in reduced call graph. The corrected and failing execution is reduced to same structure. In the proposed approach, after reduction of produced call graph the structure will have different structure. Hash map data structure is used which stores node as key and edge weights as values. If the node calls same method again the value will get updated. The algorithm covers self recursion as well as indirect recursions also. The organization of the paper is as follows. Section 2 provides related work of my research problem. Design of MRTBA algorithm is provided in Section 3. Section 4 describes experiments and result of the algorithm. Final section contributes conclusion and future work.

2. Related Work

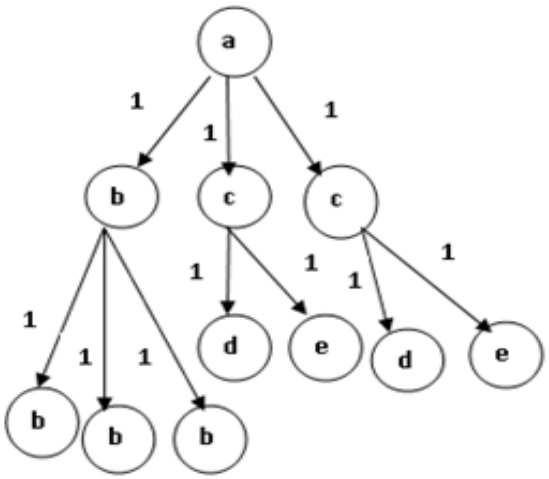
For researching problems in source code, a very useful tool called Call Graph is used. It allows an individual to track back to a piece of code by seeing a visual reference of the order calls which are performed. Such technique makes an individual to test the data output in various aspects faster without having to open more number of files and searching to track back manually. The referred work[3,4] involves are two reduction techniques that are to be considered: Total Reduction[5] and Zero-one-many Reduction[6]. Total reduction technique is proposed by[5]. Total reduction technique projects every node representing same method in the call graph to a single node. Total reduction shortens the size of the source call graph. This reduction technique limits the size of the call graph to number of methods in a program. The major disadvantage of this technique is that it changes the structure of the source call graph totally. Lot of information regarding frequencies of execution of methods and information of different structural patterns within the graph is lost. So it is very difficult to retrieve the required information from this reduced graph.

Figure 1(a) is the input source call graph created from source code. Figure 1(a) depicts 'a' is the root node and the nodes which are directly connected to are considered as child nodes given 1 as edge weight. Nodes are repeated if it is called again. Figure 1(b) is a reduced call graph for the call graph shown in Figure 1(a).

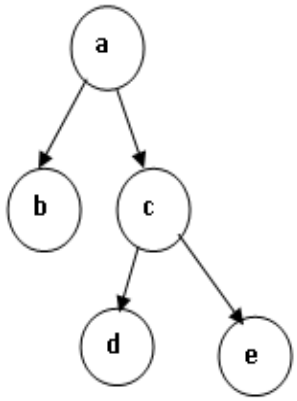
2.1 Call Graph based Fault Detection

In the first study of dynamic call graphs, graph mining techniques are applied are considered to be software behaviour graphs. These are called as reduced call graphs which infer some temporal information. By applying variant close graph algorithm, the behaviour graphs which represent correct and failing program execution are mined. The resultant frequent sub graphs are used as the binary features. The binary features are the one used for characterising program execution. Every execution is represented by a boolean. If a certain sub graph is included in the corresponding behaviour graph it will be indicated by the element in that vector.

A support vector machine is learned from the feature vectors which decide whether the program execution is



(a)



(b)

Figure 1. (a) Call graph. (b) Reduce call graph.

correct or a failing one. Briefly, two classifiers are learned for every method; the first one is based on behaviour graphs which include the respective method and another one is based on graphs. When adding graphs containing a particular method and if the precision rises significantly that particular method will likely to contain more bugs. Siemens programs are experimented and five out of 130 bugs demonstrates good classification performance but it is not evaluating bug precision. And moreover the authors have not generated any ranking for the methods suspected to contain bug. As we do so, our approach cannot be compared directly.

From approach[7], a reduction technique is again applied to the raw call trees first. The next step is same as the one which is described above. Graph mining algorithms are used for analysing collection of reduced call graphs representing correct and failing program executions. For finding all frequent sub trees the authors use tree miner FREQT. The analysed call trees are large to examine and may lead to scalability problems of the algorithm and hence the size of the sub trees searched are limited by the authors to a value up to 4. Then the resultant sub graphs which are frequent in the failing program executions are analysed by the authors and they have not considered the frequent ones occur in correct executions. This set of subgraphs is called as specific neighbourhood. Probability of method containing bug is calculated based on support values for all the method when invoked within subgraphs which is a part of the specific neighbourhood.

2.2 Call Graph Mining and Reduction

The technique involves tracing bugs using test oracle for every program. Then each subgraph is reduced resulting in smaller subgraphs. Now frequent sub graphs are mined. For mining frequent subgraphs, algorithms like graph mining or tree mining can be used in different invariants. The final step involves calculation of likelihood value of containing a bug. For every method invocation, this can be fine granular or more coarse grained. The likelihood value which is calculated facilitates ranking of methods which can then be given to software developer and the calculation is based on the frequent sub graphs mined.

In the referred existing works[8–10] say that on call graph mining, two different approaches exist which lead to reduced call graphs. The reduction technique used in projects every node representing the same method in the call graph to a single node in the reduced graph. We call this technique total reduction. Note that this may give way to the existence of loops and limits the size of the graph (in terms of nodes) to the total number of methods in the program analysed. As an example, the raw ordered call tree in Figure 2(a) would result in the reduced graph displayed in Figure 2(b). In addition to the reduction, so called temporal edges are inserted between all methods which are executed consecutively and are invoked by the same method. This technique integrates the temporal order from the raw ordered call trees into the graph representations.

Technically, temporal edges are directed edges having another label, e.g., "temporal", compared to other edges which are labelled, say, "call". Figure 2(c) serves as an example of a graph using the reduction technique and temporal edges, called software behaviour graph. This reduction is rather severe, e.g., from several millions of nodes to several hundreds. Even for larger software

projects this allows graph based bug localization. In contrast, much information about the program execution is lost. This concerns frequencies of the execution of methods as well as information on different structural patterns within the graphs. In particular, the information in which context a certain substructure is executed. Furthermore, the temporal edges increase the size of the graphs significantly. An increased precision of fault detection by using temporal edges has not been evaluated. The approach keeps more information.

It omits substructures of subsequent executions, which are invoked more than twice in a row from the same node. See Figure 2(d) for an example. This reduction ensures that many equal substructures called within a loop do not lead to call graphs of an extreme size. In contrast, the information that some substructure is executed several times is still encoded in the graph structure, but without exact numbers. Compared to[3], much more information about a program execution is kept, compromised by a call graph which is generally much larger. For example, graphs are reduced from several millions of nodes to several ten thousand nodes. Figure 2(a) is the generated call graph. Figure 2(a) can be reduced to form a subgraph shown in Figure 2(b). Figure 2(c) can be said as an example for a graph produced as a result of reduction technique and temporal edges. In Figure 2(d), this reduction ensures that many equal substructures called within a loop do not lead to call graphs of an extreme size. Figure 2(e) shows about the edge weights. They give the detailed analysis of a software bug. It is different from previous graph which has same subgraph for success and failing executions.

2.3 Various Approaches

In convention approach, the referred works[11,12] say that they discovered subgraphs which occur frequent are considered within set of failing executions, but not frequent in the set of corrected ones. In order to gain a scoring of the methods, the probability Pc of bug that a method can contain is calculated. In entropy based approach, the referred work[13] says they mine frequent sub graphs which occur more common in both classes of program executions i.e., the class of correct and the class of failing ones are focussed. The main goal is to find out edge weights which are most significant to discriminate correct and failing classes of executions. To this end, feature table is assembled. This table contains all edge weights in all sub graphs is covered by Close Graph in the columns and all program executions (represented by the call graphs) in the rows. The referred work[14] says about the apriori

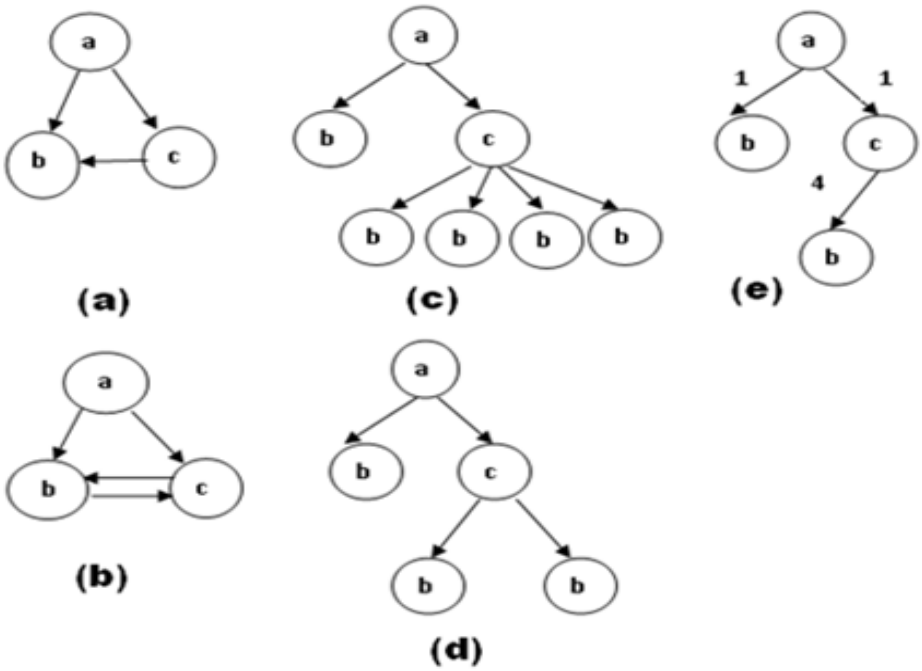


Figure 2. Process of call graph reduction.

algorithm. All subsets of a frequent item set must also be frequent.

- Because any transaction that contains X must also contain any subset of X.
- If we have already verified that X is infrequent, there is no need to count X supersets because they must be infrequent too.

2.4 Graph Mining and its Applications

The referred work[9] says about graph mining and various graph mining algorithms. A (sub) graph is frequent if its support (occurrence frequency) in a given dataset is no less than a minimum support threshold. Applications of graph pattern mining includes Mining biochemical structures, Program control flow analysis, Mining XML structures or Web communities, Building blocks for graph classification, clustering, compression, comparison, and correlation analysis. The Properties of Graph Mining Algorithms includes Search order, breadth vs. Depth, Generation of candidate sub graphs, apriori vs. pattern growth, Elimination of duplicate subgraphs, passive vs. Active Support calculation, embedding store or not, discover order of patterns, path tree graph.

Referred work[15] says about the essence of quality of software systems. Generally quality is accessed using on the basis of fault proneness of the systems. Building highly reliable software is becoming essential part in most of the existing system. Both fabric of modern society and the public safety mainly depend on software-intensive systems. A software fault prediction is a proven technique in achieving high software reliability. The defect prone modules are predicted which is one of the ways to improve software quality through improved scheduling and project control. The testing related issues and software quality is becoming increasingly important one for building a quality software. Although there is diversity in the definition of software quality, it is widely accepted that a project with many defects lacks quality. Methodologies and techniques for predicting the testing effort, monitoring process costs, and measuring results can help in increasing efficiency of software testing.

Referred work[16] defines about defect modelling. Defect modelling has received substantial attention from the empirical software engineering community. Researchers have studied the impact that product (e.g., number of lines of code and code complexity) and social (e.g., code ownership and developer experience) metrics have on defects in source code modules. Existing methods have used regression, data mining and machine learning techniques to model defects. The importance of size (i.e., lines of code) in understanding software quality has been acknowledged by many researchers. Size has consistently been found to be one of the most important metrics when modelling defects in source code modules .However, there has not been a consensus on the functional form (i.e., the exact mathematical form) of the size-defect relationship.

Referred work[17] explains about software defect prediction technique called feature selection. For maintaining the high quality of software systems and for reducing the costs of software development Software Defect Prediction (SDP) is used. This facilitates the project managers of the software industry to allocate limited time and the manpower resources to defect-prone modules by identifying through early prediction. In many machine learning and pattern recognition applications feature selection is widely used for decades. The feature selection technique is used to find the minimally sized feature subsets which are useful and are sufficient for specified task. The feature selection is mainly classified into three classes: 1. Embedded-type methods 2. Wrapper-type methods 3. Filter-type methods. In wrapper-type methods, for evaluating each candidate feature subset according to their predictive power uses a learning machine of interest as a black box testing and it is usually computationally expensive. In the process of training, Embedded-type methods perform feature selection and also they are specific to the given learning algorithm. The filter-type methods involve selecting features according to some criteria such as mutual information and it won't use any learning algorithm unlike wrapper-type and embedded-type methods and hence they are widely accepted and adopted in practice as it is simple and computationally efficient one. Referred work[18] depicts graph mining tools which are used for mining graph data. The tools include Cytoscape, Gephi, GraphInsight, and Knime.

3. Design of MRTBA Algorithm

In conventional approach, sub graphs are discovered. The frequent sub graphs under failing conditions are considered. In entropy based approach, the call graphs for both set of correct classes and incorrect ones are studied. The edge weights are used to distinguish correct and failed ones. The proposed technique MRTBA involves generation of call graphs and reduction of call graphs. It not only

takes same node at same level but also considers the same node at multiple levels. The call graphs generated are large which makes the mining more complex. Therefore necessary reduction techniques have to be used for reducing call graph. The main method will be considered as the root node and the methods directly linked to it are considered as child nodes and the initial edge weight is assigned as 1. By using apriori algorithm states that if a node is fault, all the subgraphs following it will also be fault one.

Algorithm:

Step1: Upload input program for which bug has to be detected

Step2: Compute all the methods defined in the input program
Call getMethodNames():
{
Return methodnames as array;
}

Step3: Check for method call in the input program
checkmethodbodyfunction(String s)
{
Construction of regular expression for checking method call
Equates s to regex
If matches
Return name
Else return ""
}

Step4: Read input program
For String s:line l
String s= checkmethodbody()
If(s != "")
{
Create a list
For each method in methodary
{
Call extractmethodbody()
Append function that computes callers and callees of this particular method
If called store name of method called and the callers name with '-' separated in list
}
Create log file
Iterate the list
Write contents to log file
}

Step5: Execute the newly created file and after execution log file will be created and all the callers and callees of each method had been written

Step6: Read the log file
For String s: lines of log file
String ar[]=s.split("-")
Input to graphviz tool as ar[0]← source node and ar[1]← destination node with edge weight as 1

Step7: Reducing call Graph:
Read the log file
For String s: lines of log file
String ar[]=s.split("-")
If(map.size()==0){
Store ar[0] as key and ar[1] as value
}else
{
Iterate map
If(map.getkey()==ar[0])
update count i.e.,edge weight+1
Store ar[0] as key and updatedcount as value
Else
Store ar[0] as key and ar[1] as value
}

Step8: Run the JUnittestcase and compute the bug

Step9: If a bug persists in some methods,the following subgraphs are also considered as bugs as per apriori algorithm.

The entire procedure of this algorithm is depicted in Figure 3 for better understanding the operations.

4. Experiments and Discussion

The conventional approaches have the drawbacks that every method occurs once within the graph. This leads to small graphs, which allows for graph-mining-based bug localization 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 Entropy approach omits substructures which are called more than twice in a row. 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.

Table 1 refers to call graph generation. For example a program that gets number from user and perform

arithmetic operations such as addition and multiplication. The program execution begins with main method. It calls put method which gets input two times and after that calls add() and mul() methods twice for performing tasks. Table 2 briefs about the reduced call graph. Here the same method calls are updated by its edge weight incremented by one for each call with initial value set to 1.

Each testable block said to be units that are tested individually for assuring that the software works perfectly in real world. Here for finding bugs test cases are written for that particular program and are tested with JUnit4.jar. The run time errors are captured and are shown.

The below graph (Figure 4.) shows that from main method put method is called twice indicated with two

Table 1. Call graph generation data

	Main-put	Main->put	Put->add	Put->add	Put->mul	Put->mul
Edge weight for the respective methods	1	1	1	1	1	1

Table 2. Reduced call graph data

	Main-put	Put->add	Put->mul
Updated Edge weights	2	2	2

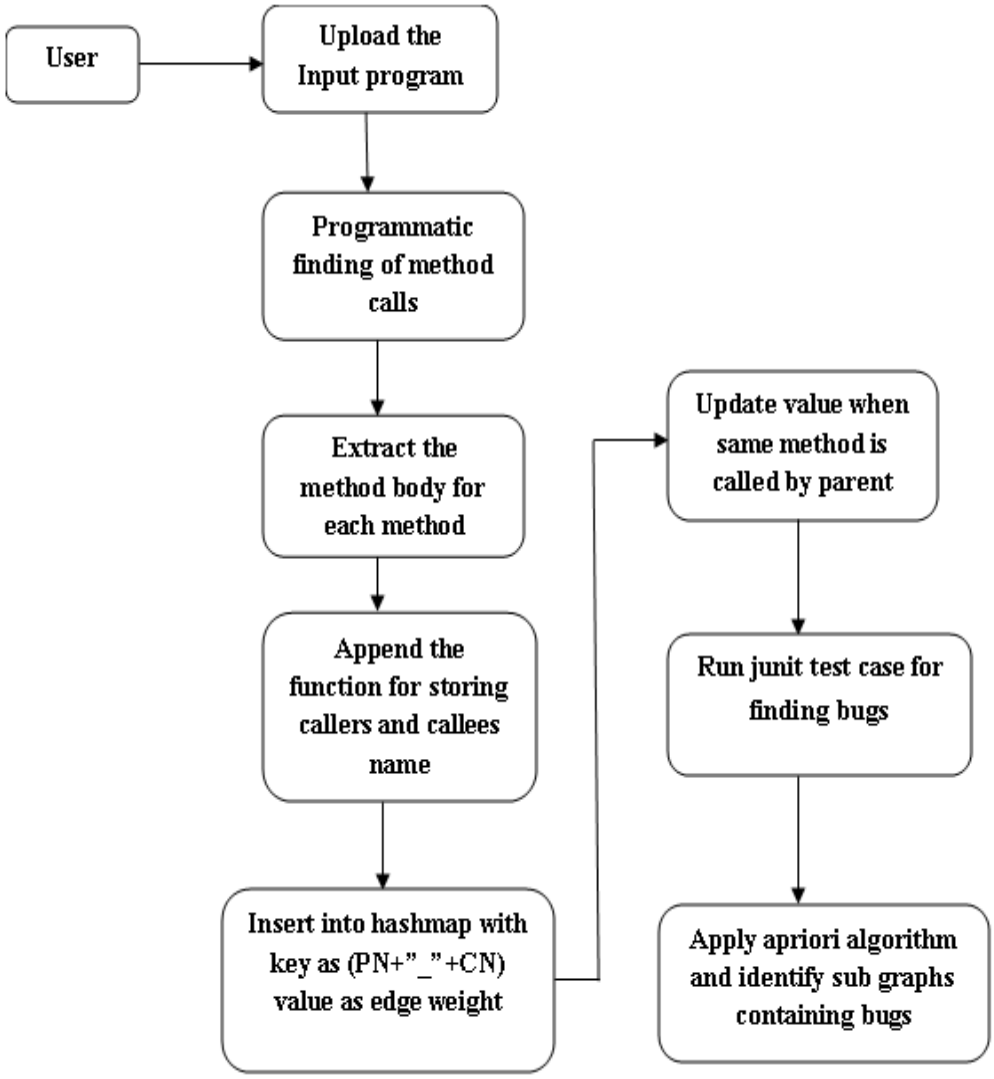


Figure 3. Architecture diagram.

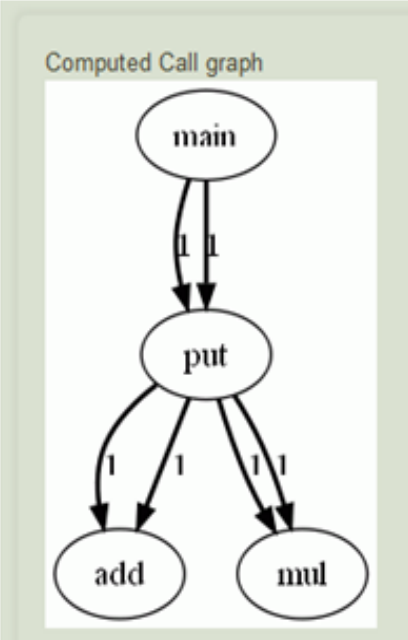


Figure 4. Computed call graph.

arrows with edge weights 1 and from put method add and mul methods are called twice therefore two arrows for each method are drawn with edge weights with 1.

The graph (Figure 5.) is shown with updated edge weights. The previously generated output is processed and edge weights for same method calls are updated. The final graph (Figure 6.) is computed by executing Junit test cases for that particular program. Buggy methods are identified and the sub graphs which follow it also considered as error-prone ones and are indicated with red colour dotted lines.

5. Conclusion and Future Work

In this work MRTBA, a dynamic control flow centred approach for bug localization has been presented. It involves generation and reduction of call graphs. The call graphs are reduced by the proposed technique. The reduction is done with the introduction of edge weights which represents call frequencies. As none of the recently developed graph mining algorithms allows for the analysis of weighted graphs, this technique developed a respective technique. Our experiments show that the previously frequent pattern mining was used for bugs localization. The proposed technique can be used for generating call graphs for improvising software behaviour with granularity. In future this technique can be extended with the

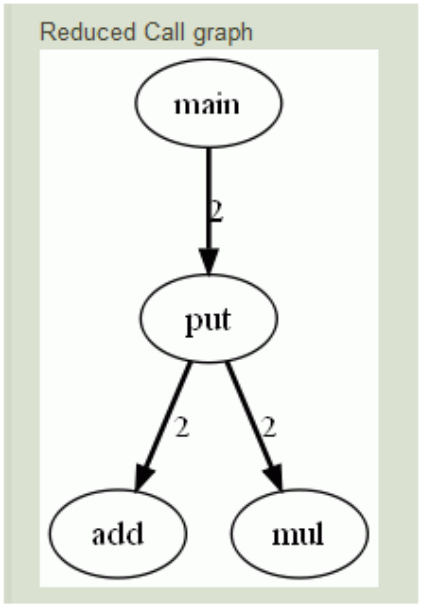


Figure 5. Reduced call graph.

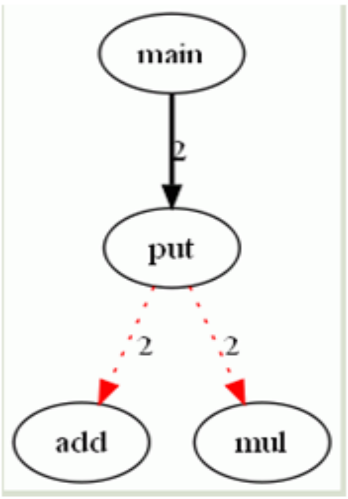


Figure 6. Final graph.

proposal of steps involves in connecting same nodes at different levels, self-recursions process to get the maximum number of minimum map reduced nodes. It also further enhances to find out indirect recursions for both the trivial and non-trivial map reduction in software bug localization.

6. References

1. Nagappan, et al. Advanced computer and communication engineering. ACM Springer publications; 2014 Nov. p. 100–64.

2. Ben L, et al. Bug isolation via remote program sampling. ACM Sigplan Notices. 2003; 38(5):141–54.
3. Frank E, B-ohm K, Huber M. Improved software fault detection with graph mining. Workshop on Mining and Learning with Graphs; 2008.
4. Frank E, Bohm K, Huber M. Mining edge-weighted call graphs to localise software bugs. Machine Learning and Knowledge Discovery in Databases. Springer Berlin Heidelberg. 2008; 333–48.
5. Chao L, et al. Mining behavior graphs for "backtrace" of noncrashing bugs. SDM. 2005.
6. Di Fatta G, Leue S, Stegantova E. Discriminative pattern mining in software fault detection. Proceedings of the 3rd International Workshop on Software Quality Assurance, ACM; 2006.
7. Borgwardt K, Stegle O. An introduction to graph mining. Available from: http://agbs.kyb.tuebingen.mpg.de/wikis/bg/RGV-GraphMining-slides.pdf
8. Elseidy M, Abdelhamid E, Skiadopoulos S, Kalnis P. Frequent subgraph and pattern mining in a single large graph. International Conference on Very Large Databases; 2014 Mar 1-5.
9. Gudes E. Graph and web mining-motivation, applications and algorithms. International Journal on Software Bug Management. 2010 May.
10. Böhm K, Reussner RH. Data-mining techniques for call graph based mining. Karlsruher Institute for Technology; 2011 May. p. 4-9
11. Kang U, Faloutsos C. Big graph mining: Algorithms and discoveries. ACM SIGKDD Explorations Newsletter. 2013; 14(2):29–36.
12. Singh P, Batra S. A novel technique for call graph reduction for bug localization. International Journal for Computer Applications. 2012 Jun; 47(15).
13. Han J, Yan X. Mining, indexing and similarity search in graphs and complex structures. University of illionois at Urbana-champaign; 2006. p. 4–17.
14. Toza L, Thomas D, Myers B. Visualizing call graphs. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC); 2011.
15. Bisht A, et al. A survey on quality prediction of software systems. IEEE Conference on Software Quality Assurances, IEEE Computer Society; 2012. p. 23-36.
16. Mark DS, et al. Replicating and re-evaluating the theory of relative defect-proneness. IEEE Transactions on Software Engineering. 2015; 41(2):176–97.
17. Mingxia L, Miao L, Zhang D. Two-stage cost-sensitive learning for software defect prediction. IEEE Transactions on Reliability. 2014; 63(2):676–86.
18. Vedanayaki M. Graph mining techniques, tools and issues-A study. Indian Journal of Science and Technology. 2014 Nov; 7(S7):188–90.