2018 42nd IEEE International Conference on Computer Software & Applications

Test Case Prioritization Based on Method Call Sequences

*Jianlei Chi, *Yu Qu, *Qinghua Zheng, †Zijiang Yang, *Wuxia Jin, *Di Cui, *Ting Liu
*Ministry of Education Key Lab for Intelligent Networks and Network Security
Xi'an Jiaotong University, Xi'an, China
Email: chijianlei@stu.xjtu.edu.cn, {quyuxjtu, qhzheng}@xjtu.edu.cn, {wx_jin, cuidi, tliu}@sei.xjtu.edu.cn
†Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA
Email: zijiang.yang@wmich.edu

***Abstract*—Test case prioritization is widely used in testing with the purpose of detecting faults as early as possible. Most existing techniques exploit coverage to prioritize test cases based on the hypothesis that a test case with higher coverage is more likely to catch bugs. Statement coverage and function coverage are the two widely used coverage granularity. The former typically achieves better test case prioritization in terms of fault detection capability, while the latter is more efficient because it incurs less overhead.**

In this paper we argue that static information such as statement and function coverage may not be the best criteria for guiding dynamic executions. Executions that cover the same set of statements /functions can may exhibit very different behavior. Therefore, the abstraction that reduces program behavior to statement/function coverage can be too simplistic to predicate fault detection capability. We propose a new approach that exploits function call sequences to prioritize test cases. This is based on the observation that the function call sequences rather than the set of executed functions is a better indicator of program behavior. Test cases that reveal unique function call sequences may have better chance to encounter faults. We choose function instead of statement sequences due to the consideration of efficiency. We have developed and implemented a new prioritization strategy AGC (Additional Greedy method Call sequence), that exploit function call sequences. We compare AGC against existing test case prioritization techniques on eight real-world open source Java projects. Our experiments show that our approach outperforms existing techniques on large programs (but not on small programs) in terms of bug detection capability. The performance shows a growth trend when the size of program increases.

***Index Terms*—software testing, test case prioritization, call behavior graph**

I. Introduction

During development and maintenance, software continuously evolves due to numerous reasons such as modifying old features and adding new features. In order to avoid the introduction of new bugs, it is necessary to apply regression testing that aims at detecting regressions and validating software changes by using the existing test case suite [1]. However, regression testing can be very expensive. As reported in some previous studies, a regression testing may last for more than seven weeks [2], [3].

In order to alleviate the cost of regression testing, Test Case Prioritization (TCP) seeks to find the ideal ordering of the test cases, so that a regression testing obtains maximum benefit under limited resources or when the testing is prematurely halted at some arbitrary point.

Since the abstract definition does not specify f, there exist various concrete ways to achieve test case prioritization. A large number of approaches [4], [5], [6], [7], [8], [9], [10] have been proposed that mainly focus on two aspects of test case prioritization. The first is the criterion that measure the effectiveness of a test case and the second is the strategy that exploits the criterion to prioritize the test cases. Most existing algorithms use structural coverage [4], [11], [12], [13], [14] as the criteria, based on the hypothesis that a test case with a higher coverage rate has a better chance to detect faults. A coverage criterion at finer granularity, such as statement coverage, typically detects faults sooner at a cost of more overhead; while a coverage criterion at coarser granularity, such as function coverage, gives faster but less accurate prioritization. Both function coverage and statement coverage are widely used in TCP today [14] and various studies have examined their trade-off [15], [16], [17]. Based on the criterion, different prioritization strategies can be adopted. For example, in greedy strategy, [18], [2] test cases can be ranked by their absolute coverage, i.e. select the one that has the highest coverage among the remaining test cases, or by the relative coverage, i.e. select the one that has the highest *new* coverage not covered by already selected test cases.

We argue that structural coverage may not be the best criteria to guide the prioritization of dynamic executions. What the existing techniques try to achieve is to reduce program behavior to statement or function coverage. However, such abstraction leads to severe information loss and thus may lead to inaccurate test case prioritization.

In this paper, we propose a new test case prioritization technique that is based on call sequences. Compared with structural coverage, we believe that call sequence is a better indicator of the dynamic behavior of a program. Our hypothesis is that the same call sequence may exhibit similar program behavior and thus the test cases with the richest call sequences should be considered first. Of course there will still be information loss by reducing executions to call sequences. But we believe the reduction is more accurate than those obtained by structural coverage. While structural coverage considers the vertices, our approach considers paths

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00039

along the edges. Based on our new criterion, we propose a new prioritization strategy called Additional Greedy Call sequence (AGC) that will be explained in Section III.

We have implemented our approach and multiple state-of-the-art techniques, and compare their performance on eight real-world open source Java projects. The experimental results show granularity indeed makes a difference, as pointed out by prior research [7], [1]. Statement-coverage based techniques are the best when programs are small. However, our experiments confirm that fine granularity incurs the larger overhead. This is the main reason that we target function call sequence instead of statement sequence. Indeed, our approach is the most effective one in terms of APFD when the programs are large.

The rest of this paper is organized as follows. Section II summarizes the related work, followed by a detailed explanation of our approach in Section III. In Section IV, we present our empirical study. Section V concludes the paper.

II. Related Work

Definition 1: [19] Given a test suite T, the set PT of permutations of T, and a function f from PT to the real numbers. The problem of test case prioritization is to find $T' \in PT$ such that $\forall(T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.
Here, PT represents the set of all possible orderings of T and f is a function that, applied to any such ordering, yields an award value for that ordering. The goal of the prioritization is to increase the likelihood of revealing faults earlier in the testing process.

There are two aspects of TCP. The first is the criterion that measure the effectiveness of a test case. As stated earlier most existing algorithms use structural coverage [4], [11], [12], [13], [14] as the criteria, based on the hypothesis that a test case with a higher coverage rate has a better chance to detect faults. Both function coverage and statement coverage are widely used in TCP today [14] and various studies have examined their trade-offs [15], [16], [17]. There exists other types of structural coverage criteria, including branch-coverage [20], Fault-Exposing-Potential (FEP) [20].

In this section, we focus on the related work on prioritization strategies, especially those that we use to compare with our proposed approaches.

A. Greedy Technique

The greedy technique [18], [2] attempts to select a test case with the best coverage. Under this guideline there are two strategies: the total strategy and the additional strategy. The total strategy always selects the one that offers the best coverage, in terms of certain criteria, among remaining test cases regardless what have already been chosen. The additional strategy selects the test case that covers the most statements, function or units specified by the criterion that have not been covered before.

B. Adaptive Random Technique

Adaptive Random Technique (ART) is a random-based test case prioritization strategy proposed by Jiang *et al*. [21].

Prior empirical study [21] has shown that using the *min* distance in ART typically leads to the best performance. Thus, in this paper, we have implemented ART based on [21] and chosen the *min* distance to compute the prioritized set.

C. Other Approaches

Mondal et al.[22] propose new approach for bi-objective optimization of diversity and test execution time, using α-Shape analysis of the Pareto front solutions. However, it utilize static method sequence for analysis. Several other test case prioritization techniques that leverage dynamic program information have been proposed. Li *et al*. [10] exploit search-based technique in test case prioritization. We have implemented their approach with the hope to compare it against our approach. However, our initial empirical study show that it is not scalable so we decide to exclude it from our experiments.

III. Our Approach

We treat the execution under each test case as a sub-graph and apply complex network theory into our approach. Our approach consists of three stages: Execution Monitoring, Graph Construction, and Sampling and Prioritization.

During execution monitoring, we instrument the source code to obtain traces that record the function call sequences under each test case. Each trace is represented as a graph. Then at the second stage, we integrate all the individual graphs into a total graph. To make it more efficient, the first two stages are interwoven and the total graph is built on the fly. Once the total graph is obtained, the we apply two strategies to prioritize the test cases: one is coverage based and the other is distribution based.

A. Graph Model

Graph plays a key role in our approach, thus we first explain the our graph model first. With the help of AspectJ-based [23] instrument tool *Kieker* [24], we are able to obtain the full signatures of an invoked method during an execution, including the method name, the number, types and values of its parameters, timestamps before and after the execution of the method, the global unique session number and trace number, the calling order and calling stack of the method. Based on the collected information we are able to create calling graphs [25], [26], [27] that model the method call relationship[28].

An example on how we collect method call traces and construct calling graphs is given in Figure 1. The left figure is the methodal call sequences obtained by the instrumentation of a particular execution, where *main* calls *func1* , *func1* calls *func2*, *func2* calls *func3*, and so on. However, a straightforward recording consumes significant amount of memory. We exploit weight w defined in calling graph to merge duplicated nodes in the graph. As shown in Figure 1 (b), there is only one node corresponding to each method. In the example, *main* calls *func1* twice so $w(main, func1) = 2$. Each node may have

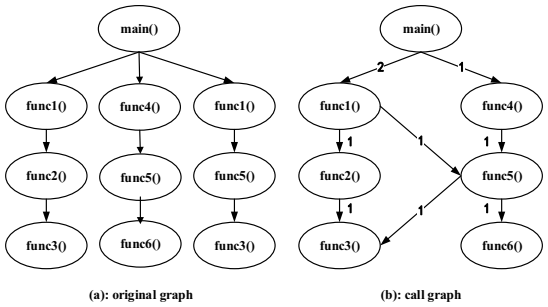

Fig. 1. Calling graph construction.

```
public static void org.apache.commons.io.IOUtils.closeQuietly(java.io.OutputStream) ->
    public static void org.apache.commons.io.IOUtils.closeQuietly(java.io.Closeable)

public void org.apache.commons.io.FileUtilsTestCase.testSizeOf() -> public static long
    org.apache.commons.io.FileUtils.sizeOf(java.io.File)

protected void org.apache.commons.io.FileUtilsTestCase.setUp() -> public static
    java.io.File org.apache.commons.io.testtools.FileBasedTestCase.getTestDirectory()
```

Fig. 2. Different types of method calls

multiple incoming and outgoing edges. As a result, instead of 10 nodes and 9 edges in Figure 1 (a), the corresponding calling graph has only 7 nodes and 8 edges.

Matrix $TG = [e_{i,j}]_{m \times n}$ is utilized to integrate the set of calling graphs $\{CG_1, \ldots CG_i\}$. $e_{i,j}$ is the frequency of node i calling node j.

B. Additional Greedy Method Call Sequence Strategy

We propose a prioritization strategy called additional greedy method call sequence strategy (AGC) that exploit method call sequences.

AGC follows the principle of greedy coverage strategy that always selects the test case that covers the most units that have not been covered so far. The principle is based on the hypothesis that the more newly covered units the better chance to reveal faults [18], [2]. Generally speaking, the coverage-based critirion at the fine granularity outperforms the critirion at coarse granularity in terms of fault detection capability, but at a cost of larger overhead [19]. In our opinion, the method call sequence based criterion is a good balance between statement coverage and function coverage. Compared with the function coverage criterion, we consider multiple methods instead of individual method in isolation. Thus our prioritization is based on richer information. Compared with statement coverage, our unit is method thus incurs less overhead.

Algorithm 1 gives the pseudo-code of AGC. Figure 2 illustrates three types of method calls (or edges in the graph) obtained from dynamic execution traces . The top one is a method call from source code to source code, which has the highest possibility of detecting faults. This is because all the faults are in the source code itself, not in the test code. We set the weight of this kind of edges to 2. The middle one is a method call from test code to source code, which has some possibility of detecting faults. We set the weight of such type of edges to 1. The lowest edge is called from test code to test code, which has no chance of detecting faults. Therefore the weight of such type of edges is set to 0 . The total weight of each test case is calculated by accumulating the weight of the edges of its calling graph.

Algorithm 1 Main process of AGC

Input: test suite T
1: **while** Any test cases haven't been calculated **do**
2: Calculate weight for each test case, weight = 0
3: **while** Any edges haven't been calculated **do**
4: **if** The edge is called from test to test code **then**
5: weight = weight+0
6: **else if** The edge is called from test to source code **then**
7: weight = weight+1
8: **else if** The edge is called from source to source code **then**
9: weight = weight+2
10: **end if**
11: **end while**
12: Add weight to candidate set $C = \{< t_1, w_1 >, < t_2, w_2 >, \ldots.\}$
13: **end while**
14: Use weight as coverage criterion, greedy additional strategy to prioritize T.
Output: test order T'

In AGC, we do not use w weight functions in CG because these test cases have many repetitions. The frequency of each edge is not as effective as calculating the specificity of each test case.

C. Random Precess Reduction

In the process of prioritization, some random selection behavior may affect the stability of strategies. A random selection may affect the fault detection efficiency especially when there are lots of such choices.

In order to reduce random process, we introduce Lexicographical Ordering proposed by Eghbali[29]. It augment additional greedy strategy by considering all the entities (method consequences, functions, statements) even if they have been covered in the previous steps. Entities that are covered less will be given higher priorities.

Generally speaking, lexicographical ordering can reduce but not eliminate randomness in the additional greedy strategy.

IV. Empirical study

In this section, we conduct an empirical study to answer the following two research questions. All the experiments are carried out on a Lenovo PC with Intel Core i7-4790 3.60GHz processor and 16GB DDR3 RAM.

1) RQ1: Is method call sequence an appropriate criterion that can improve the performance of test case prioritization in regression testing?
2) RQ2: Are the prioritization strategies based on the new criterion achieving a good trade-off between those based on function-coverage and statement-coverage?

A. Implementation and Subject Programs

In our implmentation, we use *Kieker* [24] that can dynamically instrument the classes loaded into the JVM through a Javaagent command without any modification to the source code. However, *Kieker* can only record coverage information at the method level. Although sufficient for our approach, it is not sufficient for the existing approaches that are based on statement coverage. Thus we also use the Java testing tools *Jacoco* [30] to collect coverage information at the statement

level. In order to measure fault detection rate, we inject faults into our subject programs by using Java mutation tool *MuJava* [31]. As concluded in previous work [32], [33], mutation faults are close to real faults and are suitable for software testing experiment.

We choose eight open source Java programs[1] [2] [3] [4] [5] [6] [7] that have been widely used in previous studies [14], [32] from the GitHub and Apache projects as our benchmark. For each program there are about 1% to 5% test cases that cannot be executed due to various reasons such as version mismatching and environment, so we remove these test cases. Each of these programs applies Junit auto testing framework.

Table I lists the eight subject programs that include their names (Column 1), versions (Column 2), lines of code (Column 3) and number of methods (Column 4). The number of edges in the total graphs is given in Column 5. Columns 6 and 7 give the number of test cases at class level and method level, respectively. The last column shows the number mutations generated by *MuJava*.

B. Design of the Empirical Study

In this empirical study, the only information we exploit is the execution information obtained by dynamic instrument tools. That is, we do not require extra information such as user requirements and historical code changes. This applies to all the approaches that we implement. Then we will explain the experimental progress we utilize.

Algorithm 2 Compare TCP

```
Start and choose the program
Filter original faults
Utilize mutation tools to inject mutation faults
Randomly choose five faults
if Faults are repetitive then
    return Step 4
end if
Creat faulty versions (1000 groups)
while All of the faulty versions have not been executed do
    Select a faulty version as the source code
    while Testing process hasn't been finished do
        Execute test suite in particular order
        if i_th Test case detects fault (s) then
            Examine the test case
            if Caused by inject mutations && Have not been detected then
                i_th Test case indeed detects fault (s)
            end if
        end if
    end while
    Calculate APFD metric
end while
Calculate average APFD metric
```

Algorithm 2 gives the pseudo-code on how to compare the effectiveness of different TCP techniques. Generally speaking, fault detection capability is widely used for evaluating TCP techniques. Before starting our experiments we use unit testing to eliminate buggy test cases in order to have controlled experiments. After that we have a reasonable assumption that the subject programs have no testing errors, we randomly choose 5 different mutation faults which are generated by *MuJava* and inject them into the program to simulate a faulty program version.

In order to measure the effectiveness of fault detecting capabilities for each prioritization technique, we choose Average Percentage of Faults Detected (APFD) metric, defined by Equation 1, that is widely utilized in Regression Testing domain [18], [2], [19], [20], [34].

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}, \quad (1)$$

where TF_i is the first fault detecting location that detects fault i in this prioritization order, n is the number of test cases, and m is the number of faults. Since we create 1000 faulty versions for each subject program, we calculate 1000 APFD values and utilize the average metric for the evaluation.

C. Experiments

1) Performance: In this subsection, we present experimental data to answer the two research questions. The goal of RQ1 is to justify the effectiveness of method call sequences in regression testing prioritization. Figure 3 shows the boxplots of the APFD values for all the TCP techniques, the x-axis represents different techniques as follows:

- AGC: Additional Greedy Method Call Sequence Technique;
- ASC: Additional Greedy Statement-coverage Technique;
- AFC: Additional Greedy Function-coverage Technique;
- TFC: Total Greedy Function-coverage Technique;
- ART: Adaptive Random Technique;
- NO: Natural Order (Represent the result without any prioritization, which is tested in alphabetical order).

In Table II, the TCP technique that has the best performance is marked in red and the next best is marked in blue.

Based on these boxplots, we can make the following some observations. Firstly, an interesting phenomenon is that with the size of the programs size grows, the performance of AGC also has a continuous growth trend compared with traditional TCP techniques. For example, in the program *Commons.io*, the median APFD values of the AGC, ASC and AFC are 0.7766, 0.7998, and 0.7899, respectively. In the program *Commons.lang*, the median APFD value of the ASC technique is 0.6762 followed by AFC, AGC, TFC, ART and NO.

However, in large subject programs such as *Jfreechart, Google Closure Compiler* with LOC ranging from 56039 to 140237, the performance of AGC is comparable or even better than ASC. For example, in the program *Google Closure Compiler*, the median APFD value of AGC is 0.9113, followed by ASC, ART, AFC, TFC and NO.

Previous results in RQ1 have shown that the method call sequence based criterion is competitive with function-coverage criterion at the same granularity and performs better in big-sized programs. Even when comparing with finer granularity, it

[1] http://commons.apache.org/proper/commons-io/
[2] http://commons.apache.org/proper/commons-lang/index.html
[3] http://www.joda.org/joda-time/
[4] http://commons.apache.org/proper/commons-math/
[5] http://www.jfree.org/jfreechart/
[6] http://ant.apache.org/
[7] http://closure-compiler.appspot.com/home

TABLE I
BASIC INFORMATION OF PROGRAMS (ORDERED BY LOC)

Subject Programs	version	LOC	Methods	Edges	TCnum (class-level)	TCnum (method-level)	Mutant num
Commons.io	2.4	9957	764	1227	84	903	9241
Commons.lang	3.5	26578	2143	4292	137	2883	37466
Jodatime	2.1	27213	3583	11412	154	3975	38378
Commons.math	2.2	56039	3936	9584	264	1859	192791
Jfreechart	1.0.19	98335	6974	16086	359	2300	37271
Apache.ant	1.9.7	108132	8195	21756	233	1907	70320
Commons.math3	3.6.1	105191	7257	20251	510	4545	339774
Google Closure Compiler	v20160713	140237	10774	49182	306	10824	19935

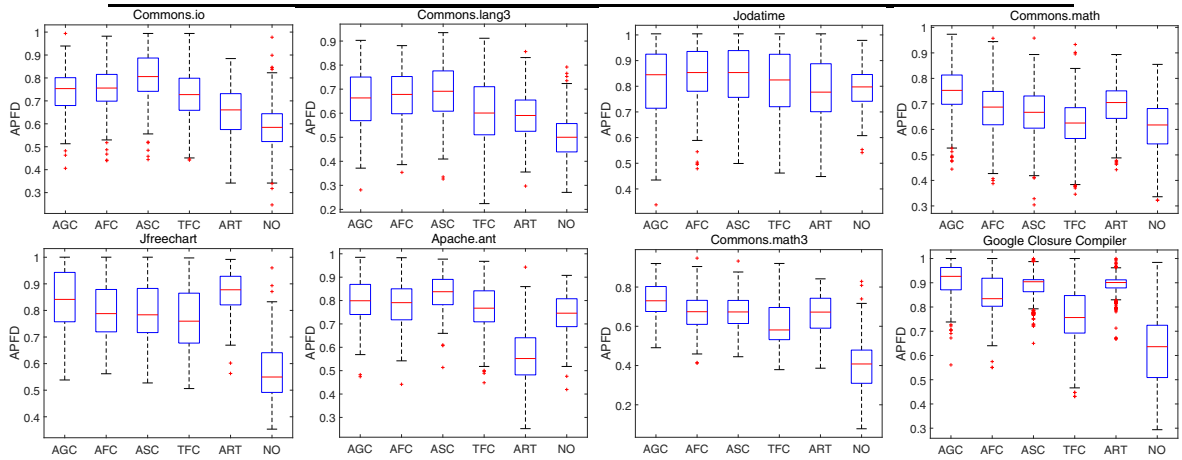

Fig. 3. Result for our techniques and traditional TCP techniques on 8 open source programs. The box and whisker plots represent the values of APFD metric for different TCP techniques. The x-axis represents the different techniques and the y-axis represents the APFD values. The central box of each plot represents the values from 25 to 75 percentage.

TABLE II
COMPARISON OF AVERAGE APFD VALUES (%) FOR DIFFERENT TCP TECHNIQUES

Subject Programs	AGC	ASC	AFC	TFC	ART	NO
Commons.io	77.66	79.98	78.99	72.11	62.10	58.71
Commons.lang	66.53	67.62	66.94	60.70	59.32	50.04
Jodatime	80.30	83.69	81.81	81.00	78.07	79.16
Commons.math2	67.64	74.69	66.27	62.18	69.47	60.81
Jfreechart	82.84	78.47	78.78	76.15	87.05	56.52
Apache.ant	78.62	82.59	78.21	75.67	55.40	73.08
Commons.math3	73.15	67.43	66.37	60.83	66.15	39.58
Google Closure Compiler	91.13	89.44	84.41	76.52	89.52	62.64

TABLE III
PRIORITIZATION COST FOR THE THREE DIFFERENT COVERAGE CRITERIA TECHNIQUES.

Subject Programs	Time Cost (AFC)	Time Cost (AGC)	Time Cost (ASC)
Commons.io	1s	1s	1s
Commons.lang	2s	13s	65s
Jodatime	5s	301s	1361s
Commons.math2	8s	139s	322s
Apache.ant	43s	309s	1093s
Google Closure Compiler	390s	7794s	22546s
Jfreechart	16s	193s	468s
Commons.math3	60s	806s	2597s

outperforms statement-coverage in large programs *Jfreechart, Commons.math3* and *Google Closure Compiler*.

2) Time Usage: Table III answers the RQ2 by comparing the time usage of the three additional greedy strategies ASC, AFC and AGC that are based on three criteria statement, function, and method call sequence, respectively. Other techniques are not in this table because we just want to compare the effectiveness of different granularities, not sampling strategies. We try our best to use the similar data structure in order to avoid the noise and the interrupt of the raw data. It can be observed that AFC is most efficient, followed by AGC and then ASC. AGC uses one-third to one-fifth of the time used by ASC. Overall, if developers want to obtain a sweet spot between fault detection capability and prioritization cost, we believe our method call sequence technique is competitive with other TCP techniques, especially in large programs.

3) Evaluation: Experiments above shows that our AGC technique performs better in programs of big size, even better than statement-granularity in some programs. The reason that we think is the discrimination of test cases in different granularities.

We counted th entity growth in eight programs. Based on the space constrains, the result figure is not given in this paper. We utilize three prioritization techniques in different granularities. Each technique we count the percentage of entity coverage

(Function, Call Sequence, Statement) by executed test cases. As the same reason in time usage experiment, other techniques are not considered in order to control variables.

We find that when AGC performs better than AFC, the curve of AGC is detached obviously with the curve of AFC. In other words, test cases in edge(call sequence)-granularity are more discriminable than those in function-granularity. Those test cases with more unique call sequences will be labeled as fault-prone candidates and put into the front of the test order. Big-sized programs contain complex structure and more edges, but not the same growth rate of functions. That is the reason we think our technique shows up a growth trend of performance with the size of the program increases.

V. CONCLUSION

In this paper, we presented a TCP criterion that is based on method call sequences. As traditional coverage-based TCP techniques often face the problem to balance the prioritization efficiency and effectiveness. We believe the edge information that corresponds to method call sequences in our graph model offers a good trade-off between the two factors. Based on the new criterion we have implemented the prioritization algorithm AGC.

We have conducted experiments on our graph-based TCP techniques and other traditional TCP techniques on eight open source programs. Experimental results indicate that AGC is particular effective on large programs. The reason we believe is that these programs have complex structural information and some bugs are hard to be detected by the traditional function-coverage criterion. As for efficiency, as expected our approach is between the function-coverage and statement-coverage techniques. Edge information in our graph model performs well as a trade-off between fault detection capability and prioritization cost.

VI. ACKNOWLEDGMENT

This work was supported by National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61772408, 61721002, 61532015,61702414) and Ministry of Education Innovation Research Team (IRT_17R86)

REFERENCES

[1] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *ICSE*. ACM, 2016, pp. 535–546.

[2] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *ICSM*. IEEE, 1999, pp. 179–188.

[3] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *Journal of Systems and Software*, vol. 85, no. 3, pp. 626–637, 2012.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[5] R. H. Rosero, O. S. Gmez, and G. Rodrguez, "15 years of software regression testing techniquesa survey," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 05, pp. 675–689, 2016.

[6] D. Hao, L. Zhang, and H. Mei, "Test-case prioritization: achievements and challenges," *Frontiers of Computer Science*, vol. 10, no. 5, pp. 769–777, 2016.

[7] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *TSE*, 2012.

[8] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *TOSEM*, vol. 24, no. 2, p. 10, 2014.

[9] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, 2011.

[10] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *TSE*, vol. 33, no. 4, 2007.

[11] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE*. IEEE Computer Society, 2001, pp. 329–338.

[12] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia, "The impact of test suite granularity on the cost-effectiveness of regression testing," in *ICSE*. ACM, 2002, pp. 130–140.

[13] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *SQJ*, vol. 21, no. 3, pp. 445–478, 2013.

[14] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *FSE*. ACM, 2016, pp. 559–570.

[15] S. McMaster and A. Memon, "Call-stack coverage for gui test suite reduction," *TSE*, vol. 34, no. 1, pp. 99–115, 2008.

[16] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *ICSM*. IEEE, 2013, pp. 540–543.

[17] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[18] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *TSE*, vol. 27, no. 10, pp. 929–948, 2001.

[19] S. Elbaum, A. G. Malishevsky, and G. Rothermel, *Prioritizing test cases for regression testing*. ACM, 2000, vol. 25, no. 5.

[20] ——, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[21] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *ICSE*. IEEE Computer Society, 2009, pp. 233–244.

[22] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in *ICST*. IEEE, 2015, pp. 1–10.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *European Conference on Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[24] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012, pp. 247–248.

[25] Y. Qu, X. Guan, Q. Zheng, T. Liu, J. Zhou, and J. Li, "Calling network: A new method for modeling software runtime behaviors," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–8, 2015.

[26] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *ISSRE*. IEEE, 2016, pp. 24–35.

[27] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *TIFS*.

[28] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *TIFS*, 2018.

[29] S. Eghbali and L. Tahvildari, "Test case prioritization using lexicographical ordering," *TSE*, vol. 42, no. 12, pp. 1178–1195, 2016.

[30] M. Hoffmann, B. Janiczak, E. Mandrikov, and M. Friedenhagen, "Jacoco code coverage tool. online, 2009," 2016.

[31] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system," *STVR*, 2005.

[32] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE*. ACM, 2014, pp. 654–665.

[33] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*. ACM, 2005, pp. 402–411.

[34] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri, "Understanding the effects of changes on the cost-effectiveness of regression testing techniques," *Software testing, verification and reliability*, vol. 13, no. 2, pp. 65–83, 2003.