The Journal of Systems and Software 163 (2020) 110539

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Relation-based test case prioritization for regression testing

Jianlei Chi [a], Yu Qu [c,*], Qinghua Zheng [a], Zijiang Yang [b], Wuxia Jin [a], Di Cui [a], Ting Liu [a]

[a] Ministry of Education Key Lab for Intelligent Network and Network Security, Xi'an Jiaotong University, Xi'an, Shaanxi, 710049 China
[b] Department of Computer Science, Western Michigan University, Kalamazoo, MI, 48167 USA
[c] Department of Computer Science and Engineering, University of California, Riverside, 900 University Ave, Riverside, CA, 92521 USA

ARTICLE INFO

Article history:
Received 12 January 2019
Revised 8 January 2020
Accepted 29 January 2020
Available online 30 January 2020

Keywords:
Software testing
Test case prioritization
Dynamic call sequence

ABSTRACT

Test case prioritization (TCP), which aims at detecting faults as early as possible is broadly used in program regression testing. Most existing TCP techniques exploit coverage information with the hypothesis that higher coverage has more chance to catch bugs. Static structure information such as function and statement are frequently employed as coverage granularity. However, the former consumes less costs but presents lower capability to detect faults, the latter typically incurs more overhead.

In this paper, dynamic function call sequences are argued that can guide TCP effectively. Same set of functions/statements can exhibit very different execution behaviors. Therefore, mapping program behaviors to unit-based (function/statement) coverage may not be enough to predict fault detection capability. We propose a new approach AGC (Additional Greedy method Call sequence). Our approach leverages dynamic relation-based coverage as measurement to extend the original additional greedy coverage algorithm in TCP techniques.

We conduct our experiments on eight real-world java open source projects and systematically compare AGC against 22 state-of-the-art TCP techniques with different granularities. Results show that AGC outperforms existing techniques on large programs in terms of bug detection capability, and also achieves the highest mean APFD value. The performance demonstrates a growth trend as the size of the program increases.

1. Introduction

Software evolves continuously during development and maintenance on account of numerous reasons such as adding new features, modifying old features and refactoring. Existing test suite is widely utilized by regression testing aims at detecting regressions, validating software changes and avoiding the introduction of new bugs. Lu et al. (2016). However, previous studies report that the cost of regression testing can be very high. One regression testing may last for more than seven weeks (Rothermel et al., 1999; Huang et al., 2012), which can account for as much as one-half of the software maintenance cost (Kaner, 1997; Chittimalli and Harrold, 2009). Therefore, measures of saving testing budgets are necessary and indispensable.

Test Case Prioritization (TCP) is one of the measures to alleviate the budget of regression testing, which can increase the likelihood of revealing faults earlier in the testing process. Existing algorithms usually exploit **unit-based** information such as function or statement (Yoo and Harman, 2012; Elbaum et al., 2001; Rothermel et al., 2002; Catal and Mishra, 2013; Luo et al., 2016) as common coverage features to guide TCP, based on the hypothesis that a test case with a higher coverage rate has a higher probability to detect faults. Coverage criterion at coarser granularity, such as function coverage, gives faster but less accurate prioritization; while a finer granularity such as statement coverage, typically detects faults accurately at a cost of more overhead. Both function and statement coverage are widely used in TCP today (Luo et al., 2016) and various studies have examined their trade-off (McMaster and Memon, 2008; Marijan et al., 2013; Thomas et al., 2014). *Furthermore, relation-based information such as branch or path naturally contains richer content than unit-based information. It inspires us the way to improve TCP performance.*

Various prioritization strategies can be adopted based on the criterion. For example, in greedy strategy, (Rothermel et al., 1999; 2001) test cases can be ranked by their absolute coverage, i.e.

* Corresponding author.
E-mail addresses: chijianlei@stu.xjtu.edu.cn (J. Chi), yqu@sei.xjtu.edu.cn (Y. Qu), qhzhengg@xjtu.edu.cn (Q. Zheng), zijiang.yang@wmich.edu (Z. Yang), wx_jin@sei.xjtu.edu.cn (W. Jin), cuidi@sei.xjtu.edu.cn (D. Cui), tliu@sei.xjtu.edu.cn (T. Liu).

https://doi.org/10.1016/j.jss.2020.110539
0164-1212/

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. However, structural coverage may not be the best criteria to guide the prioritization of dynamic executions (Fang et al., 2014). Existing techniques map program behaviors to function or statement coverage may be too simplistic to detect faults. Moreover, such abstraction leads to information loss and thus may lead to inaccurate fault detection.

In this paper, method call sequence is exploited to describe program behavior relations and guide TCP. Compared with unit-based coverage, **relation-based** coverage method call sequence is a better indicator to map 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 larger number of call sequences should be considered priorly. We believe that the reduction of call sequence is more precise than those obtained by unit-based information. Treating program behaviors as a calling network, our approach considers both vertices and paths along the edges while static structure coverage only considers vertices.

We propose a new prioritization strategy called Additional Greedy Call sequence (AGC). In this strategy we extended traditional additional greedy algorithm and reduce the random process in the algorithm to lift accuracy. We have implemented our approach and other 22 state-of-the-art techniques, comparing their performance on eight real-world open source Java projects and a real faulty dataset. Experimental results exhibit that granularity indeed makes a difference. Our approach outperforms other state-of-the-art prioritization techniques in the same granularity and achieves the best average fault detection capability (APFD). Compared with the finer granularity technique, our approach is better on average fault detection capability metric with one-third to one-eighth cost. That is, our approach indeed is much more efficient than the approach based on unit-based coverage. We also compare AGC with multiple similarity-based approach, result shows that AGC outperforms these techniques. As a conclusion, our approach performs well as a trade-off between fault detection capability and prioritization cost.

The paper makes the following contributions:

1. Dynamic criterion based on method call sequence to guide test case prioritization;
2. Proposing relation-based prioritization strategy AGC, the best average fault detection capability among other comparing strategies.
3. Real-world mutation environment and faulty dataset experiments.

The rest of this paper is organized as follows. Section 2 explains the motivation of this work. Section 3 summarizes the related work, followed by a detailed explanation of our approach in Section 4. In Section 5, we present our empirical study. The threats to validity are discussed in Section 6 and Section 7 introduces the future work. Finally, Section 8 concludes the paper.

2. Motivation example

Consider the function *writeSwappedInteger* given in Figs. 1 and 3 with four test cases *k=1, k=2, k=3* and *k=4*. There is a static array *arrays* initialized with length 3 at Line 1. An *ArrayIndexOutOfBoundsException* fault occurs in *func4* under user input *k=4*. This simplified function is part of the package *apache.commons.io* whose complete call sequences are depicted in Fig. 1. In the graph a vertex represents a function and edge $f_1 \rightarrow f_2$ indicates that f_1 calls f_2. The amplified sub-figure gives the call sequences of *writeSwappedInteger*, which is magnified at the bottom of Fig. 3. The executions under the four test cases produce the call sequences:

1. $TC_1 = \pi_{k=1} = \langle func1, func2, func3, func4\rangle$;
2. $TC_2 = \pi_{k=2} = \langle func5, func6, func7, func8\rangle$;
3. $TC_3 = \pi_{k=3} = \langle func5, func6, func7, func9\rangle$;
4. $TC_4 = \pi_{k=4} = \langle func1, func6, func3, func8\rangle$.

If the test case TC_2 has been selected first and unit-based function coverage is used as the criterion to prioritize test cases, TC_1 will be select as the next one since it contains four not-covered functions. Then the TC_3 that contains only one will be put into the test order. Finally, since the TC_4 does not contain any not-covered function, it will be given a low priority and put into the bottom of the test order. At this moment this selection is reasonable because it is hard to predicate which path may lead to bugs. However, the selection order of $\{TC_2, TC_1, TC_3, TC_4\}$ significantly delays the testing under TC_4 because the functions covered by $\pi_{k=4}$ are a subset of those covered by $\pi_{k=1}$ and $\pi_{k=2}$. That is, since an execution under TC_4 does not cover any new functions, the test case is very unlikely to be checked under test case prioritization. The similar scenario happens under statement coverage based criteria.

Another motivating example is given in Fig. 2, the executions under the four test cases produce the call sequences:

1. $TC_1 = \pi_{k=1} = \langle func1, func2, func3, func4\rangle$;
2. $TC_2 = \pi_{k=2} = \langle func1, func2, func4, func3\rangle$;
3. $TC_3 = \pi_{k=3} = \langle func2, func1, func3, func4\rangle$;
4. $TC_4 = \pi_{k=4} = \langle func4, func3, func2, func1\rangle$.

When the test case TC_1 has been selected as the first order, as for unit-based function coverage, TC_2, TC_3, TC_4 has the same priority due to the reason that there are no not-covered functions for these three test cases. Strategy based on unit-based function coverage will randomly choose one test case as the next order and repeat this action until there is no candidate. However, If we consider sequential information between function entities, although TC_2, TC_3, TC_4 has the same set of function entity compared with TC_1, they exhibit very different execution behaviors. TC_2 and TC_3 contain less not-covered sequences than TC_4. Therefore, TC_4 will be selected as the next order. The similar scenario happens under other unit-based criteria.

In this paper, we propose a new test case prioritization technique that is based on call sequences. Consider four traces in Fig. 3 and TC_2 will be selected first again. TC_1 and TC_3 contain the same or fewer sub-sequences than that of TC_4, they do not delay TC_4. This is because that none of the sequences $func1 \rightarrow func6$, $func6 \rightarrow func3$, $func3 \rightarrow func8$, $func1 \rightarrow func6 \rightarrow func3 \rightarrow func8$ are covered by $\pi_{k=1}$ or $\pi_{k=2}$ or $\pi_{k=3}$. It is straightforward to believe that our call-sequences-based technique can map program execution behaviors more accurately.

3. Background and related works

In order to alleviate the cost of regression testing, three main techniques have been proposed (Yoo and Harman, 2012; Wong et al., 1997): Test Case Minimization, Test Case Selection and Test Case Prioritization. Test suite minimization aims to remove redundant test cases, the size of the test suite is reduced. Test case selection considers the changes between the current and previous versions, selects only those test cases that are relevant to the changes. Neither test suite minimization nor test case selection guarantees the integrity of the test suite. Test Case Prioritization (TCP) seeks to find the ideal ordering of the test cases, so that a regression testing obtains a maximum benefit under limited resources or when the testing process is prematurely halted at some arbitrary point.

Definition 1 gives the formal definition of TCP.

Definition 1 (Elbaum et al., 2000)**.** Given a test suite *T*, the set *PT* is all the permutations of *T*. *f* is a function from *PT* to the real

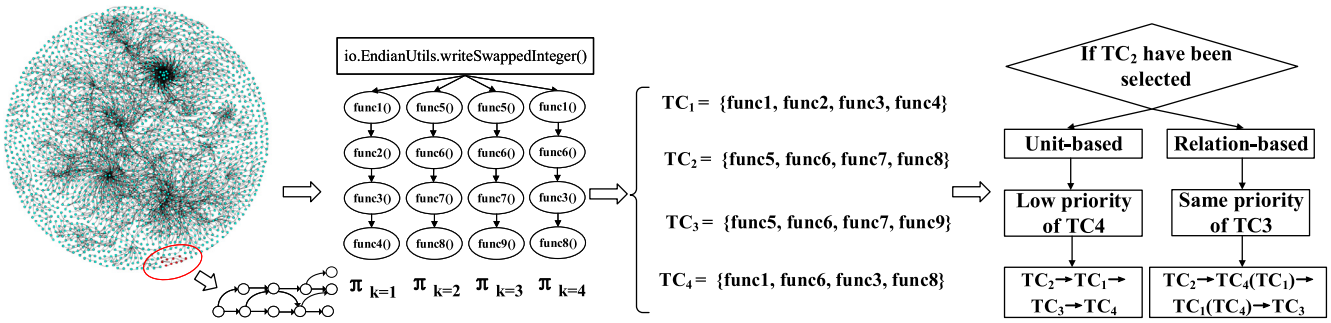

Fig. 1. Motivation Example 1.

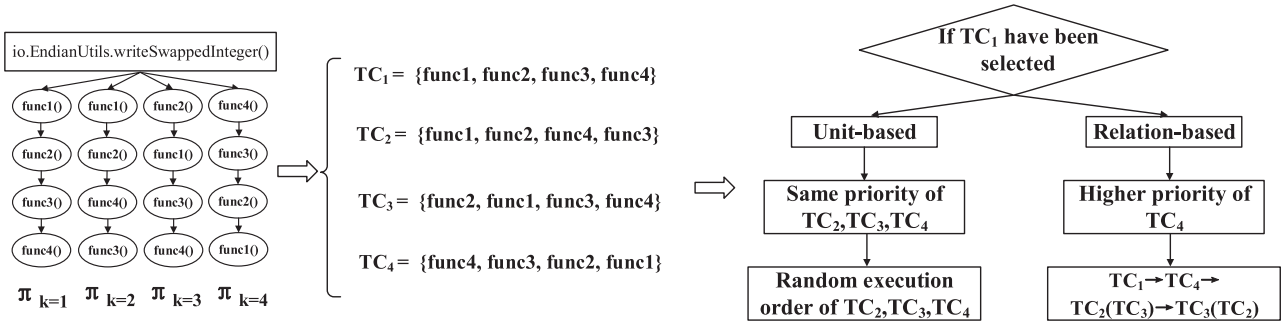

Fig. 2. Motivation Example 2.

```
static int[] arrays = new int[3];
public static void main(String[] args) {
      try {int k = System.in.read();
          switch(k){
          case 1:func1(k);
          case 2:func5(k);
          case 3:func5(k);
          case 4:func1(k);}}
      catch (Exception e) {e.printStackTrace();}
}
public static void func1(int k){switch(k){
      case 1:func2(k);
      case 4:func6(k);}
}
public static void func2(int k){func3(k);}
public static void func3(int k){switch(k){
      case 1:func4(k);
      case 4:func8(k);}
}
public static void func4(int k){arrays[k-1]=k;}
public static void func5(int k){func6(k);}
public static void func6(int k){switch(k){
      case 2:func7(k);
      case 3:func7(k);
      case 4:func3(k);}
}
public static void func7(int k){switch(k){
      case 2:func8(k);
      case 3:func9(k);}
}
public static void func8(int k){
      arrays[k-1]=k;//bug when k=4
}
public static void func9(int k){arrays[k-1]=k;}
```

Fig. 3. A Snippet of "writeSwappedInteger".

numbers. The problem of test case prioritization is to find $T' \in PT$ such that $\forall(T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

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.

There exist various concrete ways to achieve test case prioritization. A large number of approaches (Yoo and Harman, 2012; Rosero et al., 2016; Hao et al., 2016; Mei et al., 2012; Hao et al., 2014; Jia and Harman, 2011; Li et al., 2007) have been proposed that mainly focus on two steps of test case prioritization. Some studies focus on the criterion that measures the effectiveness of a test case and others concentrate on the strategy that exploits the criterion to prioritize the test cases.

Previous studies normally exploit unit-based information (Yoo and Harman, 2012; Elbaum et al., 2001; Rothermel et al., 2002; Catal and Mishra, 2013; Luo et al., 2016) as the criteria, based on the hypothesis that a test case with a higher coverage rate has a better chance to detect faults. Function coverage (Do et al., 2004) and statement coverage (Rothermel et al., 1999) are widely used in TCP today (Luo et al., 2016). Various studies have examined their trade-offs of time cost and fault detection capability (McMaster and Memon, 2008; Marijan et al., 2013; Thomas et al., 2014). There exist other types of coverage criteria, including branch-coverage (Elbaum et al., 2002), Fault-Exposing-Potential (FEP) (Elbaum et al., 2002), transition and round-trip coverage (Xu and Ding, 2010). Method call sequence is also be utilized by some works. However, the works of Zhang et al. (2012); Hemmati et al. (2011) obtained the method call sequence from static analysis. In this work, we obtain method call sequence from dynamic execution due to the reason that we think regression testing is a good environment to make use of history data for future testing. Dynamic analysis gains more accurate method call sequences than static analysis. Wang et al. (2015) and Fang et al. (2014) have also mentioned method call sequence, but they just utilized it to calculate edit distance for clustering. None of them used the call sequence as coverage criteria. We have added their cluster-based method GC, FCS, GOS

Table 1
Strategy example.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
t_1	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**
t_2	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**
t_3	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
t_4	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

in different granularities as the baseline. It should be noticed that the Greed-aided-clustering Ordered Sequence (GOS) algorithm has serious bugs. We will discuss it in Section 5. The work of Noor and Hemmati (2015) mainly focuses on proposing similarity-based metrics, which is different from our coverage-based technique.

In this section, we focus on related prioritization strategies, especially those that we exploit to compare with our proposed approach. In order to facilitate the understanding of different strategies, we use an example given in Table 1. In the table there are eight functions $f_1 \dots f_8$ and four test cases $t_1 \dots t_4$. If an execution under t_i covers f_j, the corresponding cell has value 1; otherwise the value is 0.

3.1. Greedy technique

The greedy technique (Rothermel et al., 2001; 1999) attempts to select a test case with the best coverage rate. 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 of what has already been chosen. The additional strategy selects the test case that covers the most statements, function or units specified by the criterion that has not been covered before.

Assume that function coverage is used as the criterion in Table 1. The total strategy chooses t_1 first because it covers 5 functions. Then t_2 is chosen because it covers four functions. The remaining t_3 and t_4 are selected randomly because both of them cover 3 functions. That is, the test case order given by total strategy is either $T = \langle t_1, t_2, t_3, t_4 \rangle$ or $T = \langle t_1, t_2, t_4, t_3 \rangle$.

After choosing t_1, the additional selects t_4 as the second test case because it covers $\{f_4, f_5\}$ that have not been covered by t_1. On the other hand, both t_2 and t_3 cover one function that has not been covered by t_1. Next, t_2 is selected because it covers f_6 that is covered neither by t_1 nor t_4. Thus, the only possible test case order given by additional strategy is $T = \langle t_1, t_4, t_2, t_3 \rangle$.

Both greedy strategies are straightforward but effective so that they are often used as the baseline in the evaluation of existing studies. Our implementation is based on the work by Rothermel et al. (1999). A previous study (Jiang et al., 2009) has shown that additional strategy outperforms total strategy. Fang et al. (2014) utilized cluster merge strategy to reduce the global additional greedy cost, we will compare with this baseline and discuss if this cluster-based technique is effective.

3.2. Similarity-based Techniques

Similarity-based techniques are introduced to utilize the execution profiles of test cases (Dickinson et al., 2001; Jiang et al., 2009; Hemmati and Briand, 2010; Yan et al., 2010; Zhang et al., 2010; Wang et al., 2015). They are mainly divided into two categories: distribution-based and Adaptive Random Testing (ART). Distribution-based techniques cluster test cases according to their dissimilarities. Clusters can be utilized to guide test case selection and test case prioritization. The purpose of similarity-based techniques is to maximize diversity (i.e. minimize similarity) of selected test cases. The diversity of test cases is computed by a certain dissimilarity measure between each pair of test cases.

In this paper we choose distribution-based techniques Graph Similarity and Clustering Technique (GC), Function Call Sequence (FCS) (Wang et al., 2015) and ART (Jiang et al., 2009) as representative Similarity-based Techniques.

The original work of Wang et al. (2015) chooses three edit distances which treat profile information as vector, sequence and tree respectively. In this paper, we implement the same prioritization strategy but choose two edit distance for comparison. GC is based on our graph model which chooses graph edit distance (Kinable and Kostakis, 2011) to calculate the similarity between each entity pair. Each test case is regarded as a directed call graph. We will introduce the graph model more clearly in the next section. After computing the graph similarity, K-means clustering algorithm is utilized for clustering the similar test cases together and guide TCP process.

FCS technique also utilizes function call sequence, but it is just used to calculate the Levenshtein edit distance for clustering. In general, GC and FCS are mainly based on cluster technique and sample strategy which chooses the test case from each cluster and forms the testing order.

Adaptive Random Technique (ART) is a random-based test case prioritization strategy proposed by Jiang et al. (2009). There are two sets maintained by ART: Prioritized set T and candidate set C. At the beginning, T is empty and C contains all the test cases. In their original work, the first test case is randomly chosen. However, we have found that the performance is uneven so we select the first to be the one that covers the maximum number of functions or statements. After selecting the first test case and moving it from C to T, ART calculates a distance array by computing the distance between every pair of test cases between C and T. The distance between candidates A and B normally refers to Jaccard distance that is computed by $D(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$. Finally, ART exploits the *maxmin* sampling to select the farthest and the most particular test case from already selected test cases. In other words, the most representative test case in the candidate set. The procedure repeats until all the test cases have been selected. In the max-min sampling, there are three types of distances, *min, avg, max*, that can be utilized to calculate the distance $f(D)$ between the candidate set and the prioritized set.

In fact, Jiang et al. (2009) proposed 3 criteria (*maxmin, average* and *maxmax*). However, Jiang said in their paper that the *maxmin* group is more sensitive to different levels of coverage information than the *maxavg* and *maxmax* groups. The prior empirical study (Jiang et al., 2009) has also shown that using the *min* distance in ART typically leads to the best performance. Thus, in this paper, we have implemented ART based on (Jiang et al., 2009) and chosen the *maxmin* method to compute the prioritized set. Using Table 1 as an example, t_1 is selected first because it covers the most number of functions. The distances between t_1 and $\{t_2, t_3, t_4\}$ are {1/2, 2/3, 6/7}, so t_4 is added to T and T becomes $\{t_1, t_4\}$. Next, the distance between t_1 and $\{t_2, t_3\}$ is {1/2, 2/3} with the minimum value being 1/2. The distances between t_4 and $\{t_2, t_3\}$ is {5/6, 4/5} with the minimum being 4/5. To maximize the minimum distance, ART chooses t_3. Therefore, the prioritized test case by ART is $T = \langle t_1, t_4, t_3, t_2 \rangle$.

3.3. Search-based Techniques

Li et al. (2007) introduced the meta-heuristic search algorithm into the TCP domain. They proposed two search-based algorithms, hill-climbing and genetic-based algorithm. The hill-climbing algorithm for TCP searches all the neighbors and locates the ones that can achieve the largest increases in fitness. The genetic-based algorithm (GA) represents a class of adaptive search techniques based on the processes of natural genetic selection according to Darwinian theory of biological evolution (Holland, 1992). In

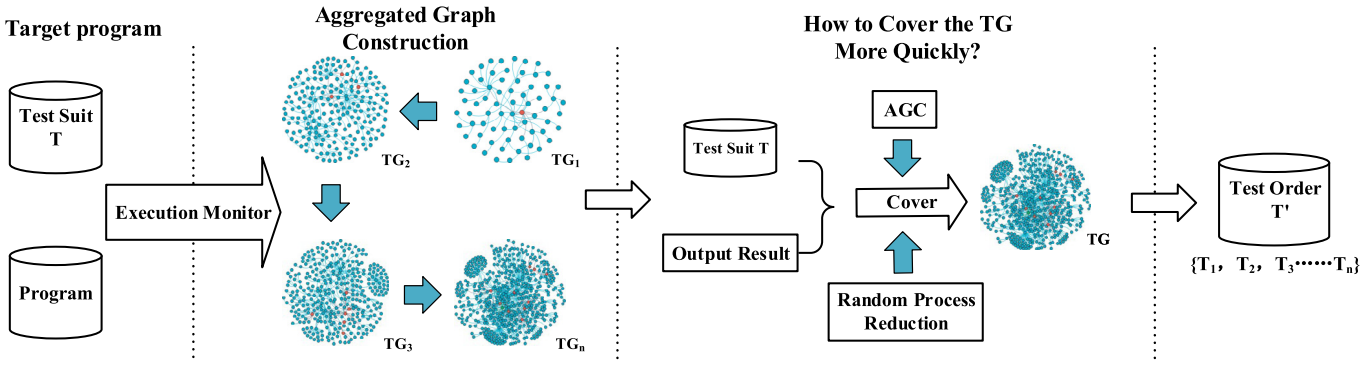

Fig. 4. Overview of relation-based approach.

this paper, we choose the GA approach as the representative search-based technique because Li had demonstrated that GA is more effective in fault detection (Li et al., 2007).

3.4. Other Approaches

Mondal et al. (2015) proposed a new approach for bi-objective optimization of diversity and test execution time, using α-Shape analysis of the Pareto front solutions. However, they utilized static method sequence for analyzing. Several other test case prioritization techniques that leverage dynamic program information have been proposed. Islam et al. (2012) presented an approach that recovers traceability links between system requirements and test cases using IR techniques (FIX). Korel et al. (2007) proposed a model-based test prioritization technique that uses the system modeling to model state-based systems and prioritizes test cases. Kim and Porter (2002) proposed a so-called history-based test prioritization technique that exploits historical execution data in resource-constrained environments. However, these techniques generally need extra information than coverage information such as execution cost or user knowledge.

4. Our approach

We introduce relation-based information to guide TCP process. The execution traces under each test case are treated as a subgraph of the program. The overview of our approach is given in Fig. 4, which contains three stages: Execution Monitoring, Aggregated Graph Construction, Sampling and Prioritization.

We instrument the program code to obtain traces during the execution that record the function call sequences under each test case. These traces are represented as a call graph. Then at the second stage, we integrate all the individual graphs into an aggregated total graph. To make it more efficient, the first two stages are interwoven and the aggregated graph is built on the fly. Once the aggregated graph is obtained, our AGC strategy is applied and sample the testing order based on method call sequence coverage criterion.

4.1. Graph model

We will firstly explain our graph model, it plays a important role in our approach. With the help of AspectJ-based (Kiczales et al., 2001) instrumentation tool *Kieker* (Van Hoorn et al., 2012), we 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 (Graham et al., 1982; Qu et al., 2015) that model method call relationships.

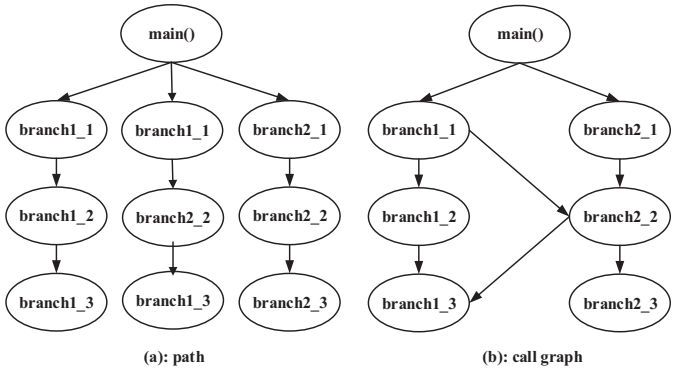

Fig. 5. Calling graph construction.

Definition 2. A calling graph is a directed network $CG = (V, E)$ where the set of nodes V represents the set of methods in a program, and the set of directed edges E represents the method invocation relation. Let m_i denotes the method that v_i refers to. Then $v_i \rightarrow v_j \in E$ if and only if m_i has at least one method invocation that calls m_j.

Definition 2 gives the definition of the calling graph. An example about how we collect method call traces and construct calling graphs is given in Fig. 5. The left figure is the method 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 a significant amount of memory. Thus, we present call graph as shown in Fig. 5 (b), there is only one node corresponding to each method. In the example, each node may have multiple incoming and outgoing edges. As a result, instead of 10 nodes and 9 edges in Fig. 5 (a), the corresponding calling graph has only 7 nodes and 8 edges.

Each calling graph corresponds to an execution under one single test case. Then, we form an aggregated graph that integrates all the calling graphs into a single graph.

Matrix $TG = [e_{i,j}]_{m \times n}$ is utilized to integrate the set of calling graphs $\{CG_1, \ldots CG_i\}$. e_{ij} is the frequency of node i calling node j. *TG* records each call sequence in all test cases. TG records each call sequence in all test cases. Each row represents a caller function entity and each column represents a callee function entity. For example, if *func*1 calls *func*2, a serial number of *func*1 is x and *func*2 is y, $TG[x][y]$ will be labeled to 1. We have thought about whether we should consider the frequency of each call sequence. But we

```
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. 6. Different types of method calls.

find that some test cases contain a large number of loop or repetitiveness. Weight may not be useful in this scenario.

4.2. Additional greedy method call sequence strategy

We propose a prioritization strategy called Additional Greedy method Call sequence strategy (AGC) that exploits method call sequences.

It is based on the greedy coverage strategy that always selects the test case that covers the most units that have not been covered so far. Then, we extend the original additional greedy strategy. It is based on the hypothesis that the more newly covered units the better chance to reveal faults (Rothermel et al., 1999; 2001). However, it is worth noting that in traditional additional greedy coverage strategy, numbers of additional functions or statements or branches are exploited to calculate the priority. In our AGC technique, we define three kinds of edges and calculate the weight for prioritization. Another problem for traditional additional greedy coverage strategy is that if some test cases contain the same number of coverage entities, it is hard to decide which test case should be chosen and randomly choose one as the next, which may causes performance reduction. We will introduce the solution called Lexicographical Ordering to reduce random selection in the next subsection.

In general, the coverage-based criterion at the fine granularity outperforms the criterion at coarse granularity in terms of fault detection capability, but at a cost of larger overhead (Elbaum et al., 2000). 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 the individual method in isolation. Our technique is naturally superior to function-level coverage. Compared with statement coverage, our unit is method thus incurs less overhead.

Algorithm 1 gives the pseudo-code of AGC. Figure 6 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 probability to detect 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. Using Table 1 as an example, in AGC, each value of f_i can be 0 or 1 or 2. The total weight of each test case is calculated by accumulating all of the edge weights.

Each circulation will select one test case t that contains the maximum number of important edges (may not the most) and put it into the prioritization order T'. If there are some test cases that contain the same number of weight, Lexicographical ordering is exploited to prioritize them in order to reduce random selection.

Algorithm 1 Main process of AGC.

Input: test suite $T = \{t_1, t_2, ...\}$
Calculated edge set$E = \{e_1, e_2, ...\}$
Template Stack S for the same weight test cases
while $T.size() != 0$ **do**
  Calculate weight for each test case t, $w = 0$
  while Any test case t haven't been calculated **do**
    while Any edge e haven't been calculated **do**
      if $!E.contains(e)$ **then**
        if The edge e is called from test to test code **then**
          $w+ = 0$
        else if The edge e is called from test to source code **then**
          $w+ = 1$
        else if The edge e is called from source to source code **then**
          $w+ = 2$
        end if
      end if
    end while
    Add t and w to candidate set $C = \{< t_1, w_1 >, < t_2, w_2 > , ...\}$
    Finding S which contain the largest same weight w_i
    if $w_i == w_{i-1}$ **then**
      $S.pop(t_i)$
    else if $w_i > w_{i-1}$ **then**
      $S.clear(), S.pop(t_i)$
    end if
  end while
  if $S.size() != 0$ **then**
    Apply *Lexicographical Ordering* and put these test cases into T'
    Delete these t_i from T, Put these edges in t_i into E
  else if $S.size() == 0$ **then**
    Find the largest w in C, put $t_i$into T'
    Delete t from T, Put edges in t into E
  end if
end while
Output: Prioritization order T'

4.3. Randomprocess reduction

Random selection behaviors may affect the stability of strategies in the process of prioritization. Table 2 shows an example about the main random process in the additional greedy strategy. There are eight functions $f_1 \dots f_8$ and four test cases $t_1 \dots t_4$, the same as Table 1. If the additional greedy strategy is applied to Table 2, there is no doubt that t_1 will be selected as the first test case. Then, t_2 is selected since it covers two function f_3 and f_4 which are not yet covered by t_1. However, the traditional additional greedy strategy cannot decide whether select t_3 or t_4 as the third test case, because both contain one function that has not

Table 2
Lexicographical example.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
t_1	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
t_2	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
t_3	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**
t_4	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**

been covered. A random selection may affect the fault detection efficiency especially when there are lots of such choices.

In order to reduce the random process, we introduce Lexicographical Ordering proposed by Eghbali and Tahvildari (2016). It augments 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. In Table 2, f_5 is covered twice by t_1 and t_2, f_8 is covered once by t_1, so t_4 gets a higher priority and is selected as the next test case.

Of course, in Table 2, if t_4 covers f_1 and f_5, not f_8, this algorithm cannot decide which test case to be selected. Generally speaking, lexicographical ordering can reduce but not eliminate randomness in the additional greedy strategy.

We choose the basic algorithm and related definition is given as follows:

Definition 3. When executing n test cases in n steps by any order. In each step, it selects one of the test cases in the order. Let denote $s \in \{1, 2....n\}$ as step s, then $cc^s \in \mathbb{N}^n$ is the total entity coverage of selected test cases until step s forms a vector, called *Cumulative Coverage vector*, $\mathbb{N}$ is the set of non-negative integers. We also denote *Ordered Cumulative Coverage vector* as occ^s, in which the smallest element of cc^s is the leftmost element of occ^s.

For example, in Table 2, if test cases t_1 and t_2 have been chosen, the vector cc^2 is [0, 0, 1, 1, 2, 1, 1, 1], the vector occ^2 is [0, 0, 1, 1, 1, 1, 1, 2]. Selecting t_3 as the next test case, the occ_3 is [0, 1, 1, 1, 1, 1, 1, 3], while selecting t_4 results in $occ^3 = [0, 1, 1, 1, 1, 1, 2, 2]$. Due to the reason that the latter has a higher lexicographical rank, t_4 is selected as the next test case.

The random process reduction algorithm is given as follows:

In Algorithm 2, lines 2 to 13 are the main loop of the algorithm. Stack S storages candidate test cases that cover the same number of covered entities. Each circulation will select one containing the highest lexicographical rank to the output order T' and pop it from S. It is worth noting that sometimes the cc_s of two or more test cases will be completely the same. In this case, we choose the last one to avoid the endless loop.

In general, our AGC technique utilizes two times of additional greedy strategy in both the global and the partial prioritization, which can reduce random selection and enhance the performance.

Algorithm 2 Main process of random reduction.

Input: Coverage matrix $\mathbf{C}= [c(i, j)]_{m\times n}$
Input: stack S which contains the largest same weight w_i
1: Initialize $cc = [0, ...0]_n$, $cc.sort() = occ$
2: **while** $S.size() \neq 0$ **do**
3: $candidate = null$
4: **for** Test cases $\{t_1, ...t_i\}$ in S **do**
5: $tmp = occ^i$
6: **if** $candidate < tmp$ **then**
7: $candidate = tmp$
8: $tbest = t_i, cbest = cc^i$
9: **end if**
10: **end for**
11: $S.pop(tbest)$
12: $T'.add(tbest)$
13: **end while**
Output: Random process reduction order T'

5. Empirical study & experiment

In this section, an empirical study is conducted to answer the following four research questions. All the experiments are carried out on a Lenovo PC with Intel Core i7-4790 3.60 GHz processor and 16GB RAM. Firstly we propose four research questions as follows.

1. **RQ1:** Comparing with other TCP strategies in different coverage criteria, is AGC superior enough?
2. **RQ2:** Comparing with different strategies in the same coverage criterion, is AGC superior enough?
3. **RQ3:** How is the time cost and efficiency of AGC?
4. **RQ4:** How about the performance of AGC in the real faulty environment?

RQ1 aims to measure the performance of relation-based technique against other TCP strategies. In order to answer RQ1, we compared the AGC with other 22 prioritization strategies in different coverage criteria. We choose the APFD value to measure the fault detection capability of each technique. It is widely utilized in TCP domain.

RQ2 is built for verifying whether AGC strategy performs well compared with other strategies in the same coverage criterion. We compare AGC with GA, TC, GOS and ART algorithm in the same method call sequence coverage criterion. We will also discuss the APFD value of each technique and try to verify the efficiency of AGC.

RQ3 aims to verify whether AGC achieves a good trade-off or even surpasses the strategies based on function or statement or branch criterion. We will measure the fault detection capability of each strategy and the execution time of each criterion in different strategies to answer RQ3.

RQ4 is designed to prove the availability of AGC in a real faulty dataset, due to the reason that some researchers (Just et al., 2014b; Andrews et al., 2005) doubted if mutant test cases are effective in simulating real-world regression testing environment. We collect the detected fault number and the number of executed test cases to make the curve in Fig. 8. The lower the curve is, the higher the fault detection capability it achieves. Experiments on a real faulty dataset called *Defects4J* are convincing enough to answer RQ4.

5.1. Implementation and Subject Programs

Kieker Van Hoorn et al. (2012) is utilized in our implementation as the experimental framework. *Kieker* 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. Then we realized that *Kieker* does not support statement-level and branch-level coverage collection. In order to collect statement-level and branch-level information, we chose *Jacoco* (Hoffmann et al., 2016), it does not support function call sequence collection. In order to be consistent with statement coverage, function coverage is also collected by *Jacoco* Both of these two instrumental tools are based on Java bytecode instrumentation. We think they can obtain similar results. In order to measure the fault detection rate, we inject faults into our subject programs by using Java mutation tool *MuJava* (Ma et al., 2005). As concluded in previous work (Just et al., 2014b; Andrews et al., 2005), mutation faults are close to real faults and are suitable for software testing experiment. Our experiment is based on class-level test cases.

Table 3
Basic information of programs (Ordered by LOC).

Subject Programs	version	LOC	Methods	Edges	TCNum	Mutant num
Commons.lang	3.5	26578	2132	4292	137	37466
Jodatime	2.1	27213	3591	11412	154	38378
Log4j-core	2.10.0	51769	6711	16140	362	7731
Commons.math	2.2	56039	3984	9584	264	192791
Jfreechart	1.0.19	98335	6897	16086	359	37271
Ant	1.9.7	108132	8123	21756	233	70320
Commons.math3	3.6.1	105191	7265	20251	510	339774
Google Closure Compiler	v20160713	140237	10653	49182	306	19935

Table 4
Method-level mutation operation.

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement
SDL	Statement Deletion
ASR	Variable Deletion
CDL	Constant Deletion
ODL	Operator Deletion

We choose eight open source Java programs[1][2][3][4][5][6][7] that have been broadly used in previous studies (Luo et al., 2016; Just et al., 2014b) from 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 unsuitable environment, so we remove these test cases. It is worth noting that version 3.0 of *Commons.math* conducts code refactoring and we regard version 3.x of *Commons.math* as a new program. Each program applies auto testing framework *Junit*.

Table 3 lists the nine subject programs including their names (Column 1), versions (Column 2), lines of code (LOC, Column 3) and the number of methods (Column 4). The number of edges in the aggregated graphs is given in Column 5. Columns 6 provides the number of test cases at class level. The last column shows the number of mutations generated by *MuJava*.

5.2. Design of the empirical study

The only information we exploit in our empirical study are the execution results obtained by *Kieker* and *Jacoco*. 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.

Algorithm 3 gives the pseudo-code on how to compare the effectiveness of different 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. Then, mutation faults which are generated by *MuJava* are injected into the program in order to simulate the faulty version of the program. The mutation operators contain in *MuJava* are shown in Table 4. They can be classified as follows:

1. Arithmetic Operators: (1) +, (2) −, (3) *, (4) /, and (5) %;
2. Relational Operators: (1) > , (2) >=, (3) < , (4) <=, (5) ==, and (6) ! =;
3. Conditional Operators: (1) &&, (2) ||, (3) &, (4) |, and (5) ∧;
4. Shift Operators: (1) > > , (2) < < , and (3) > > > > ;
5. Logical Operators: (1) &, (2) |, (3) ∧ and (4) ~
6. Assignment Operators: (1) + =, (2) − =, (3) ∗ =, (4) / =, (5) % =, (6) & =, (7) | =, (8) ∧ =, (9) <<=, (10) >>=, and (11) >>>=
7. Deletion Operators: delete statements, variables, constants, and objects.

We randomly choose 5 different mutations into one faulty version (e.g., a mutant group) and totally produce 1000 versions for each subject program based on the work of Luo et al. (2016). That is, we generate 5000 mutants in each of these programs. None of these 5000 mutants is repetitive. The real number of mutants adopted in the experiment is 5000 except for *Log4j*, we will ensure that each faulty version contains 5 mutants that can be killed by at least one test case. As for *Log4j*, the total mutant number is 7731 but we cannot find sufficient mutants that can be detected by test cases. The real number of mutants adopted in *Log4j* is 3210.

Algorithm 3 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
Create 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
```

[1] http://commons.apache.org/proper/commons-lang/index.html.
[2] http://www.joda.org/joda-time/.
[3] https://logging.apache.org/log4j/2.x/.
[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 5
Comparison of average APFD Values (%) for Different TCP techniques.

Subject Programs	AGC	AFC	ASC	ABC	GC	FCS	GA_fc	GA_cs	GA_st	GA_br	TFC	TCC	TSC	TBC
Commons.lang	67.55	65.32	69.22	66.74	60.10	58.29	66.56	67.02	69.73	66.67	60.47	61.29	58.11	57.99
Jodatime	83.65	80.98	83.69	80.98	72.34	68.30	80.12	82.04	83.33	82.03	80.77	81.34	79.99	78.59
Log4j-core	84.54	76.32	87.44	87.41	69.31	66.71	72.11	74.56	82.22	73.94	69.18	69.74	69.85	70.04
Commons.math	74.75	66.98	66.18	63.42	73.19	56.68	64.78	72.34	66.86	63.15	60.75	62.30	54.24	53.18
Jfreechart	82.84	78.45	78.25	81.90	77.78	78.39	80.94	82.91	78.21	81.11	77.35	79.59	77.65	79.23
Ant	82.59	77.78	79.17	78.81	75.13	52.86	77.54	81.79	78.53	79.22	75.28	76.16	75.53	75.20
Commons.math3	73.15	67.03	71.45	67.43	67.76	53.44	66.36	71.16	66.16	71.57	68.32	60.84	54.45	52.85
Google Closure Compiler	91.13	84.12	89.44	85.42	88.86	65.40	85.13	88.31	87.71	86.10	75.15	76.51	75.84	74.49
Average	80.81	74.62	79.63	77.00	73.07	62.33	74.19	77.52	76.59	75.47	70.91	70.97	68.21	68.19

Subject Programs	AGC(repeated)	GOS_fc	GOS_cs	GOS_st	GOS_br	ART_fc	ART_cs	ART_st	ART_br	NO
Commons.lang	67.55	66.17	66.90	69.65	66.21	59.26	62.54	61.42	61.33	50.04
Jodatime	83.65	79.44	80.27	82.11	79.97	68.54	75.19	78.44	74.71	79.16
Log4j	84.54	73.13	80.18	82.22	81.19	75.21	78.92	80.10	79.64	64.41
Commons.math	74.75	66.33	71.53	63.38	62.18	67.09	69.47	66.54	64.17	60.81
Jfreechart	82.84	82.31	85.11	84.63	82.85	85.33	86.98	85.42	84.35	56.52
Ant	82.59	77.37	81.75	77.39	77.86	55.31	63.44	62.63	61.26	73.08
Commons.math3	73.15	67.11	70.79	64.60	62.91	59.66	66.15	65.84	62.96	39.58
Google Closure Compiler	91.13	84.36	87.03	87.19	84.59	89.17	89.67	87.16	89.05	62.64
Average	80.81	7453	78.63	77.92	72.90	69.95	74.97	67.76	73.22	61.07

It is noted that we know what is the mutation operation and where will the mutant be injected to in advance. When *Junit* reports one fault after executing test cases, it is easy for us to locate the mutant based on the testing log of *Junit*. We have filtered these mutations and tried our best to make sure that each faulty version has 5 independent mutants. In other words, if there is a new fault detected by the test case, it is only caused by one mutant.

In fact, large numbers of mutants cannot be detected by executing test cases. One mutant per version is really an expensive way to simulate the real fault version. In order to validate that whether our faulty version can successfully simulate the real faulty environment, we ran AGC and other TCP techniques on real faulty dataset named *Defects4J*. We will introduce it in the following.

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 Eq. 1. It is widely utilized in TCP domain (Rothermel et al., 2001; 1999; Elbaum et al., 2000; 2002; 2003).

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

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. Recall that we created 1000 faulty versions for each subject program and ran all TCP techniques over these 1000 faulty version. In other words, we executed all test cases and ran each technique 1000 times for each subject. In addition, *Jacoco* indeed does not support per-test coverage and only gives the total coverage report. In order to solve this problem, each time we only run one test case and collect the execution report. If one test case finishes executing, the next test case starts executing. Since we know the location of these mutations in advance, we can measure which mutation causes the fault based on the testing log of *Junit*. In order to precisely measure the fault, we use one single thread to execute test cases one by one, which is time consuming. After running all test cases, we calculate 1000 APFD values and utilize the mean value for evaluation.

5.3. Performance

In this subsection, we present experimental data to answer RQ1, RQ2 and justify the effectiveness of AGC in regression testing prioritization. Fig. 7 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;
- AFC: Additional Greedy Function-coverage Technique;
- ASC: Additional Greedy Statement-coverage Technique;
- ABC: Additional Greedy Branch-coverage Technique;
- GC: Graph Similarity Clustering Technique;
- FCS: Function Call Sequence;
- GA_fc: Search-based Technique (Genetic-based Algorithm in function-level);
- GA_cs: Search-based Technique (Genetic-based Algorithm in call-sequence-level);
- GA_st: Search-based Technique (Genetic-based Algorithm in statement-level);
- GA_br: Search-based Technique (Genetic-based Algorithm in branch-level);
- TFC: Total Greedy Function-coverage Technique;
- TCC: Total Greedy Method Call Sequence Technique;
- TSC: Total Greedy Statement-coverage Technique;
- TBC: Total Greedy Branch-coverage Technique;
- GOS_fc: Greed-aided-clustering Ordered Sequence (Function-level);
- GOS_cs: Greed-aided-clustering Ordered Sequence (Call-Sequence-level);
- GOS_st: Greed-aided-clustering Ordered Sequence (Statement-level);
- GOS_br: Greed-aided-clustering Ordered Sequence (Branch-level);
- ART_fc: Adaptive Random Technique (Function-level);
- ART_cs: Adaptive Random Technique (Call-Sequence-level);
- ART_st: Adaptive Random Technique (Statement-level);
- ART_br: Adaptive Random Technique (Branch-level);
- NO: Natural Order (Represent the result without any prioritization, which is tested in alphabetical order).

The values of APFD for all subject programs are given in Fig. 7 and Table 5. Each sub-figure in Fig. 7 presents a subject program. It has detailed APFD values under different TCP techniques. In addition, each boxplot represents the APFD distribution from 5% to 95%. In Table 5, the TCP technique that has the best performance is marked in **red** color, the next best is marked in **blue** color, the third best is marked in **yellow** color and the fourth best is marked in **green** color.

Table 6
Mann–Whitney tests between AGC and other techniques.

	AFC	ASC	ABC	GC	FCS	GA_fc	GA_cs	GA_st	GA_br	TFC	TCC	TSC	TBC
Mann-Whitney U	43110	43180	53726	43031	36499	53087	57891	53784	51781	36417	45325	31458	38413
Wilcoxon W	102531	102865	113411	102716	96174	113564	117894	113429	112784	92314	102482	96357	96341
Z	-6.282	-6.239	-2.210	-6.296	-8.790	-2.754	-1.944	-6.357	-3.495	-6.927	-2.887	-8.152	-7.145
η^2	0.621	0.626	0.597	0.626	0.644	0.599	0.586	0.597	0.602	0.641	0.620	0.658	0.639
p	.000	.000	.027	.000	.000	.000	.039	.011	.016	.007	.010	.000	.004
	GOS_fc	GOS_cs	GOS_st	GOS_br	ART_fc	ART_cs	ART_st	ART_br					
Mann-Whitney U	51157	58904	57372	53268	49361	54293	53449	57181					
Wilcoxon W	111246	118589	117057	112953	109433	113978	113134	116866					
Z	-3.111	-0.232	-6.312	-2.385	-3.831	-1.994	-2.316	-2.119					
η^2	0.601	0.583	0.587	0.598	0.609	0.596	0.598	0.588					
p	.002	.816	.000	.017	.000	.046	.021	.037					

Based on these boxplots, we can make the following observations:

Finding 1: An interesting phenomenon is found as that with the size of programs growing, the performance of AGC also exhibits a continuous growth trend compared with traditional TCP techniques. In the small subject programs *commons.lang, jodatime* and *log4j-core* with LOC ranging from 26578 to 51769, the performance of AGC is worse than that of ASC. For example, in the program *Jodatime*, the median APFD value of the ASC technique is 0.8365 followed by AGC, GA_st, GOS_st, GA_cs, GA_br, AFC, TCC, TFC, ABC, GA_fc, GOS_cs, TSC, GOS_br, GOS_fc, NO, TBC, ART_st, ART_cs, ART_br, GC, ART_fc and FCS. A half number of TCP strategies are even worse than NO (Table 5).

However, in large subject programs such as *Math, Jfreechart, Ant, Math3, Google Closure Compiler* with LOC ranging from 56039 to 140237, the performance of AGC is superior and even better than ASC. For example, in the program *Google Closure Compiler*, the median APFD value of AGC achieves the best performance 0.9113, followed by ART_cs, ART_fc, ASC, ART_br, GC, GA_cs, GA_st, GOS_st, ART_st, GA_br, GOS_cs, GA_fc, ABC, AFC, GOS_fc, TCC, TSC. TFC, TBC, FCS and NO. In the program *Jfreechart*, the median APFD value of AGC is 0.8284, better than 0.7845 of AFC and 0.7825 of ASC, but a little worse than GA, ART and GOS algorithm. However, our experiments show that ART are not stable enough. In the program *Ant*, performances of all the ART strategies are even worse the alphabetical order NO. GA is a competitive strategy bu we also achieve a better performance in the average fault detection rate. GOS will be discussed in Finding 2.

As a conclusion, no matter comparing with different granularity techniques ASC and AFC or comparing with GA, TFC, GOS and ART based on the same coverage criterion, our AGC technique achieves the best average fault detection capability. It shows a clear growth trend when the size of the program becomes larger and larger. **Performance results can answer RQ1 and RQ2 that AGC is superior to improve the performance of test case prioritization in regression testing.**

5.4. Significance analysis

In order to investigate whether there are significant differences among the 22 test case prioritization techniques, we performed a one-way ANOVA[8] on the APFD values gathered from each program with mutation faults. The ANOVA test reveals whether there is a significant difference between all studied techniques based on the APFD value. Statistical analysis results for the eight object programs are shown in Table 7, where *SS* denotes Sum of Squares, *DF* denotes Degrees of Freedom, *MS* denotes Mean Square, *F* denotes the statistical F-value, *p* denotes the calculated p-value, and η^2 denotes the Partial Eta Squared. η^2 will help us quantify how strong effects are and tell us the correlation between two variables. When the p-value is less than 0.001, it will be represented by 0.000.

As the result shows in 7, the p-value in all programs are 0.000, smaller than the significance level of 0.05. It is clear that there are significant differences among the 22 test case prioritization techniques at different granularities on each program. η^2 in one-way ANOVA test ranges from 0.14 in *Jodatime* to 0.511 in *Google Closure Compiler*. It means that different methods are strongly related to results in our experimental project. There are significant differences between different methods.

Moreover, in order to illustrate the relationship between individual subject programs, we performed the *Wilcoxon-Mann-Whitney* test between AGC and other techniques in Table 6. It can be used to determine whether two independent samples were selected from populations having the same distribution. *Mann – Whitney U* and *Wilcoxon W* are our test statistics. They summarize the difference in mean rank numbers in a single number. Z denotes z-score, p denotes p-value, and η^2 denotes the Partial Eta Squared.

Finding 2: There is a statistically significant difference between techniques across subjects (e.g., $p < 0.05$, $\eta^2 > 0.5$), only GOS_cs has a relationship with our method. We have checked the reason and found that in some cases the cluster process of GOS algorithm does not work, even reduce the fault detection capability. We have carefully read the paper and find that in the clustering process, GOS algorithm merge the two clusters with minimum distance in each loop. However, there is a risk that if we always merge two clusters with minimum distance, it will gather the most of entities into one large cluster so that the clustering process is meaningless. Even it will decrease the fault detection capability. We have found this case in nearly half of our subject programs. Therefore, in most cases GOS_cs is an inferior AGC algorithm, that is the reason why it is significantly related to AGC.

5.5. Time usage

Table 8 shows the prioritization cost of each TCP technique. The prioritization cost increases significantly when choosing finer granularity. However, there is no big gap between TCP techniques in the same granularity. We have tried our best to use the similar data structure in order to avoid the noise and the interrupt of the raw data. It is valuable to mention that there are two parts of GOS algorithm. The first part is merge clusters based on edit distance to k clusters and the second part is apply additional greedy algorithm in each cluster. In the first part, we utilize the edit distance results of FCS. Therefore, the actual time costs of GOS algorithm is higher than that are listed in the Table. The prioritization cost of

[8] The statistical analyses described in this paper are performed using SPSS 19.0, analytical software accessible at http://www.spss.com.

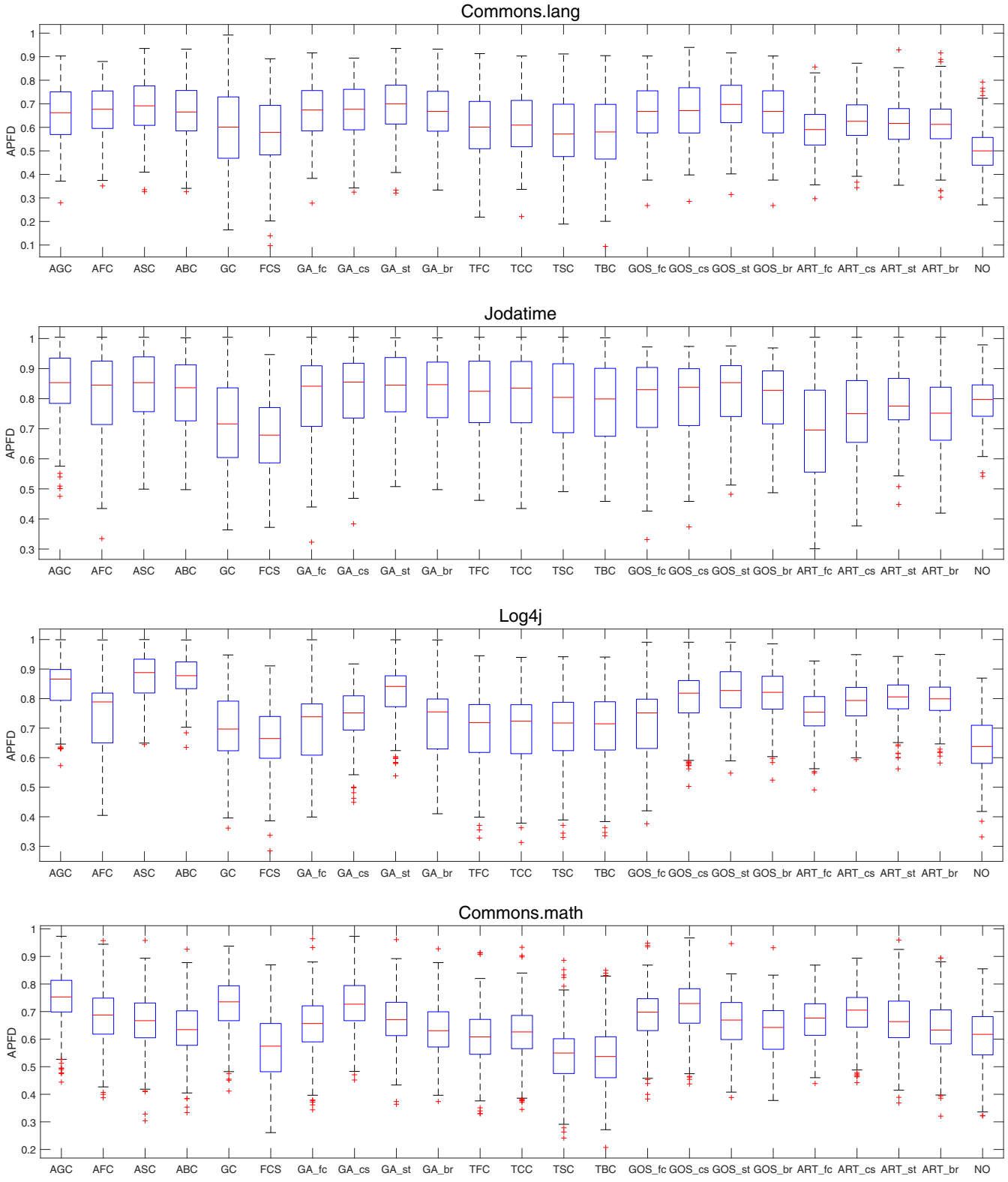

Fig. 7. Result for our techniques and traditional TCP techniques on 8 open source programs. The box and whisker plots represent the values of APFD metric for different TCP techniques. The x-axis represents the different techniques and the y-axis represents the APFD values. The central box of each plot represents the values from 25 to 75 percentage.

AGC is between AFC and ASC, nearly the same as ABC and more efficient than other similarity-based techniques except for ART. However, we achieve better fault detection performance than it. Comparing with other techniques, total greedy strategy takes the least time, ART takes the most time. The time cost of GA and additional greedy is in the same level.

Additional greedy algorithm needs to repetitively search all candidates to find the next test case which contains the highest number of unique entities. In other words, it always searches for the

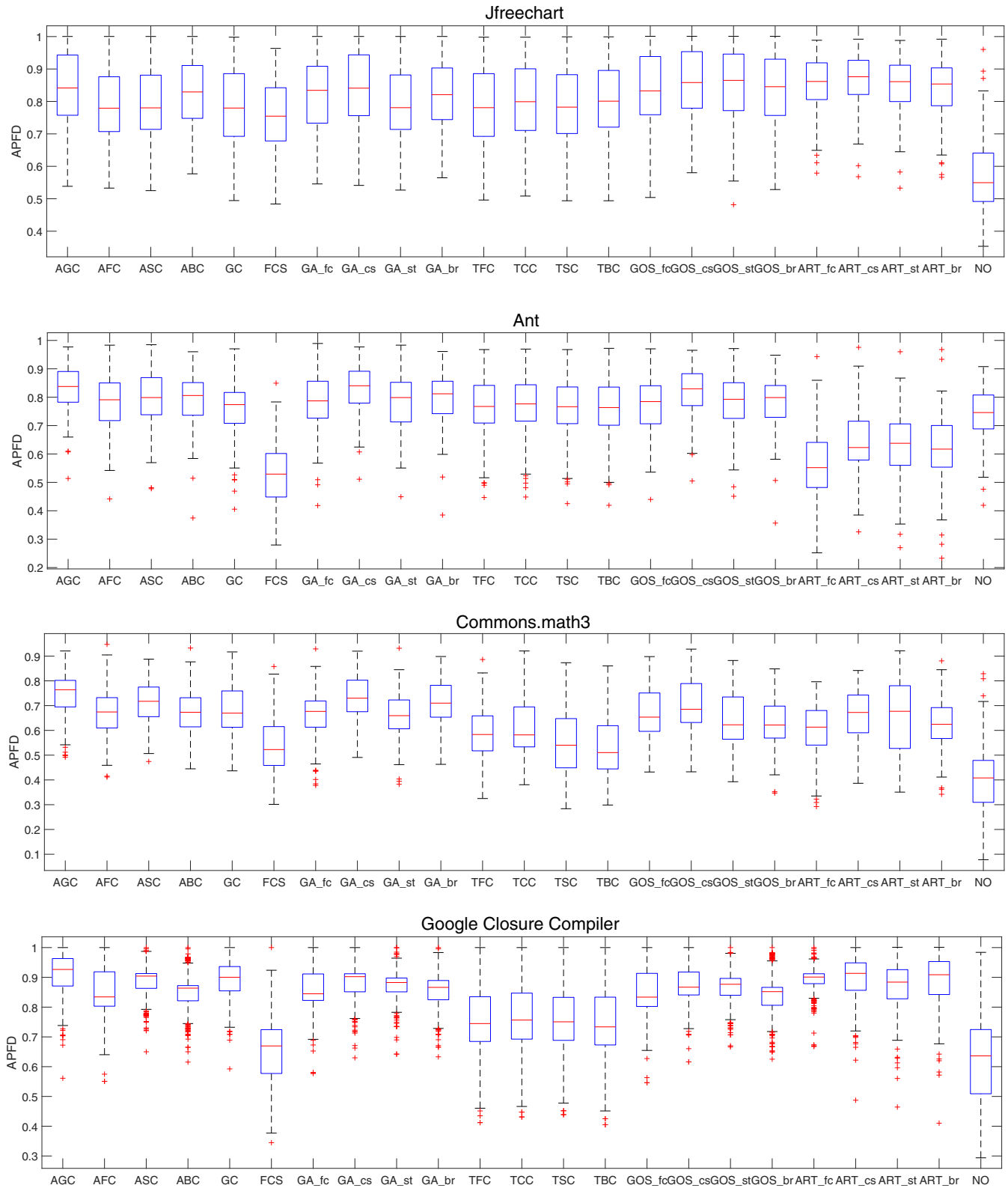

Fig. 7. Continued

test case that covers the most units that have not been covered. That is the main prioritization cost. Lexicographical ordering will increase 10%-20% time cost. However, it indeed will improve the performance. The implementation of the additional greedy algorithm still has space to be optimized.

As a conclusion, the cost of AGC is one-third to one-eighth as long as ASC but achieves a better mean APFD value. Both the performance and the cost of AFC is less than AGC. We can answer RQ3 that it is a good trade-off between time cost and fault detection capability. If developers want to obtain a sweet spot between fault detection capability and prioritization cost, we believe our AGC technique is competitive with other TCP techniques, especially in large programs.

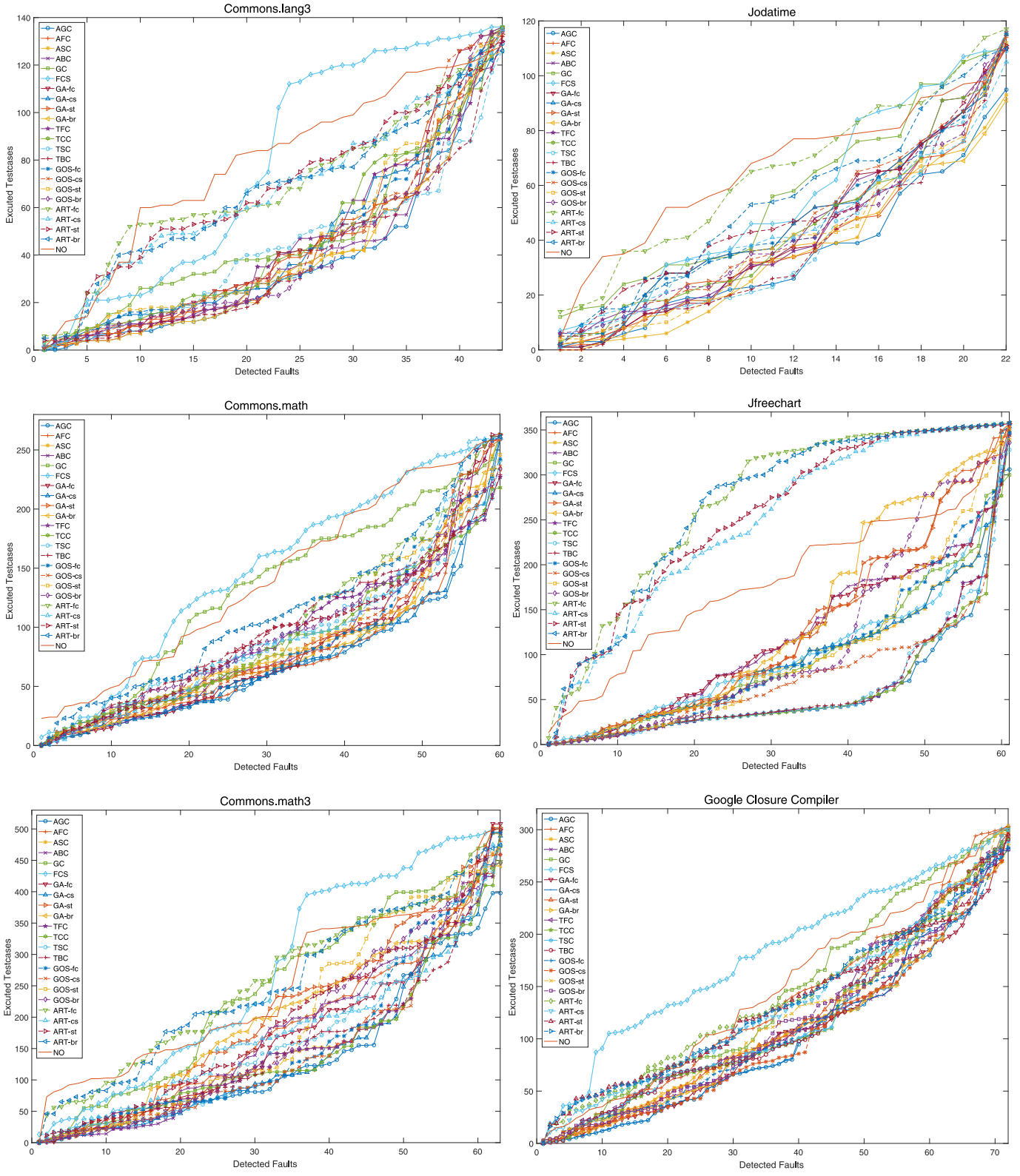

Fig. 8. Different performances on Defect4J.

5.6. Real world experiment

Experiments in the previous subsection utilize mutant test cases that contain faults to simulate faulty versions of programs in the real world. In order to answer RQ2, this subsection we will implement our approach and other comparative prioritization strategies on real faulty versions which are collected by the dataset named *Defects4J* (Just et al., 2014a).

Defects4J is a collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research. Defects4J contains 395 bugs from the following open-source projects in Table 9:

Table 7
One-way ANOVA results.

Commons.lang	SS	DF	MS	F	p	η^2
Between Groups	6.348	5	.058	71.372	0.000	0.183
Within Groups	28.430	1918	0.015			
Total	34.777	1924				
Jodatime	SS	DF	MS	F	p	η^2
Between Groups	2.362	5	0.186	11.643	0.000	0.141
Within Groups	15.677	980	0.016			
Total	16.795	986				
Log4j	SS	DF	MS	F	p	η^2
Between Groups	15.496	5	3.099	368.408	0.000	0.434
Within Groups	20.190	2400	0.008			
Total	35.686	2405				
Commons.math	SS	DF	MS	F	p	η^2
Between Groups	10.435	5	1.191	135.606	0.000	0.401
Within Groups	26.003	2961	0.009			
Total	33.149	2967				
Jfreechart	SS	DF	MS	F	p	η^2
Between Groups	19.352	5	3.225	298.732	0.000	0.427
Within Groups	25.998	2408	0.011			
Total	45.350	2414				
Ant	SS	DF	MS	F	p	η^2
Between Groups	5.196	5	0.866	86.490	0.000	0.400
Within Groups	7.780	777	0.010			
Total	12.976	783				
Commons.math3	SS	DF	MS	F	p	η^2
Between Groups	32.318	5	5.386	541.940	0.000	0.509
Within Groups	31.168	3136	0.010			
Total	63.486	3142				
Google Closure Compiler	SS	DF	MS	F	p	η^2
Between Groups	28.208	5	4.701	533.007	0.000	0.511
Within Groups	27.043	3066	0.009			
Total	55.251	3072				

Each bug in Table 9 has the following properties:

- Issue filed in the corresponding issue tracker, and issue tracker identifier mentioned in the fixing commit message;
- The Defects4J maintainers manually pruned out irrelevant changes and the bug is fixed in a single commit (e.g., refactorings or feature additions);
- The test failure is not random or dependent on test execution order, a triggering test that failed before the fix and passes after the fix exists.

Some bugs can be triggered by the same test case in the different commit versions and others can be triggered by more than one test cases. Therefore, we collect and count test cases triggered by all the bugs in the last row of Table 9. It is necessary to explain *Lang* and *Math*. The version 3.0 of *Commons.lang* and *Commons.math* are refactoring versions, we singly collect referred test cases of *Commons.lang* higher than version 3.0 and divide the program of *Commons.math* into two programs *Commons.math* and *Commons.math3*.

Fig. 8 shows the result of different prioritization strategies on Defect4J. The x-axis represents the fault number that detected by executed test cases. The y-axis shows the number of executed test cases. We use different sighs to represent the AGC, AFC, ASC, ABC, GC, FCS, GA_fc, GA_cs, GA_st, GA_br, TFC, TCC, TSC, TBC, GOS_fc, GOS_cs, GOS_st. GOS_br, ART_fc, ART_cs, ART_st, ART_br and natural alphabetic order respectively, the blue curve with circular mark represents the AGC strategy. Generally speaking, the closer the curve is to the x-axis, the better performance the technique gets. From the figure, we have the following observations:

First, our technique AGC (blue curve with circular mark) gets a good performance in real world programs. For example, AGC performs very well in *Commons.math3, Jfreechart* and *Google Closure Compiler*, always the best in the overall test process and sometimes performs the best in *Jodatime, Commons.lang* and *Commons.math*.

Second, results on real faulty versions of Defects4J is similar with the results on mutant faulty versions in Figure 5.3. Our technique AGC have a clear growth trend when the size of programs becomes bigger and bigger. For example, performances in three bigger programs *Commons.math3, Jfreechart* and *Google Closure Compiler* are obviously better than three smaller programs. We think the reason to be that larger lines of code may bring more complex structure and rapidly method call sequence increase than function increase. This is beneficial for our AGC strategy.

In conclusion, just like what RQ4 said, AGC is effective not only in the mutant environment but also in real faulty cases.

5.7. Evaluation

Experiments above show 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.

Fig. 9 shows the entity growth in nine programs. The x-axis represents the number of test cases that have been executed. The y-axis represents the percentage of entity coverage (Function, Call

Table 8
Prioritization Cost (Seconds) for different techniques.

Subject Programs	AGC	AFC	ASC	ABC	GC	FCS	GA_fc	GA_cs	GA_st	GA_br	TFC	TCC	TSC	TBC
Commons.lang	2	13	65	12	178	108	59	60	95	47	1	1	2	1
Jodatime	6	202	1064	119	7830	3007	120	242	336	156	3	5	10	5
Log4j	66	221	901	240	4760	2336	350	437	713	314	20	26	33	24
Commons.math	8	139	322	74	591	295	128	263	581	170	4	5	7	4
Jfreechart	16	193	468	75	6355	1926	196	222	448	222	4	7	9	4
Ant	37	244	869	198	2320	840	264	433	891	452	3	6	7	4
Commons.math3	47	568	1043	508	4817	1416	273	626	922	434	11	13	17	12
Google Closure Compiler	390	3076	9867	4652	24853	5648	1029	3394	6751	1800	68	95	144	89

Subject Programs	AGC(repeated)	GOS_fc	GOS_cs	GOS_st	GOS_br	ART_fc	ART_cs	ART_st	ART_br
Commons.lang	13	4	10	40	16	2	4	11	4
Jodatime	202	12	74	120	15	6	118	123	8
Log4j	221	51	263	894	196	60	106	1333	84
Commons.math	139	9	111	331	44	2	27	226	11
Jfreechart	193	5	25	85	12	7	12	297	18
Ant	244	41	156	817	26	30	177	1877	194
Commons.math3	568	11	76	506	50	10	122	910	50
Google Closure Compiler	3076	463	2548	10335	3586	412	11371	18309	4642

Table 9
Bugs contained in Defect4J.

Identifier	Project name	Number of bugs
Time	Jodatime	22
Lang	Apache.commons.lang	45
Math	Apache.commons.math	60
Math3	Apache.commons.math3	62
Chart	Jfreechart	61
Closure	Google Closure compiler	72

Sequence, Statement, Branch) by executed test cases. We utilize three prioritization techniques in different granularities. The dotted line represents the execution result of Statement in ASC strategy. The full line represents the execution result of Call Sequence in AGC strategy. The imaginary line denotes the execution result of Function in AFC strategy. The dash-dot line denotes the execution result of Branch in ABC strategy. Other techniques are not considered in order to control variables.

We can find that in big size programs, the curve of the call sequence is detached obviously with the curve of the function. In other words, test cases in 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 the complex structure and more call sequences, but not the same growth rate of functions, statements and branches. That is the reason we think our technique shows up a growth trend of performance with the size of the program increases.

6. Threats to validity

To justify the effectiveness of our relation-based TCP techniques we implemented some TCP techniques presented in prior works for comparison. However, traditional TCP techniques we implemented are based on the guidelines in original works. We are afraid of misunderstanding the authors' original intention. For example, when we tried to implement the ART technique, we found that the first test case that selected from the candidate set has a great influence on the APFD metric. In the original work, the author randomly chose a test case as the first but in this case, the APFD value was unstable and not matching our requirement. Thus, in this study, we locked the first candidate test case and chose the one that covers the biggest number of functions as the first test case.

In order to collect the raw data at function-level and statement-level, we use different monitoring and testing tools *Kieker* and *Jacoco*. It is not guaranteed that these two tools' data obtainment is complete and credible. Thus, the missing data may impact the effective comparison between different granularities.

Another issue is that *JaCoCo* reports branch coverage in line level granularity, i.e. for every line the number of covered branches vs. the number of missed branches is reported. Due to the limitation of *JaCoCo*, we only know how many branches are executed or missed, but it doesn't tell us which branch is missed. In this case, if two test cases execute a if/switch block and both of them miss 1 of 4 branches. We can judge if these two test cases attain the same execution status. If not, they will be labeled as covering different branch elements. Another scenario is that, if one test case misses 1 of 4 branches and the other test case miss 3 of 4 branches, we are not sure if the second test case executes the missing branch of the first test case. However, we will choose the first test case because it covers more branch elements.

In fact, the only correlation we know is whether these two test cases cover the same branches. We cannot verify the inclusion relation between the two. This function is not as accurate as exactly knows each branch information. However, we have manually ana-

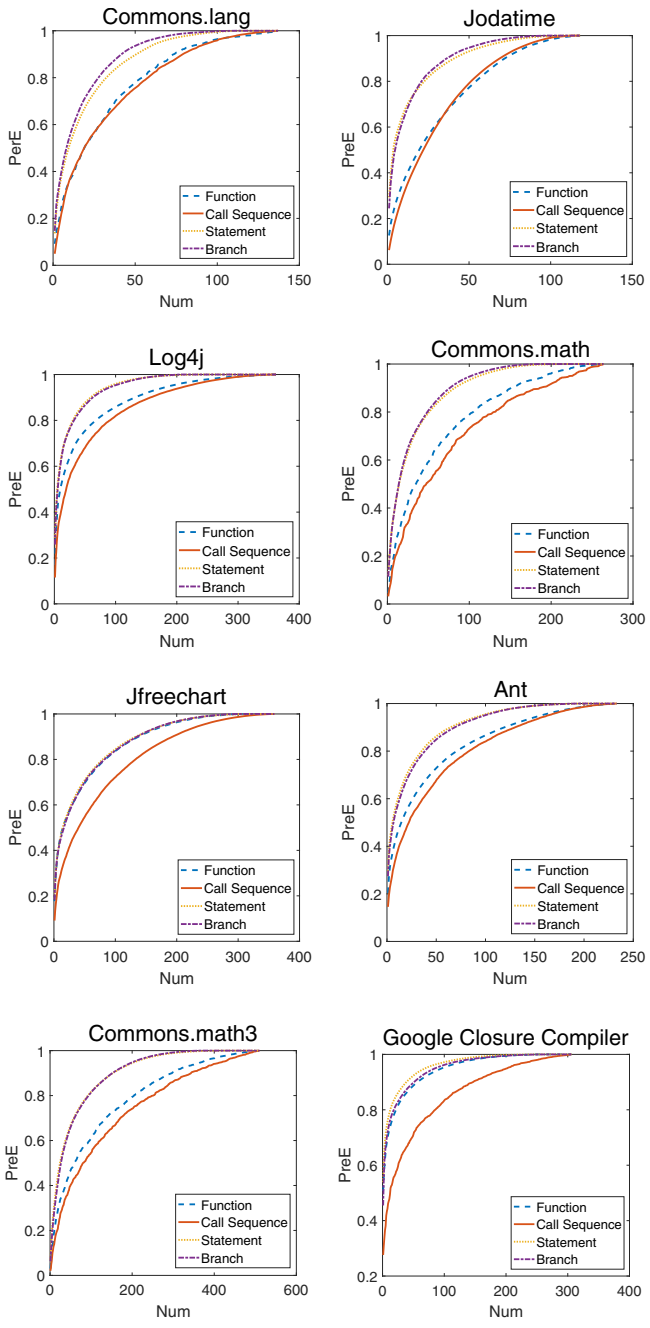

Fig. 9. Entity growth in different granularities.

lyzed some branches and prioritized them based on the concrete branch coverage information, results are nearly the same. It may depend on the quality of test cases.

In this research, we try our best to ensure the equity for each TCP technique, such as filter the raw data, using the same data structure and repeating more than once to calculate the mean value, which may reduce the risk of using different testing and monitoring tools.

The main external threat to our study is that the experiment is conducted on 8 Java software systems, which may impact the universality of results. However, the size of programs that we chose varies from 26K to 140K, most of these programs were used as benchmarks in previous papers (Luo et al., 2016; Just et al., 2014b). Thus, we believe our study has sufficiently mitigated this threat to a point that the conclusion can be drawn in the context of our research questions.

7. Future work

GOS technique inspires us that some heuristic cluster technique may be helpful to improve the performance and time cost of AGC. If we can introduce more efficient cluster algorithm to reduce the global addition greedy search cost. However, there are numbers of parameters that cluster-based techniques need to choose. Such as the k value of cluster number, which sample strategy should these clusters choose. These parameters will significantly affect the performance of cluster-based techniques. We have tried some technique such as k-means and k-medoids. The performance is not good and it is really hard to decide which number of clusters should we choose. Hierarchical clustering or even machine learning technique are good choices but it is the future work.

8. Conclusion

In this paper, a relation-based TCP technique AGC based on method call sequences is presented. Balance the prioritization efficiency and effectiveness is a big problem that traditional coverage-based TCP techniques need to be faced. The relation information that corresponds to method call sequences in our graph model performs well in TCP process. It gives us the confidence that it can offer a good trade-off between the two factors Based on the new criterion we have implemented the prioritization algorithm AGC.

Experiments are conducted between our AGC techniques and other traditional TCP techniques on eight open source programs and a real faulty dataset. AGC is particularly effective on large programs in our experimental results. As for the reason, these programs have complex structural information, bugs are hard to be detected by the traditional unit-based coverage criterion. Our AGC approach achieves a better average fault detection capability index than finer granularity statement-coverage techniques with one-third to one-eighth cost. It is proved both outstanding in the mutant and real faulty environment. Therefore, we believe relation-based technique AGC performs well as a trade-off between fault detection capability and prioritization cost.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Jianlei Chi: Visualization, Formal analysis, Writing - review & editing, Writing - original draft, Investigation, Software, Methodology, Conceptualization. **Yu Qu:** Validation, Visualization, Writing - review & editing, Supervision. **Qinghua Zheng:** Project administration, Resources, Writing - review & editing, Supervision, Funding acquisition. **Zijiang Yang:** Writing - review & editing. **Wuxia Jin:** Writing - review & editing. **Di Cui:** Writing - review & editing. **Ting Liu:** Supervision, Writing - review & editing.

Acknowlgedgments

This work was supported by National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61632015, 61772408, U1766215, 61721002, 61532015, 61833015), Ministry of Education Innovation Research Team (IRT_17R86), and Project of China Knowledge Centre for Engineering Science and Technology.

References

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th international conference on Software engineering. ACM, pp. 402–411.

Catal, C., Mishra, D., 2013. Test case prioritization: a systematic mapping study. Softw. Qual. J. 21 (3), 445–478.

Chittimalli, P.K., Harrold, M.J., 2009. Recomputing coverage information to assist regression testing. IEEE Trans. Softw. Eng. 35 (4), 452–469.

Dickinson, W., Leon, D., Podgurski, A., 2001. Pursuing failure: the distribution of program failures in a profile space. In: ACM SIGSOFT Software Engineering Notes, 26. ACM, pp. 246–255.

Do, H., Rothermel, G., Kinneer, A., 2004. Empirical studies of test case prioritization in a junit testing environment. In: Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on. IEEE, pp. 113–124.

Eghbali, S., Tahvildari, L., 2016. Test case prioritization using lexicographical ordering. IEEE Trans. Softw. Eng. 42 (12), 1178–1195.

Elbaum, S., Kallakuri, P., Malishevsky, A., Rothermel, G., Kanduri, S., 2003. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Softw. Test. Verif. Reliab. 13 (2), 65–83.

Elbaum, S., Malishevsky, A., Rothermel, G., 2001. Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, pp. 329–338.

Elbaum, S., Malishevsky, A.G., Rothermel, G., 2000. Prioritizing Test Cases for Regression Testing, 25. ACM.

Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. 28 (2), 159–182.

Fang, C., Chen, Z., Wu, K., Zhao, Z., 2014. Similarity-based test case prioritization using ordered sequences of program entities. Softw. Qual. J. 22 (2), 335–361.

Graham, S.L., Kessler, P.B., Mckusick, M.K., 1982. Gprof: a call graph execution profiler. In: ACM Sigplan Notices, 17. ACM, pp. 120–126.

Hao, D., Zhang, L., Mei, H., 2016. Test-case prioritization: achievements and challenges. Front. Comput. Sci. 10 (5), 769–777.

Hao, D., Zhang, L., Zhang, L., Rothermel, G., Mei, H., 2014. A unified test case prioritization approach. ACM Trans. Softw. Eng.Methodol. (TOSEM) 24 (2), 10.

Hemmati, H., Arcuri, A., Briand, L., 2011. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. IEEE, pp. 327–336.

Hemmati, H., Briand, L., 2010. An industrial investigation of similarity measures for model-based test case selection. In: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on. IEEE, pp. 141–150.

Hoffmann, M., Janiczak, B., Mandrikov, E., Friedenhagen, M., 2016. Jacoco code coverage tool. online, 2009.

Holland, J.H., 1992. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press.

Huang, Y.-C., Peng, K.-L., Huang, C.-Y., 2012. A history-based cost-cognizant test case prioritization technique in regression testing. J. Syst. Softw. 85 (3), 626–637.

Islam, M.M., Marchetto, A., Susi, A., Scanniello, G., 2012. A multi-objective technique to prioritize test cases based on latent semantic indexing. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, pp. 21–30.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.

Jiang, B., Zhang, Z., Chan, W.K., Tse, T., 2009. Adaptive random test case prioritization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 233–244.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, pp. 437–440.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 654–665.

Kaner, C., 1997. Improving the maintainability of automated test suites. Software QA 4 (4).

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. An overview of aspectj. In: European Conference on Object-Oriented Programming. Springer, pp. 327–354.

Kim, J.-M., Porter, A., 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th international conference on software engineering. ACM, pp. 119–129.

Kinable, J., Kostakis, O., 2011. Malware classification based on call graph clustering. J. Comput. Virol. 7 (4), 233–245.

Korel, B., Koutsogiannakis, G., Tahat, L.H., 2007. Model-based test prioritization heuristic methods and their evaluation. In: Proceedings of the 3rd international workshop on Advances in model-based testing. ACM, pp. 34–43.

Li, Z., Harman, M., Hierons, R.M., 2007. Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. 33 (4).

Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., Zhang, L., 2016. How does regression test prioritization perform in real-world software evolution? In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 535–546.

Luo, Q., Moran, K., Poshyvanyk, D., 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 559–570.

Ma, Y.-S., Offutt, J., Kwon, Y.R., 2005. Mujava: An automated class mutation system. Softw. Test. Verif. Reliab. 15 (2), 97–133.

Marijan, D., Gotlieb, A., Sen, S., 2013. Test case prioritization for continuous regression testing: an industrial case study. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE, pp. 540–543.

McMaster, S., Memon, A., 2008. Call-stack coverage for gui test suite reduction. IEEE Trans. Softw. Eng. 34 (1), 99–115.

Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., Rothermel, G., 2012. A static approach to prioritizing junit test cases. IEEE Trans. Softw. Eng. 38 (6), 1258–1275.

Mondal, D., Hemmati, H., Durocher, S., 2015. Exploring test suite diversification and code coverage in multi-objective test case selection. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on. IEEE, pp. 1–10.

Noor, T.B., Hemmati, H., 2015. A similarity-based approach for test case prioritization using historical failure data. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 58–68.

Qu, Y., Guan, X., Zheng, Q., Liu, T., Zhou, J., Li, J., 2015. Calling network: a new method for modeling software runtime behaviors. ACM SIGSOFT Softw. Eng. Notes 40 (1), 1–8.

Rosero, R.H., Gmez, O.S., Rodrguez, G., 2016. 15 years of software regression testing techniques a survey. Int. J. Softw. Eng.Knowl. Eng. 26 (5), 675–689.

Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B., 2002. The impact of test suite granularity on the cost-effectiveness of regression testing. In: Proceedings of the 24th International Conference on Software Engineering. ACM, pp. 130–140.

Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 1999. Test case prioritization: an empirical study. In: Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on. IEEE, pp. 179–188.

Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27 (10), 929–948.

Thomas, S.W., Hemmati, H., Hassan, A.E., Blostein, D., 2014. Static test case prioritization using topic models. Empir. Softw. Eng. 19 (1), 182–212.

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

Wang, R., Qu, B., Lu, Y., 2015. Empirical study of the effects of different profiles on regression test case reduction. IET Software 9 (2), 29–38.

Wong, W.E., Horgan, J.R., London, S., Agrawal, H., 1997. A study of effective regression testing in practice. In: Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on. IEEE, pp. 264–274.

Xu, D., Ding, J., 2010. Prioritizing state-based aspect tests. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE, pp. 265–274.

Yan, S., Chen, Z., Zhao, Z., Zhang, C., Zhou, Y., 2010. A dynamic test cluster sampling strategy by leveraging execution spectra information. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE, pp. 147–154.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.

Zhang, C., Chen, Z., Zhao, Z., Yan, S., Zhang, J., Xu, B., 2010. An improved regression test selection technique by clustering execution profiles. In: Quality Software (QSIC), 2010 10th International Conference on. IEEE, pp. 171–179.

Zhang, Z.-h., Mu, Y.-m., Tian, Y.-a., 2012. Test case prioritization for regression testing based on function call path. In: Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on. IEEE, pp. 1372–1375.

Jianlei Chi received the B.S. degree in computer science and technology from Harbin Engineering University, China, in 2010 and 2014. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Xian Jiaotong University, China. His research interests include trustworthy software, software testing, and software behavior analysis.

Yu Qu received the B.S. and Ph.D. degrees from Xian Jiaotong University, Xian, China in 2006 and 2015 respectively. He is a post-doctoral researcher at Department of Computer Science and Engineering, University of California, Riverside. His research interests include trustworthy software and applying complex network and data mining theories to analyzing software systems.

Qinghua Zheng received the B.S. degree incomputer software in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xian Jiaotong University, China. He was a postdoctoral researcher at Harvard University in 2002. He is currently a professor in Xian Jiaotong University, and the dean of the Department of Computer Science. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software.

Zijiang Yang is a professor in computer science at Western Michigan University. He holds a Ph.D. from the University of Pennsylvania, an M.S. from Rice University and a B.S. from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. He was also a visiting professor at the University of Michigan from 2009 to 2013. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xian Jiaotong University, Xian, China, in 2003 and 2010, respectively. Currently, he is an associate professor of the Systems Engineering Institute, Xian Jiaotong University. His research interests include smart grid, network security and trustworthy software.

Wuxia Jin is a Ph.D. candidate studying in Xian Jiaotong University, Xian, China. Her major is Computer Science and technology. Her research interests include trustworthy software and software analysis especially for distributed software system.

Di Cui is a Ph.D. candidate studying in Xian Jiaotong University, Xian, China. His major is Computer Science and technology. His research interests include trustworthy software, architecture recovery of software system.