Prioritizing JUnit Test Cases in Absence of Coverage Information

Lingming Zhang, Ji Zhou, Dan Hao*, Lu Zhang, Hong Mei
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, P. R. China
{*zhanglm07, zhouji07, haod, zhanglu, meih*}*@sei.pku.edu.cn*

Abstract

Better orderings of test cases can detect faults in less time with fewer resources, and thus make the debugging process earlier and accelerate software delivery. As a result, test case prioritization has become a hot topic in the research of regression testing. With the popularity of using the JUnit testing framework for developing Java software, researchers also paid attention to techniques for prioritizing JUnit test cases in regression testing of Java software. Typically, most of them are based on coverage information of test cases. However, coverage information may need extra costs to acquire. In this paper, we propose an approach (named Jupta) for prioritizing JUnit test cases in absence of coverage information. Jupta statically analyzes call graphs of JUnit test cases and the software under test to estimate the test ability (TA) of each test case. Furthermore, Jupta provides two prioritization techniques: the total TA based technique (denoted as JuptaT) and the additional TA based technique (denoted as JuptaA). To evaluate Jupta, we performed an experimental study on two open source Java programs, containing 11 versions in total. The experimental results indicate that Jupta is more effective and stable than the untreated orderings and Jupta is approximately as effective and stable as prioritization techniques using coverage information at the method level.

1. Introduction

Test suite reuse in the form of regression testing is prevalent in software evolution [13, 14]. Regression testing is a time-consuming task, accounting for as much as one-half of the cost in software maintenance [2, 11, 14]. For example, an industrial collaborator of Rothermel et al. [14] reported that running the entire test suite for one of their product costs nearly seven weeks. Test case prioritization aims at improving the efficiency of regression testing, and has been intensively studied. Generally speaking, test case prioritization techniques reorder test cases to maximize some objective function. For example, one such function is the rate of fault detection, indicating how quickly faults are exposed in testing. Rothermel et al. [14] formally define the test case prioritization problem as finding $T' \in PT$, such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$. In the definition, PT denotes the set of all possible permutations of a given test suite T, and f denotes a function from PT to real numbers.

Aiming to improve the rate of fault detection in regression testing, many prioritization techniques have been proposed. According to several existing studies [4, 6, 14, 20], most existing test case prioritization techniques are based on coverage information of the reused test suite. Typically, traditional techniques calculate a value for each test case based on coverage information, and prioritize test cases according to their values. Further more, these techniques can be categorized into total and additional techniques. While total techniques do not change values of test cases during the prioritization process, additional techniques adjust values of remaining test cases taking into account the influence of already prioritized test cases. However, test case prioritization techniques based on coverage information may have the following disadvantages. First, before applying these techniques, testers need to run instrumented source code to collect coverage information, and the process may be time-consuming. Second, the coverage information can be of large volume for large programs, and thus the storage and management of coverage information may be a burden for testers. Third, testers may modify some existing test cases in regression testing, making previous coverage information inconsistent with test cases in the test suite. These techniques may thus be negatively impacted. Finally, these techniques can hardly deal with test cases newly added for regression testing, whose coverage information is absent.

Nowadays, the JUnit testing framework has become a widely used facility for developing Java software. As shown

*Corresponding Author.

978-1-4244-4828-9/09/$25.00 2009 IEEE Proc. ICSM 2009, Edmonton, Canada

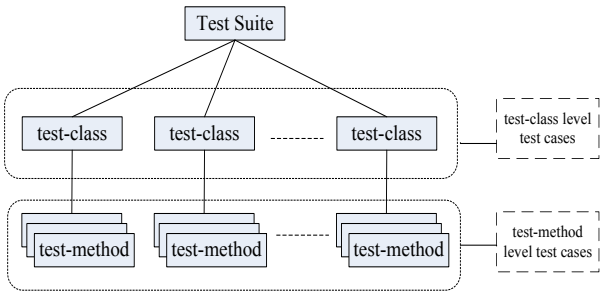

Figure 1. JUnit test suite structure

in Figure 1 [4], JUnit has two levels of test cases: the test-class level and the test-method level. The study by Do et al. [4] shows that traditional test case prioritization techniques can also be effective for prioritizing JUnit test cases. Besides, the study also shows that there is no much difference between test case prioritization at the test-method level and that at the test-class level. In this paper, we focus on prioritization at the test-method level, and the investigation of prioritization at the test-class level should be similar. For the ease of presentation, we refer to test-method level test cases simply as JUnit test cases in the rest of this paper. Different from traditional test case formats, a JUnit test case is a piece of executable source code containing some testing content (i.e., a sequence of method invocations). Thus, we can obtain the information of which methods each JUnit test case invokes, and use the information to guide the prioritization process.

In particular, we propose Jupta, an approach for prioritizing JUnit test cases based on statically analyzing the source code of JUnit test cases and the software under test without using any previous coverage information. In our approach, we define a metric to measure the testing ability (TA) of each JUnit test case, and use the TA values to guide the prioritization process. Analogous to traditional techniques, we present the total technique (named JuptaT) and the additional technique (named JuptaA) based on TA. To evaluate our approach, we performed an experimental study on two open source Java programs that have 11 versions in total. The experimental results indicate that Jupta is a competitive approach for prioritizing JUnit test cases. First, JuptaT and JuptaA outperform the untreated orderings by 23.95% and 70.81%, respectively. Second, JuptaT achieves 1.07% less than the total technique based on method coverage, and JuptaA achieves 1.62% less than the additional technique based on method coverage, indicating that Jupta is approximately as effective as the widely used techniques based on method coverage.

In summary, this paper makes the following main contributions:

- An approach for prioritizing JUnit test cases in absence of coverage information.
- Empirical evaluation of the proposed approach together with the total and the additional techniques based on method coverage, indicating that our approach is as effective as test case prioritization techniques based on coverage information.

The rest of this paper is organized as follows. Section 2 presents an example. Section 3 presents the details of our approach. Section 4 presents an experimental study. Section 5 presents related work. Section 6 concludes and discusses some future issues.

2. Example

Figure 2 depicts the source code of a Java class named $Account$ (Figure 2 (a)) and the JUnit code for testing class $Account$ (Figure 2 (b)). Class $Account$ represents a bank account, containing a constructor and six methods in charge of different operations for the bank account. To test class $Account$, we have six JUnit test cases (i.e., $test1$, $test2$, $test3$, $test4$, $test5$, and $test6$) depicted in Figure 2 (b). The problem is how to schedule the six test cases in an order that can expose faults early.

Although we have no coverage information, we can still have some clues by analyzing the source code of the program under test and the test cases. First, $test1$ tests only method $setBalance$ and $test2$ tests only $getBalance$, as the two test cases invoke only the two methods, respectively. Second, $test3$ tests $deposit$, $getBalance$, and $setBalance$, as $test3$ directly invokes $deposit$, which then invokes $getBalance$ and $setBalance$. Third, $test4$ tests $withdraw$, $getBalance$, and may test $setBalance$, as $test4$ invokes $withdraw$ and $getBalance$ directly, and $withdraw$ may invoke $setBalance$. Fourth, $test5$ tests $withdraw$, $sendto$, $getBalance$, and may test $setBalance$, as $test5$ invokes $withdraw$ and $sendto$ directly, and $withdraw$ invokes $getBalance$ and may take a branch to invoke $setBalance$. Finally, $test6$ tests $transfer$, $withdraw$, $getBalance$, and may test $setBalance$, $sendto$. With the preceding analysis, we can acquire information in the following two categories:

- **Which test cases may be more likely to expose more faults?** Let us consider test cases $test1$ and $test6$. Test case $test6$ may be more likely to expose more faults in $Account$. The reason is that $test6$ may expose faults in $transfer$, $withdraw$, $sendto$, $getBalance$, and $setBalance$, while $test1$ can only expose faults in $setBalance$. Thus, when prioritizing the six test cases, we may consider ordering $test6$ before $test1$.
- **Can some test cases be representative for others?** Let us consider test cases $test2$ and $test3$. Ac-

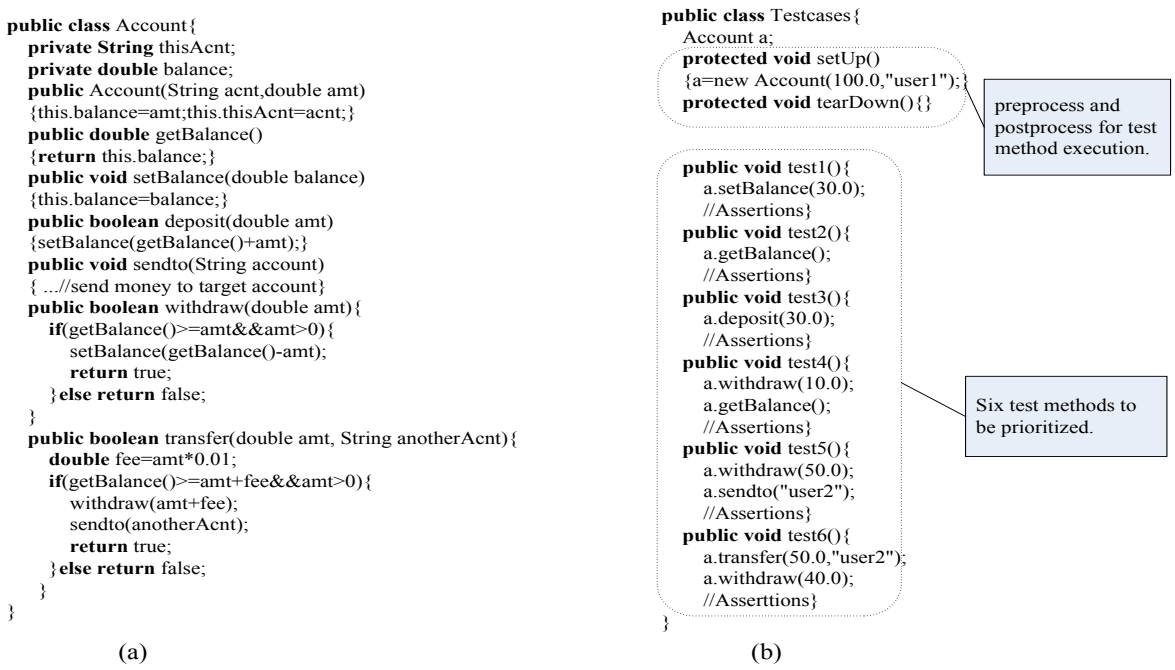

Figure 2. A small program dealing with simple cases in bank account and its JUnit test cases

cording to the preceding analysis, $test2$ only tests $getBalance$ and $test3$ tests $deposit$, $getBalance$, and $setBalance$. Thus, if we have already executed $test3$, the faults in $getBalance$ may have been revealed by $test3$, and executing $test2$ is not emergency, because $test2$ may only reveal the faults that have been exposed by $test3$. That is to say, when prioritizing the six test cases, setting $test3$ somewhere in an ordering should result in setting $test2$ further behind in the ordering.

According to the preceding reasoning, the source code of JUnit test cases and the software under test can provide useful information for test case prioritization. In fact, our Jupta approach mainly uses the preceding two categories of information. First, we use the test ability to quantify the information of which test cases may be more likely to expose more faults (Section 3.1). Second, for the additional technique of Jupta, after prioritizing each test case, we use the information of to what extent each remaining test case can be represented by already prioritized test cases to adjust the test ability of each remaining test case (Section 3.2.2).

3. Jupta

Given a set of test cases (denoted as TS) containing all the test cases initially, Jupta each time removes one test case with the highest test ability (TA) from TS, and adds it into a prioritized suite (denoted as PS). When TS becomes empty, Jupta stops the prioritization process and returns PS. In this section, we first present the way we calculate TA of each test case (Section 3.1); and then we present the two prioritization techniques based on TA calculation (Section 3.2).

3.1. Calculating TA

As Jupta prioritizes test cases in absence of coverage information, we need to measure the test ability of each test case without coverage information. From the example in Section 2, the test ability of one JUnit test case may relate to the methods called by the test case directly or indirectly in its static call graph. Therefore, before the definition of TA, we define the $relevant$ relation between methods of the program under test and test cases as follows:

Definition 1 *A method m of the program under test is $relevant$ to a JUnit test case t, iff m is covered by the static call graph of t.*

Note that Definition 1 is symmetric. That is to say, a method m is *relevant* to a test case t iff t is *relevant* to m.

Now, we define the test ability of one test case based on Definition 1 as follows:

Definition 2 *TA of test case t is the sum of FCI of methods that are* relevant *with t, i.e.,*

$$TA(t) = \sum_{m \in \gamma(t)} FCI(m).$$

```
Input: Test Case Set (denoted as TS)
Output: Prioritized Suite (denoted as PS)
Delaration: SlideSet: a data structure used to store
            test cases
General Process:
Begin
  SlideSet←Ø
  for each test case t in TS
    calculate initial TA for t
  end for
  while TS is not empty do
    for each test case t in TS
      adjust TA of t taking into account the influence
      of test cases in SlideSet
    end for
    select t with the highest TA in TS
    if TA(t)=0.0 then
      SlideSet←Ø
      continue
    end if
    TS←TS-{t}
    SlideSet←SlideSet U{t}
    add t to PS
  end while
  return PS
End
```

Figure 3. The general prioritization process of JuptaA

In Definition 2, $\gamma(t)$ denotes the set of methods *relevant* to t; $FCI(m)$ is the fault containing index (abbreviated as FCI) of method m, representing the probability that m contains faults in its body. The FCI value of each method may depend on many factors: developer ability, method complexity, changing history etc. In this paper, we treat all the methods equally, assuming that they have the same FCI values, i.e., for any two methods m_i and m_j in the software under test, $FCI(m_i) = FCI(m_j) = 1$. In other words, we use the methods covered by static call graphs to simulate the real method coverage and propose techniques to prioritize test cases.

3.2. Two techniques of Jupta

3.2.1. Total TA based prioritization. Initially, TS contains all the test cases to be prioritized and PS is empty. Jupta calculates initial TA of each test case in TS based on the call graph of the test case. Each time after Jupta selects a test case with the highest TA in TS[1], removes the test case from TS, Jupta adds the test case into PS. Jupta stops when the TS is empty. JuptaT is analogous to the total technique based on method coverage which prioritizes test cases according to the number of methods covered by test cases.

3.2.2. Additional TA based prioritization. As shown in Figure 3, the general process of this technique is similar to the first technique except that each time after Jupta selects a test case with the highest TA in TS, Jupta adjusts TA values of the remaining test cases in TS through consideration of the test cases in SlideSet (which stores already prioritized test cases). In the adjustment, the methods covered by the call graphs of already selected test cases in SlideSet will be ignored for TA calculation, because it is more likely that they expose the same faults exposed by the prioritized test cases. The adjustment is defined as below:

Definition 3 *The adjusted TA of test case t is defined in the following formula:*

$$TA_{adj}(t) = \sum_{m \in (\gamma(t) - \bigcup_{t' \in SS} \gamma(t'))} FCI(m).$$

In Definition 3, $TA_{adj}(t)$ is the adjusted TA of t; $\gamma(t')$ is the set of methods *relevant* with t'; $FCI(m)$ denotes the fault containing index of m.

In the prioritization process, the adjusted TA values of all test cases in TS may be 0 but TS is not empty. The reason is that each method in the call graph of each test case in TS is covered by one or more call graphs of test cases in SlideSet. In such a circumstance, the remaining test cases cannot be distinguished by their adjusted TA values. To address this issue, as framed in Figure 3, JuptaA starts another cycle of prioritization by clearing SlideSet. In fact, the entire prioritization process of this technique may contain many prioritization cycles, as the preceding circumstance may occur many times.

JuptaA is analogous to the additional technique based on method coverage which prioritizes test cases according to the coverage of methods not covered by already prioritized test cases.

3.3. Illustration with the example

Rothermel et al. [14] propose a metric APFD for measuring the rate of fault detection of test case prioritization techniques. The APFD can be computed as the following formula:

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

In this formula, n denotes the total number of test cases; m is the total number of faults; and TF_i is the smallest number of test cases testers have to go through in sequence until

[1] If more than one test case have the same highest TA, Jupta randomly selects one of the test cases with the highest TA.

Table 1. Fault exposing matrix of test cases depicted in Figure 2 (b).

Test Case	Faults					
	1	2	3	4	5	6
test1		X				
test2	X					
test3	X	X	X			
test4	X	X			X	
test5	X	X		X	X	
test6	X	X		X	X	X

fault i is exposed. APFD values range from 0% to 100%. For a specific test suite, n and m are specified, and thus a higher APFD value implies that the average value of TF_is is lower, implying a higher fault detection rate.

As the total technique of Jupta is easy to understand, in this subsection we illustrate the additional TA based technique of Jupta (JuptaA) with the example presented in Section 2, and results will be measured by the rate of fault detection using the APFD metric. For the ease of presentation, let us suppose that each of the six methods in $Account$ contains a fault and any test case calling a method can expose the corresponding fault.

The faults exposing information of the test cases (Figure 2 (b)) is shown in Table 1[2]. An item with "X" denotes that the corresponding test case can expose the corresponding fault, while an item with blank content denotes that the corresponding test case cannot expose the corresponding fault. Now we illustrate JuptaA step by step as follows:

Before prioritization, JuptaA extracts the static call graph of each test case (shown in Figure 4). Then JuptaA computes the initial TA of each test case according to Definitions 1 and 2.

Selection 1: In the first selection, JuptaA selects *test6* (which has the highest TA in TS) from TS. After the selection, SS becomes $\{test6\}$. Then, all the test cases in TS are recalculated for their TA according to Definition 3.

Selection 2: In the second selection, JuptaA selects *test3* (which has the highest TA in TS) from TS. After the selection, SS becomes $\{test6, test3\}$. Then, all the test cases in TS are recalculated for their TA according to Definition 3. As TA values of all test cases in TS are adjusted to 0 at this point, JuptaA clears SS and recalculates TA for test cases in TS.

Selection 3: In the third selection, JuptaA selects *test5* (which has the highest TA in TS) from TS. After the selection, SS becomes $\{test5\}$. Then, all the test cases in TS are recalculated for their TA according to Definition 3. As TA values of all test cases in TS are adjusted to 0 at this point, JuptaA clears SS and recalculates TA for test cases in TS.

[2]The faults in *getBalance*, *setBalance*, *deposit*, *sendto*, *withdraw*, *transfer* are numbered from 1 to 6, respectively.

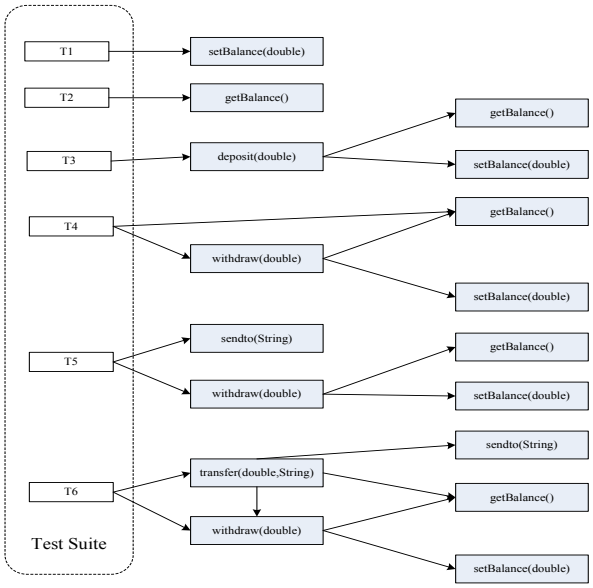

Figure 4. Static call graphs of test cases depicted in Figure 2 (b)

Selection 4: In the fourth selection, JuptaA selects *test4* (which has the highest TA in TS) from TS. After the selection, SS becomes $\{test4\}$. Then, all the test cases in TS are recalculated for their TA according to Definition 3. As TA values of all test cases in TS are adjusted to 0 at this point, JuptaA clears SS and recalculates TA for test cases in TS.

Selection 5: In the fifth selection, JuptaA selects *test2* (which is randomly selected from the 2 test cases with highest TA in TS) from TS. After the selection, SS becomes $\{test2\}$. Then, all the test cases in TS are recalculated for their TA according to Definition 3.

Selection 6: Finally, JuptaA selects *test1* as it is the only remaining test case in TS.

The details for each selection in JuptaA are shown in Table 2. The column "Sel" lists the numbering of selections. The column "Res" corresponds to the result of each selection. Other columns correspond to the TA value for each test case in each selection, where an item with "-" denotes that the test case has been selected and an item with "*" corresponds to the test case selected in the current selection.

We now get a fully prioritized test suite (*test6* → *test3* → *test5* → *test4* → *test2* → *test1*) with JuptaA after the preceding selections. To evaluate the effectiveness of JuptaA on this example, we first use APFD to measure the original permutation of the test cases (*test1* → *test2* → *test3* → *test4* → *test5* → *test6*), the faults detected versus the number of test cases used is depicted in Figure 5

Table 2. The prioritization process of test cases depicted in Figure 2 (b).

Sel	test1	test2	test3	test4	test5	test6	Res
1	1	1	3	3	4	*5	test6
2	0	0	*1	0	0	-	test3
3	1	1	-	3	*4	-	test5
4	1	1	-	*3	-	-	test4
5	1	*1	-	-	-	-	test2
6	*1	-	-	-	-	-	test1

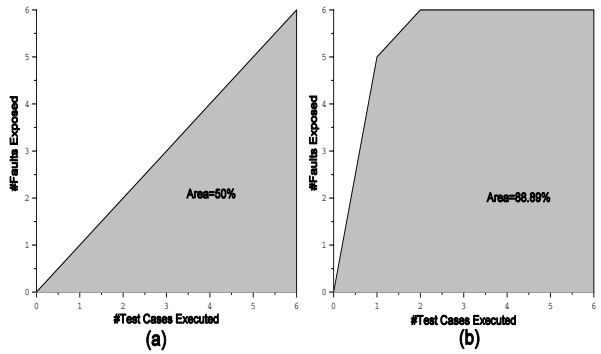

Figure 5. (a)APFD for original test case order. (b)APFD for test case order generated by JuptaA

(a). The curve represents the cumulative number of faults detected. The shaded area under the curve represents the APFD value obtained by this test ordering. The APFD for the original test ordering is 50%. Second, we use APFD to measure the test ordering (*test6* $\rightarrow$ *test3* $\rightarrow$ *test5* $\rightarrow$ *test4* $\rightarrow$ *test2* $\rightarrow$ *test1*) generated by JuptaA. As shown in Figure 5 (b), the APFD value achieved by this test ordering is 88.89%, which is 77.78% ($\frac{88.89\%-50\%}{50\%}$) higher than that achieved by the original test ordering.

4. Experimental study

4.1. Research questions

In our study, we are mainly concerned with the performance of Jupta. In particular, our study aims to answer the following two research questions:

RQ1: Is Jupta effective averagely on evolution of software and on different software?

RQ2: Is Jupta stable over evolution of software and over different software?

Note that the first research question is concerned with the effectiveness of Jupta, while the second research question is concerned with the stability of Jupta.

Table 3. Subjects

Programs	#Classes	#Test Methods	#Seeded Faults
jtopas-v1	19	126	10
jtopas-v2	21	128	12
jtopas-v3	50	209	16
ant-v1	128	137	11
ant-v2	342	219	21
ant-v3	342	219	7
ant-v4	532	521	26
ant-v5	536	557	15
ant-v6	536	559	1
ant-v7	649	877	29
ant-v8	650	878	4

4.2. Subjects

We used two Java programs (i.e., *jtopas* and *ant*) as subjects of our experimental study. Subject *jtopas*, implemented for parsing text data[3], has 5.4 KLOC (excluding comments). Subject *ant* is a Java-based build tool[4], which is similar to the widely known build tool named $make$ except that it is extended using Java classes instead of shell-based commands. Subject *ant* has a scale of 80.4 KLOC. We obtained fault seeded versions of programs *jtopas* and *ant* from Subject Infrastructure Repository (abbreviated as SIR) [3], which is a repository providing Java and C programs for controlled experimentation of program analysis and testing approaches. Each program has multiple versions seeded with faults, and each version has a JUnit test suite that was developed during software evolution, the details are shown in Table 3.

4.3. Experimental setup

To evaluate the rate of fault detection of Jupta, we use the APFD metric [14] to measure the performance of test case prioritization techniques and compare Jupta against untreated, random techniques, as well as two widely used method coverage based prioritization techniques (the total and the additional techniques based on method coverage) over the 11 subjects. The coverage information required by total and additional techniques based on method coverage is obtained by instrumentation implemented with the ASM bytecode manipulation framework [1].

Below is the prioritization techniques compared in our experimental study.

[3]http://jtopas.sourceforge.net/jtopas

[4]http://ant.apache.org

Table 4. Statistic results of APFD values obtained by UT, RT, MTT, MAT, JuptaT, JuptaA on versions of *jtopas*

Tech	Mean	Sd	Min	Max
UT	0.5420	0.2147	0.3500	0.7738
RT	0.6122	0.1922	0.4819	0.8330
MTT	0.5442	0.3542	0.2790	0.9464
MAT	0.8089	0.0128	0.8006	0.8237
JuptaT	0.6070	0.2271	0.3881	0.8415
JuptaA	0.7492	0.1233	0.6071	0.8281

Table 5. Statistic results of APFD values obtained by UT, RT, MTT, MAT, JuptaT, JuptaA on versions of *ant*

Tech	Mean	Sd	Min	Max
UT	0.4756	0.2722	0.1600	0.9601
RT	0.6976	0.1189	0.4944	0.8937
MTT	0.6465	0.1926	0.3836	0.9204
MAT	0.8754	0.0538	0.7942	0.9601
JuptaT	0.6138	0.2298	0.2120	0.9308
JuptaA	0.8787	0.0839	0.7710	0.9918

- **Untreated Technique (UT):** the original permutation of test cases provided by original test suites of subjects.
- **Random Technique (RT):** order test cases randomly (we obtain the result for random by averaging the results of 10 random permutations).
- **Method-total Technique (MTT):** prioritize test cases based on coverage of methods.
- **Method-additional Technique (MAT):** prioritize test cases based on coverage of methods not covered yet.
- **Total TA Based Technique (JuptaT):** prioritize test cases based on TA values of test cases.
- **Additional TA Based Technique (JuptaA):** prioritize test cases based on TA values of test cases, and also take into account the influence of test cases already prioritized.

4.4. Results and analysis

RQ1: *Is Jupta effective averagely on evolution of software and on different software?*

First, we evaluate the effectiveness of JuptaT and JuptaA over software evolution by analyzing the results for *jtopas* and *ant* separately. As shown in Table 4, for versions of *jtopas*, on average, JuptaT achieves an APFD value of 60.70%, and JuptaA achieves 74.92%, both outperforming UT significantly. JuptaT is relatively higher than MTT (11.54% over MTT), whereas JuptaA is relatively lower than MAT by 7.37% on versions of *jtopas*. Things are different on versions of *ant*. Shown in Table 5, on average, JuptaT achieves an APFD value lower than MTT by 5.05%, whereas JuptaA outperforms MAT with a numerically small improvement (0.39%). Besides, on versions of *ant*, JuptaA achieves the highest APFD value among all the six techniques. In summary, we can find that the JuptaT and JuptaA perform approximately as effectively as method coverage based techniques on versions of *jtopas* and *ant*.

Second, we evaluate the effectiveness of Jupta over different software by analyzing the performance of prioritization techniques over all versions of the two programs together. Over all the 11 subjects, APFD values obtained by UT, RT, MTT, MAT, JuptaT, and JuptaA are shown in Figure 6. For total strategy based techniques, on average, JuptaT achieves an APFD value of 61.20% while MTT achieves 61.90%. Besides, for additional strategy based techniques, on average, JuptaA achieves an APFD value of 84.34% while MAT achieves 85.73%. To conclude, both total and additional techniques of Jupta perform as effectively as corresponding techniques of method coverage based prioritization.

The experimental results indicate that, on average, Jupta can be as effective as coverage information based techniques. However, there are some specific circumstances that deserve discussion.

First, analyzing the APFD values achieved by Jupta on all versions of *jtopas* (shown in Figure 6), we observe that JuptaA performs better than JuptaT except on *jtopas-v3*. For *jtopas-v3*, MAT also does not perform as well as MTT. This is different from the trend in other circumstances. We further checked the source code and the execution of each test case. We found that the main reason for the exception is as follows. Many faults seeded in *jtopas-v3* were not exposed by the first test case covering the method containing the faults. Thus, techniques aiming to test more untested methods may be less effective than techniques aiming to test the faulty methods repeatedly. For example, JuptaA ranked test cases $testJavaTokenizer$, $compareSpeed1$, and $testParallelParsing$ as the 1st, the 3rd, and 23rd respectively . We found that method $setSource$ seeded with a fault was executed by test case $testJavaTokenizer$, but $testJavaTokenizer$ did not expose the fault. This fault was not exposed until $setSource$ was invoked when executing test case $compareSpeed1$. Similarly, the constructor method of $StandardTokenizerProperties$ was covered when executing test case $testJavaTokenizer$, but the fault seeded in it was not exposed until executing test case $testParallelParsing$. The preceding two and other delays in fault exposure significantly influence the performance

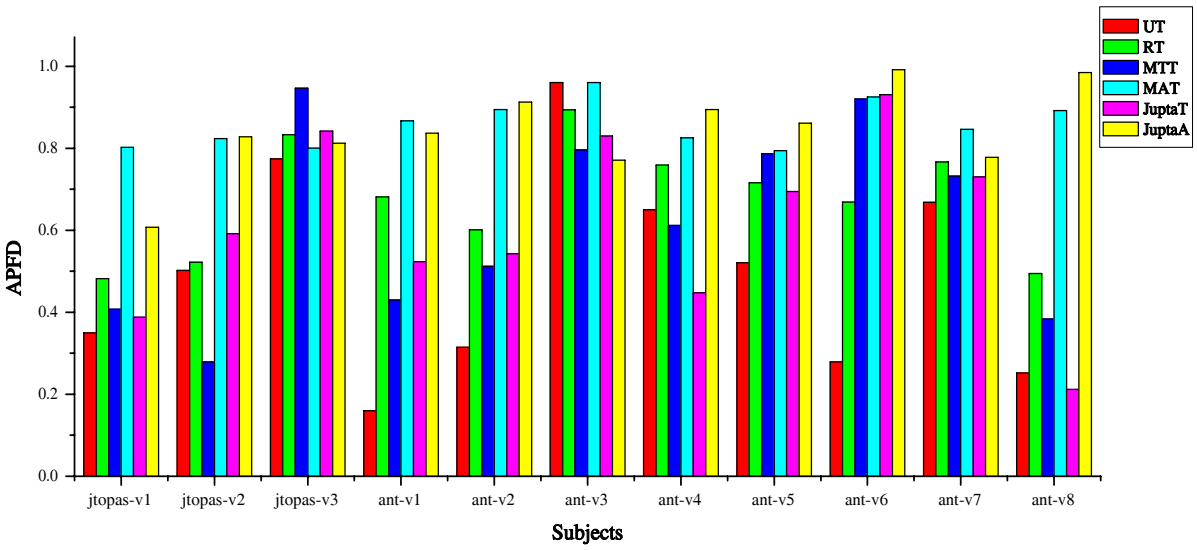

Figure 6. APFD values achieved by UT, RT, MTT, MAT, JuptaT, and JuptaA over 11 versions of *jtopas* and *ant*

of JuptaA on *jtopas-v3*. We think this indicates a weak point for JuptaA and additional coverage based prioritization techniques, for their effectiveness is based on the assumption that faults are very likely to be exposed when the entities containing them are executed for the first time.

Second, analyzing the APFD values achieved by Jupta on versions of *ant* (shown in Figure 6), we find that JuptaA does not perform well on *ant-v3*, for its APFD value is lower than that of UT and RT. We also find that there is a extremely high value for UT on *ant-v3*. It is unusual, as APFD values achieved by UT are almost the lowest on all other subjects. After analyzing the faults seeded in *ant-v3*, we observe that only two of the seven seeded faults can be exposed by test cases and both the two faults can be exposed by 9 test cases with relatively small call graphs. Therefore, the test cases with complex call graphs ranked high by JuptaA cannot expose the fault, while RT and UT can easily achieve good effect as many test cases can expose the faults. We think this kind of faults are not representative for general circumstances, because faults should be well distributed in programs and be exposed by a relatively small number of test cases in general circumstances.

Third, we observe that, on average, RT outperforms the two total techniques (JuptaT and MTT) on the 11 subjects. As we know, while additional techniques suppose faults are easy to expose, total prioritization techniques assume faults can only be exposed by repeatedly testing, thus they prioritize test cases without considering the influence of already prioritized test cases. In this 11 subjects, on average, additional techniques significantly outperforms total techniques (shown in Figure 6), indicating that faults in the subjects are relatively easy to expose. In this circumstance, total techniques may continually selects test cases testing parts where faults have been exposed or faultless parts, making it even less effective than RT.

RQ2: *Is Jupta stable over evolution of software and over different software?*

First, we evaluate the stability of Jupta over evolution of software. The standard deviation (SD) of JuptaT APFD values over verions of *jtopas* is 0.2271, which is smaller than that of MTT technique (0.3542). The standard deviation of JuptaA APFD values over the three versions of *jtopas* is only 0.1233 (Table 4), which is higher than that of MAT, but smaller than all others, indicating a stable performance over different versions of *jtopas*. Similar with the results on versions of $jtopas$, the standard deviation of JuptaT and JuptaA APFD values over the 8 versions of ant are 0.2298 and 0.0839 respectively (Table 5), indicating that Jupta is also stable over evolution of *ant*.

Second, we evaluate the stability of Jupta over different software. Over all the 11 subjects, APFD values obtained by UT, RT, MTT, MAT, JuptaT, and JuptaA are shown in Figure 6. The standard deviation of JuptaT over the 11 subjects is 0.2175, which is smaller than that of MTT. Moreover, the standard deviation of JuptaA is only 0.1078, which is the second lowest among the six techniques (only a little higher than that of MAT), showing that Jupta is stable over subjects in our experimental study.

According to the preceding analysis, the experimental study indicates that Jupta is relatively stable over evolution

of software and over different software.

The answers to RQ1 and RQ2 show that, although Jupta does not need coverage information, it is able to significantly outperform the untreated strategy. Besides, Jupta is able to perform as effectively as techniques based on coverage information. The results of the experimental study show that Jupta can be a good choice for test case prioritization in regression testing, because it delivers relatively effective and stable performance without engaging testers into tedious and costly coverage information collection, storage, and management.

4.5. Threats to validity

The main threat to internal validity is the possible faults in our implementation of the experimented techniques in our study. To reduce this threat, the first and the second authors of this paper implemented two versions of all the techniques respectively, we ensured that both implementations produced the same results in our experimental study. Furthermore, we reviewed all the code we produced to guarantee its correctness.

One main threat to external validity is the subjects used in our study. To reduce this threat, we conducted experiments on 11 versions of two open source Java programs ranging from 5.4KLOC to 80.4KLOC. However, they may be still not representative for other programs. Another threat to external validity is the quality of test cases. To reduce this threat, we use the test cases developed in the real evolution of subjects. The faults seeded in subjects is also a threat to external validity. To reduce this threat, we obtained the fault seeded versions of subjects from SIR [3]. However, the faults may not be representative for general kind of faults.

The main threat to construct validity is the metric to assess the effectiveness of prioritization techniques. To reduce this threat, we used the APFD metric that was proposed by Rothermel et al. [14] and widely used in research of test case prioritization.

5. Related work

Most of the test case prioritization techniques focus on how to use the previous coverage information efficiently. Wong et al. [20] suggest prioritizing test cases based on "increasing cost per additional coverage" . They take advantage of source code changes between versions and previous execution traces to prioritize test cases. The experimental result gives a good instruction on how to test software more effectively within limited resources. Rothermel et al. [14, 15] present a family of prioritization techniques at the statement level and give the metric APFD for assessing the rate of fault detection. Through empirical studies, they give the conclusion that these techniques (even the least sophisticated ones) can help improve the effectiveness in regression testing. Elbaum et al. [16] consider performance goals at a coarser granularity, and develop a set of techniques at the function level. Taking the test costs and fault severity into account, Elbaum et al. [5] propose a new metric APFDc, which incorporates different test costs and fault severity into assessment of the different prioritization techniques. Jones et al. [8] propose efficient prioritization algorithms to use adequate modified coverage/decision coverage. Kim and Porter [9] utilize the history information in previous versions to prioritize test cases. Jeffrey and Gupta [7] present a test case prioritization technique based on relevant slices. This approach focuses on the executed statements and branches that affect or potentially affect the test case output through relevant slices. Taking into account different testing requirement priorities and test case costs, Zhang et al. [22] suggest a general test case prioritization technique. Some researchers [19, 21] also investigate test case prioritization techniques under time constraints. However, before applying these coverage based techniques, testers have to go through time-consuming execution of instrumented code to collect coverage information.

There are also some techniques focusing on utilizing the information of current programs and test cases. Srikanth et al. [17] propose a test case prioritization approach based on combination of several factors: customer-assigned priority, requirements volatility, implementation complexity, and fault proneness of the requirements. This approach identifies the importance of requirements by their weights assigned by users. Tonella et al. [18] take relative priority information from users and integrate it with multiple prioritization indices. Then machine learning is used to improve the prioritization result. Korel et al. [10] prioritize test cases basing on EFSM (which stands for Extended Finite State Machine) system models. Ma and Zhao [12] distinguish fault severity using both users' knowledge and program information, aiming to detect severe faults first. However, these techniques cannot help much without user involvement. Moreover, these techniques all assume that users are familiar with the program under test and their knowledge is correct and accurate.

Different from existing techniques based on coverage information or user involvement, we propose Jupta, a static test case prioritization approach scheduling test cases according to their Test Ability (TA) values derived from static call graphs. According to our experimental study, although Jupta does not need the coverage information, it can perform as effectively as widely used techniques based on method coverage.

6. Conclusion and future work

In this paper, we propose Jupta, a static approach for prioritizing JUnit test cases in absence of coverage information. Jupta prioritizes test cases according to the static call graphs of test cases. Although Jupta does not rely on any previous coverage information, Jupta is able to perform as effective as techniques based on coverage information according to our controlled experimental study. The results of the experimental study show that Jupta can be a good choice for test case prioritization in regression testing, because it renders testers relatively effective and stable prioritization results without engaging testers into tedious and costly coverage information collection, storage, and management.

In the next stage, we will compare Jupta against other coverage information based prioritization techniques through doing experiments on more large scale real programs of different fields. Besides, we plan to improve Jupta by utilizing static analysis results to specify FCI for methods, and conduct experiments to evaluate the effectiveness of the improved Jupta. In addition, we intend to implement Jupta as an Eclipse IDE plug-in to facilitate regression testing under the JUnit testing framework.

Acknowledgments

This research is sponsored by the National Basic Research Program of China under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, the National Natural Science Foundation of China under Grant No. 60803012, and China Postdoctoral Science Foundation funded project (No.20080440254).

References

[1] The ASM project web site, http://asm.objectweb.org/.

[2] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co. New York, NY, USA, 1990.

[3] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[4] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *International Symposium on Software Reliability Engineering*, pages 113–124, 2004.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *International Conference on Software Engineering*, pages 329–338, 2001.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[7] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *International Computer Software Applications Conference*, pages 411–420, 2006.

[8] J. Jones and M. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, 2003.

[9] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *International Conference on Software Engineering*, pages 119–129, 2002.

[10] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of System Models in Regression Test Suite Prioritization. In *International Conference on Software Maintenance*, pages 247–256, 2008.

[11] H. Leung and L. White. Insights into regression testing. In *International Conference on Software Maintenance*, pages 60–69, 1989.

[12] Z. Ma and J. Zhao. Test Case Prioritization Based on Analysis of Program Structure. In *Asia-Pacific Software Engineering Conference*, pages 471–478, 2008.

[13] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

[14] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *International Conference on Software Maintenance*, pages 179–188, 1999.

[15] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, (10):929–948, 2001.

[16] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[17] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. *International Symposium on Empirical Software Engineering*, pages 64–73, 2005.

[18] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *International Conference on Software Maintenance*, pages 123–133, 2006.

[19] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2006.

[20] W. Wong, J. Horgan, S. London, H. Agrawal, and M. Bellcore. A study of effective regression testing in practice. In *International Symposium On Software Reliability Engineering*, pages 264–274, 1997.

[21] L. Zhang, S. Hou, C. Guo, T. Xie, and H. Mei. Time-aware test-case prioritization using integer linear programming. In *International Symposium on Software Testing and Analysis*, 2009.

[22] X. Zhang, C. Nie, B. Xu, and B. Qu. Test case prioritization based on varying testing requirement priorities and test case costs. In *International Conference on Quality Software*, pages 15–24, 2007.