Cost and Coverage Metrics for Measuring the Effectiveness of Test Case Prioritization Techniques

MS.A.ASKARUNISA [1]
MS. L.SHANMUGAPRIYA [2]
DR.N.RAMARAJ [3]

Thiagarajar College of Engineering, Madurai
Thiagarajar College of Engineering, Madurai
GKM Engineering College, Chennai
[1] (aacse)@tce.edu
[2] ()@tce.edu
[3] nishanazer@yahoo.com

Abstract. Regression testing is an important and also a very expensive activity in the software life cycle. To reduce the cost of regression testing, test cases are prioritized. One goal of test case prioritization technique is to increase a test suite's rate of fault detection and to reduce the cost of regression testing. In his paper G. Rothermel [9] has provided a metric, Average Percentage of Fault Detected (APFD), for measuring rate of fault detection during prioritization. This metric assumes that all test cases and fault costs are uniform. In practice, test case and fault costs may vary, and in such cases the previous APFD metric can be unsatisfactory. This paper presents a metric for assessing the rate of fault detection of prioritized test cases, APFDc, that incorporates varying test cases and fault costs. We have also calculated other new metrics like Average Percentage of Statement Coverage (APSC), Average Percentage of Branch Coverage (APBC),Average Percentage of Loop Coverage (APLC) and Average Percentage of Condition Coverage (APCC) based on the coverage criterion for the various prioritization techniques performed. Test cases are executed using JUnit tool. Code cover tool is used to find code coverage information. Test case prioritization is performed based on coverage and cost information. By injecting mutation faults effectiveness of prioritization is measured. Finally, we have implemented all the metrics considering a few standard java programs.

Keywords: Regression Testing, code coverage, test case prioritization, mutation faults, Average percentage of Fault Detection (APFD), Average percentage of Fault Detection with cost (APFDc).

(Received April 01, 2009 / Accepted July 04, 2009)

1 Introduction

Regression testing is needed to detect whether new faults have been introduced into previously tested code and whether newly added code functions according to specification. Regression testing is an important activity in the software life cycle, but it can also be very expensive and can account more cost [5]. During Regression testing, the test cases can either be reduced or optimized or prioritized so as to reduce the time, cost and resources of the testing process.Test case prioritization techniques schedule test cases for regression testing in an order that attempts to maximize some objective function, such as achieving code coverage quickly or improving rate of fault detection. An improved rate of fault detection can provide earlier feedback on the system under test, enable earlier debugging.[1,2,3,7].
In his paper G. Rothermel [9] has provided a metric, APFD, for measuring rate of fault detection, and tech-

niques for prioritizing test cases in order to improve APFD. This metric assumes that all test cases and fault costs are uniform. In practice, test cases and fault costs may vary, and in such cases the previous APFD metric can be unsatisfactory. This paper presents a metric for assessing the rate of fault detection of prioritized test cases, APFDc, that incorporates varying test cases and fault costs. These techniques enable practitioners to perform a new type of prioritization: cost-oriented test case prioritization. We have also calculated other new metrices like APSC [11], APBC, APLC and APCC based on the coverage criterion for the various prioritization techniques performed.

Test cases are executed using JUnit tool [13]. Code cover tool [14] is used to find code coverage information like Statement coverage, branch coverage, loop coverage and condition coverage. Test case prioritization is performed based on coverage and cost information. The effectiveness of the various prioritization techniques can be obtained by injecting faults. Regression faults vary in two ways: by locating naturally occurring faults and by seeding faults. Naturally occurring faults offer external validity, but they are costly to locate and often cannot be found in numbers sufficient to support controlled experimentation[9].But seeded faults, (hand-seeded or mutation faults) can be provided in larger numbers, allowing more data to be gathered. In this paper we have calculated the effectiveness by injecting mutation faults. We use MuJava tool [16] for generating mutants. Finally, we have analyzed our results with a few standard java programs.

Section 2 discusses the prior work on regression testing. In section3 proposed work on Test Case Prioritization is discussed. In Section 4 various Test case prioritization metrices are compared. Section 5 discusses the experimental programs, we have taken. In section 6 the metrics were implemented and the results obtained.

2 Definition of Regression Testing

Regression testing can be defined as follows:

Let P be a program, let P' be a modified version of P, and let T be a test suite developed for P. Regression testing is concerned with validating P'. To facilitate regression testing, engineers typically re-use T, but new test cases may also be required to test new functionality. Both reuse of T and creation of new test cases are important; however, it is test case reuse that is of concern here, as such reuse typically forms a part of regression testing processes [5].

2.1 Related work on Regression Testing Methodologies

In particular, researchers have considered four methodologies related to regression testing and test reuse: retest-all, regression test selection, test suite reduction, and test case prioritization. This section provides additional background on the various methodologies of regression testing [10].

2.1.1 Retest-all

When P is modified, creating P', test engineers may simply reuse all non-obsolete test cases in T to test P', this is known as the retest-all technique [5].

2.1.2 Regression Test Selection

The retest-all technique can be expensive: Regression test selection (RTS) techniques (e.g., [12, 9]) use information about P, P', and T to select a subset of T with which to test P'. One cost-benefit tradeoff among RTS techniques involves safety and efficiency. Safe RTS techniques guarantee that, under certain conditions, test cases not selected could not have exposed faults in P' [12].

2.1.3 Test Suite Reduction

Test suite reduction techniques remove redundant test cases from T by using information about P and T .

2.1.4 Test Case Prioritization

Test case prioritization techniques [6,9,10], schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases. A potential advantage of these techniques is that unlike test case reduction and non-safe regression test selection techniques, they do not discard tests. Many different prioritization techniques have been proposed [4,8,9], but the techniques utilize simple code coverage information like statement and method coverage.

3 Proposed work on Test Case Prioritization

The prioritization techniques most prevalent in literature and practice involve those that utilize simple (statement and method) code coverage information. We have performed test case prioritization based on all types of coverage's like Statement coverage, branch cover-

age, loop coverage and condition coverage including varying cost. We have developed a test case prioritization framework that prioritizes the various test cases in two ways viz. Coverage based and Cost oriented as shown in Figure 1. The description of the framework is as follows:

3.1 Test case generation

Test case is a combination of inputs, executing function, expected outputs. We have used the JUnit framework [13] for executing unit tests. JUnit test cases are Java classes that contain one or more test methods that are grouped into test suites. A test case tests the response of a single method to a particular set of inputs. For implementation we have used a calculator program that performs the basic functions of scientific calculator and have generated nearly 26 test cases and executed them using JUnit.

3.2 Measuring Code coverage

After generating test cases we found coverage's for each test case using Code cover tool [14,17]. Coverage measurement provides percentage of code that is covered by test case. Code cover is an open source tool for finding coverage and it is integrated with JUnit test cases. This tool supports statement coverage, branch coverage, loop coverage and condition coverage. The details of the various coverage percentages for the 26 test cases of Calculator program are shown in Table 1.
We have calculated the total coverage for the calculator

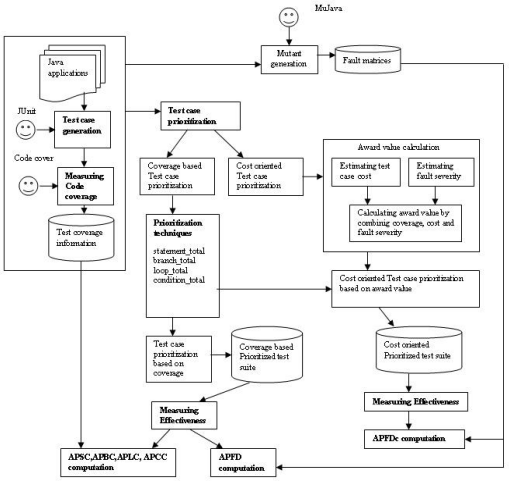

Figure 1: Overview of Test case prioritization.

program as 99% statement coverage,96% branch coverage, 70%loop coverage and 100%condition coverage.

Table 1: Coverage percentage for Calculator program

$TestCase$	$Statement$ $(in\%)$	$Branch$ $(in\%)$	$Loop$ $(in\%)$	$StrictCondition$ $(in\%)$
1	8.1	8.7	0	0
2	8.1	8.7	11.3	0
3	8.1	8.7	0	0
4	11.3	8.7	22.2	11.1
5	6.5	8.7	0	0
6	9.7	8.7	11.1	0
7	9.7	8.7	11	0
8	22.9	5.7	22.2	11.1
.	.	.	.	.
25	4.3	4.3	0	0
26	4.3	4.3	0	0

3.3 Test Case Prioritization

Test case prioritization techniques schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases.Previous work [10] has described the test case prioritization problem as follows:

Definition: The Test Case Prioritization Problem Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.
Problem: : Find $T'\epsilon$PT such that (for all T ") (T" ϵPT) (T" $\neq$T') [f(T') $\geq$ f(T")]. In our paper we have implemented two kinds of Prioritization methods viz. Coverage based and Cost oriented that are explained in the following section.

3.3.1Coverage based Test case prioritization

The various prioritization techniques are implemented based on code coverage information as shown in **Stmt_total**

Table 2: Test case Prioritization techniques

$Mnemonic$	$Description$
$Stmt\_total$	$Prioritizeinorder$ $oftotalstatementsCovered$
$Branch\_total$	$Prioritizeinorder$ $oftotalbranchesCovered$
$Loop\_total$	$Prioritizeinorder$ $oftotalloopsCovered$
$Condition\_total$	$Prioritizeinorderof$ $totalbasicbooleantermsCovered$

prioritization: prioritizes test cases according to the total number of statements they cover.
Branch_total prioritization: Same as stmt_total prioritization except that they rely on coverage measured in terms of numbers of branches executed.
Loop_total prioritization: Same as stmt_total priori-

tization except that they rely on coverage measured in terms of numbers of loops executed.
Condition_total prioritization: Same as stmt_total prioritization except that they rely on coverage measured in terms of numbers of basic Boolean terms executed.
Prioritization was performed for the calculator program for 26 test cases for the various techniques mentioned in Table 2.The results are detailed in Table 3.

Table 3: Prioritization order based on coverage for Calculator Program

Prioritization Techniques	*Prioritization Order*
NoPrioritization	$1-2-3-4-5-6-7-8$ $-9-10-11-12-13-14-15$ $-16-17-18-19-20-21$ $-22-23-24-25-26$
Stmt_total	$8-4-12-6-7-1-2-3-$ $10-11-5-912-16-18-14-$ $15-17-19-20-21-22$ $-23-24-25-26$
Branch_total	$13-16-18-1-2-3$ $-4-5-6-7-8-9-10-11$ $-12-14-15-17-19-20-$ $21-22-23-24-25-26$
Loop_total	$4-8-11-2-3-6-7-10$ $-11-1-5-9-13-14-$ $15-16-17-18-19-20-$ $21-22-23-24-25-26$
Condition_total	$4-8-12-13-16-13$ $1-2-3-5-6-7-9-10-$ $11-14-15-17-19-20-$ $21-22-23-24-25-26$

3.3.2 Cost oriented Test case prioritization

Input: Test suite T, Test costs cos_t, Test faults T_f, Test fault severity T_{sev}, Test coverage cov_t
Output: Prioritized test suite T
1: begin
2: set T' empty
3: for each test case t ? T do
4: calculate total severity $tsev_t$ of t by using T_{sev}
5: calculate criticality $_t$ of t byusing $tsev_t$ andT_f
6. calculate award value at of t as cov_t* g(criticality$_t$; cost$_t$)
7: end for
8: sort T in descending order based on the award value of each test case
9: let T' be T
10: end

Algorithm 1: Cost-oriented total statement coverage prioritization.

In this method the test cases are prioritized based on the various techniques as shown in Table 2 by combining cost and coverage measurement [10]. Algorithm 1 discusses the process of performing Cost-oriented total statement coverage prioritization. To implement the above algorithm we need Test case cost, test case coverage and fault severity.

3.3.2.1 Estimating the test case cost

The cost of a test case is related to the resources required to execute and validate it. Various measures are possible like

- When the primary required resource is machine or human time, test cost can be measured in terms of the actual time needed to execute a test case.
- Another measure considers the monetary costs of test case execution and validation; this may reflect hardware cost, wages, cost of materials required for testing, earnings lost due to delays in failing to meet target release dates, and so on . We have estimated cost by measuring the actual time required to execute each test case.Estimated cost for Calculator program is shown in Table 4.

Table 4: Sample cost for Calculator Program

TestCase	*Cost(Excecution timeinseconds)*
1	0.281
2	0.26
.	.
25	0.721
26	0.27

We have estimated cost by measuring the actual time required to execute each test case. . Estimated cost for Calculator program is shown in Table 4.

3.3.2.2 Fault Severity

For estimating the fault severity we have to generate faults and performing mutation testing.

- **Fault generation**Regression faults vary in two ways: by locating naturally occurring faults and by seeding faults. Naturally occurring faults offer external validity, but they are costly to locate and often cannot be found in numbers sufficient to support controlled experimentation[9].But seeded faults,hand-seeded or mutation faults can be provided in larger numbers, allowing more data to be gathered. In this paper we have calculated the effectiveness by injecting mutation faults. We have used MuJava tool [16] for generating mutants.

- **Mutation testing**(sometimes also called mutation analysis) is a method of software testing, which involves modifying program's source code in small. Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a mutant. The mutation operators in MuJava tool are shown in Table 5.

  For example, AOP operator replaces addition operator with a subtraction, multiplication, or division operator [16].For the Calculator program 288 mutants were generated. From these 25 mutants have been selected randomly and applied to the code and the fault matrix was constructed. The fault matrix for Calculator program is shown in Table 6.

Table 5: Sample cost for Calculator Program

$Operators$	$Descriptions$
AOP	$ArithmeticOperatorChange$
LCC	$LogicalConnectorChange$
ROC	$RelationalOperatorChange$
APC	$AccessFlagChange$
OVD	$OverridingVariableDeletion$
OVI	$OverridingVariableInsertion$
OMD	$OverridingMethodDeletion$
AOC	$ArgumentOrderChange$

Table 6: Sample fault matrix for Calculator program

$TestCase$	Fault								
	1	2	3	4	5	6	.	24	25
1	X								
2	X								
3	X								
4	X	X							
.	.	.	.	.	.	.	.	.	.
25	X						X		
26	X				X				

- **Estimating fault severity** Cost-cognizant prioritization also requires an estimate of the severity of each fault that can be revealed by a test case.Two possible estimation approaches, however, involve assessing module criticality and test criticality [10]. Module criticality assigns fault severity based on the importance of the module ,or some other code component such as a block, function, object, in which a fault may occur, while test criticality is assigned directly to test cases based on an estimate of the importance of the test, or the severity of the faults each test case may detect.

We have considered module criticality for estimating fault severity. We ranked faults based on severity values like "Cosmetic", "Moderate", "Severe", "Critical" as shown in Table 7 and based on these ranked faults we estimated the fault severity. Total severity , the total of all the severities for the test cases) and average severity, Total severity / Number of faults,for each test case were calculated.For the Calculator program the fault severity values are shown in Table 8.

Table 7: Fault severity values

$Severity\ Values$	$Severity\ Levels$	$Description$
4	$Critical$	$Productisusable$
3	$Severe$	$Productfeaturecannot beused, noworkaround$
2	$Moderate$	$Productfeaturecannot beused, onlywithworkaround$
1	$Cosmetic$	$producthasaminorinconvenience$

3.3.2.3 Award value calculation and Prioritization

For Cost oriented prioritization the test cases are prioritized based on award value.Award value(a_t)is calculated using the formula,**Award value(a_t)=cov$_t$*g (criticality$_t$,cost$_t$)**.Where criticality$_t$ - is an estimate of the average severity of faults detected by test case t,cost$_t$ - is the cost of t, and g - is a function that maps the criticality and cost of t into a value. (Function g simply divides criticality$_t$ by the cost$_t$.) The result is an award value at for t. The notion behind the use of this computation is to reward test cases that have greater cost adjustments when weighted by the total number of statements they cover.Calculated award value for Calculator program is shown in Table 8.Test case which has high award value than other test cases is executed first and so

Table 8: Fault severity values

$TestCase$	$Total\ fault\ Severity$	$Noof\ faults$	$Average\ fault\ Severity$	$Award\ Value$
1	6	2	3	85.409
2	7	3	2.3333	47.56
3	7	3	2.3333	2.708
4	13	5	26	46.598
.	.	.	.	.
25	1	1	1	7.350
26	1	1	1	19.629

on.Prioritization was performed for the calculator program for 26 test cases for the various techniques mentioned in Table 2.The results are detailed in Table 9.

Table 9: Prioritization order based on cost for Calculator Program

Prioritization Techniques	*Prioritization Order*
No Prioritization	$1-2-3-4-5-6-7-8$ $-9-10-11-12-13-14-15$ $-16-17-18-19-20-21$ $-22-23-24-25-26$
Stmt_ total	$1-10-5-20-11-18$ $-9-16-2-7-19-13$ $-15-6-2-14-26-21-17-$ $23-12-25-8-3-24$
Branch_total	$18-16-1-2-10-13-5-$ $11-9-15-7-20-6-4$ $-14-19-17-22-26-21$ $-23-12-25-3-8-24$
Loop_total	$4-2-10-11-7-6-$ $12-8-3-1-5-9-13-14$ $-15-16-17-18-19-20-21-$ $22-23-24-25-26$
Condition_total	$18-16-13-4-12-8-$ $1-2-3-5-6-7-9-10-11-$ $14-15-17-19-20-21-22$ $-23-24-25-26$

3.4 Measuring Effectiveness of the various prioritization Techniques

In our work we have predicted the effectiveness [9,10,11] of coverage based prioritization techniques using APSC, APBC,APLC APCC ,APFD metrices and for cost oriented prioritization techniques using APFDc metric.

3.4.1 APSC,APBC,APLC APCC metrics

Depending on the coverage criterion (statement, branch etc) considered [11], the following metrices were computed. In [11] APLC and APCC metrices were not calculated but we have implemented them in addition.

1. APSC (Average Percentage Statement Coverage). This measures the rate at which a prioritized test suite covers the statements.[11]
2. APBC (Average Percentage Branch Coverage). This measures the rate at which a prioritized test suite covers the branches. This metric is also represented as Average Percentage Block Coverage [11].
3. APLC (Average Percentage Loop Coverage). This measures the rate at which a prioritized test suite covers the loops.
4. APCC (Average Percentage Condition Coverage). This measures the rate at which a prioritized test suite covers the conditions.

Table 10: Test suite and branch coverage

Test	Branches Covered									
Case	1	2	3	4	5	6	7	8	9	10
1		X	X	X						
2	X	X	X	X	X					
3					X	X	X	X		
4				X						X
5		X			X		X			
6	X			X						X
.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	
25	X				X		X			
26						X				X

For example in calculating APBC metric, a test suite T containing n test cases that covers a set of m branches is taken.Let TFi be the first test case in ordering T' of T which covers branch i. The APBC for test suite T' is given by the equation.

$$APBC = 1 - \frac{TB1 + TB2 + \_\_ + TBm}{nm} + \frac{1}{2n}$$

APBC measures the weighted average of the percentage of branches covered over the life of a test suite. APBC values range from 0 to 100;higher number imply faster coverage rates.To illustrate this measure, consider the Calculator program with 11 branches and a suite of 26 test cases 1 through 26, each with branch coverage characteristic as shown in Table10.Consider two orders of these test cases, order T1(No prioritization):1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26,and order T2(Branch_total prioritization):13-16-18-1-2-3-4-5-6-7-8-9-10-11-12-14-15-17-19-20-21-22-23-24-25-26.Figure 2a. and Figure 2b.shows the percentage of branch coverage as a function of the fraction of the test suite used, for the two orders T1 and T2, respectively.The area under the curve represents the average of the percentage of branch coverage over the life of the test suite. Figure 2b. shows APBC for Calculator program as 93.27% that reflects what happens when the order of test cases is changed., yielding a "faster coverage" suite than T1 with APBC 84 %.

APSC,APLC and APCC are defined in a similar manner to APBC,except that they measure rate of

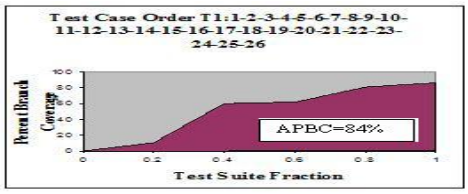

Figure 2a:APBC Test suite T1

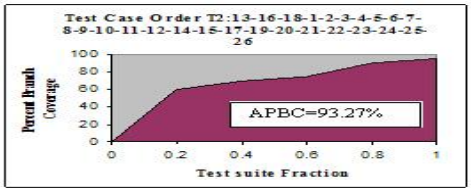

Figure 2b: APBC for prioritized suite T2

coverage of statements, loops and conditions respectively. For Calculator Program we got 94.5% APSC ,93.27% APBC,92.2%APLC and 82.2% APCC.

3.4.2 APFD (Average Percentage of Faults Detected) metric

Depending on the fault criterion considered, APFD metric was computed [3],[9] to measure the rate of fault detection of coverage based prioritization techniques. APFD measures the weighted average of the percentage of faults detected over the life of a test suite. APFD values range from 0 to 100; higher number simply faster (better) fault detection rates. Let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TFi be the first test case in ordering T0 of T which reveals fault i. The APFD for test suite T0 is given by the equation.

$$APFD = 1 - \frac{TF1 + TF2 + \_\_ + TFm}{nm} + \frac{1}{2n}$$

To illustrate this measure, consider the Calculator program with 25 faults and a suite of 26 test cases$1 through 26$ as shown in Table 11.

Consider two orders of these test cases, order T1(No prioritization):1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26 and order T2(stmt_ total prioritization) :8-4-12-6-7-1-2-3-10-11-5-9-13-16-18-14-15-17-19-20-21-22-23-24-25-26.Figure 3a. and Figure 3b. show the percentages of detected faults of the

Table 11: Fault matrix for Calculator program

$TestCase$	Fault								
	1	2	3	4	.	7	.	24	25
1	X								
2	X								
3	X								
4	X	X							
5	X								
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
25						X			
26	X						X		

fraction of the test suite used, for the two orders T1 and T2 respectively. In the graph of APFD the horizontal axis denotes "Test suite Fraction" and the vertical axis denotes "Percentage of detected faults".

The area under the curve represents the average of the percentage of detected faults over the life of the test suite. Figure 3b. Shows APFD for Calculator program as 61.6%. From Figure 3b it is clear that the prioritized order T2 results in the earliest detection of the most faults than the order T1(No prioritization)., with APFD 54.05%.

Similarly we have computed the APFD for the other coverage criterion as shown in Table 12.

Table 12: Prioritization order based on cost for Calculator Program

$PrioritizationTechniques$	$APFD(in\%)$
$NoPrioritization$	54.05
$Stmt\_total$	61.6
$Branch\_total$	57.14
$Loop\_total$	61.44
$Condition\_total$	67.7

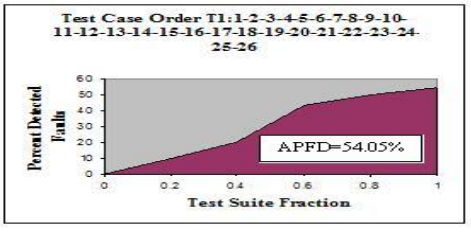

Figure 3a: APFD for Test suite T1

3.4.3 APFDc (Average Percentage Faults Detected with cost) metric

Depending on the cost criterion we have considered in section 3.3.2.1, APFDc metric is com-

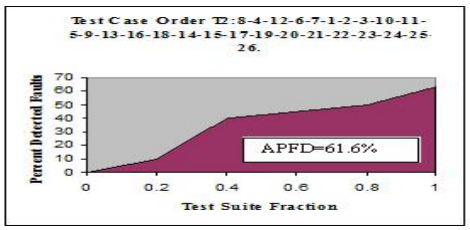

Figure 3b: APFD for Prioritized suite T2

puted [10].Let T be a test suite containing n test cases with costs t1; t2; : : : ; tn. Let F be a set of m faults revealed by T, and let f1; f2; : : : ; fm be the severities of those faults. Let TFi be the first test case in an ordering T' of T that reveals fault i. The (cost-cognizant) weighted average percentage of faults detected during the execution of test suite T'is given by the equation:

$$APFDc = \frac{\sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}\right)\right)}{\sum_{j=1}^{n} t_j \times \sum_{i=1}^{m} f_i}$$

To illustrate this measure, consider the Calculator program with 25 faults and a suite of 26 test cases (1 through 26) as shown in Table 11.

Consider two orders of these test cases, order T1(No prioritization):1-2-3-4-5-6-7-8-9-10-11-12-13-14-5-16-17-1819-20-21-22-23-24-25-26 and order T2(cost oriented stmt_total prioritization) :1-10-5-20-11-18-9-16-2-4-7-19-13-15-6-22-14-26-21-17-23-12-25-8-3-24. Figure 4a. and Figure 4b show the percentages of detected faults of the fraction of the test suite used, for the two orders T1 and T2, respectively. In the graph of APFDc the horizontal axis denotes "Percentage Total Test Case Cost Incurred". Now, each test case in the test suite is represented by an interval along the horizontal axis, with length proportional to the percentage of total test suite cost accounted for by that test case. The vertical axis denotes" Percentage Total Fault Severity Detected". Now, each fault detected by the test suite is represented by an interval along the vertical axis, with height proportional to the percentage of total fault severity for which that fault accounts.

The area under the curve represents the average of the percentage of detected faults over the life of the test suite. Figure 4b. shows APFDc for Calculator program as 84.09%.From Figure 4b. it is clear that the prioritized order T2 results in the earliest detection of the most faults than the order T1(No prioritization)., with APFDc 53.86%.Similarly we have computed the APFDc for the other coverage-cost criterion as shown in Table 13.

Table 13: APFDc metric for Calculator program

$PrioritizationTechniques$	$APFD(in\%)$
$NoPrioritization$	53.86
$Stmt\_total$	84.09
$Branch\_total$	82.48
$Loop\_total$	61.84
$Condition\_total$	70.71

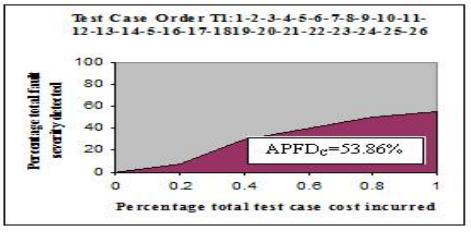

Figure4a.:APFDc for Test suite T1

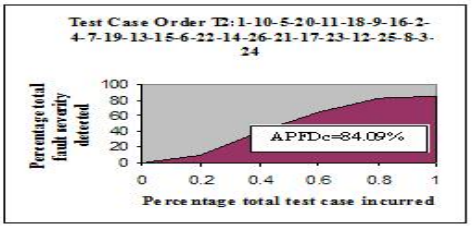

Figure4b: APFDc for Prioritized suite T2

4 Comparison of the various Test case prioritization metrices

We have made an analysis of the various metrices calculated for test case prioritization techniques .We have considered the Calculator program for our analysis with 26 test cases. For Calculator Program we got 94.5% APSC, 93.27% APBC, 92.2%APLC and 82.2% APCC. Table 14 gives the details of the APFD and APFDc metrices for the Calculator program. From the table it is clear that the APFDc metric is higher than the APFD metric for all the prioritization techniques except when No prioritization is done.

5 Experimental Programs used for prioritization

We have considered the following java programs for our experimental study viz Payroll calculation, Calculator program and two consecutive versions

Table 14: APFD and APFDc calculation for Calculator Program

$Prioritization\ Techniques$	$APFD\ (in\%)$	$APFD_c\ (in\%)$
$NoPrioritization$	54.05	53.86
$Stmt\_total$	61.6	84.09
$Branch\_total$	57.14	82.48
$Loop\_total$	61.44	61.84
$Condition\_total$	67.7	70.71

of Arrays program of java Utility package (jdk1.4and jdk1.5). The Payroll program computes the pay and loan details of employees and in the Calculator program we have designed a scientific calculator that calculates all arithmetic, logical, trignometric operations. The Arrays program from the java Utility package (jdk1.4) performs Sorting, searching, filling etc. and the Arrays program from the java Utility package (jdk1.5) performs Sorting, filling, hashing etc.The details of above programs are shown in Table 15 as follows.

Table 15: Experimental program detailsLOC- Lines of code, NOM-No.of Methods,NOC-No.of classes, CC-CyclomaticComplexity

$Program$	LOC	NOC	NOM	CC	$Noof\ TestCase$
$Payroll$	320	2	10	27	29
$Calculator$	536	1	15	55	58
$Arraysof\ Jdk1.4$	1010	1	70	188	203
$Arraysof\ Jdk1.5$	1360	1	79	250	255

In the table 15, Cyclomatic complexity provides an upper bound for the number of test cases that are to be written for complete testing of the application. Test cases were written and executed using JUnit framework.

6 Result Analysis

We have performed the analysis for the various mentioned programs and we have executed the various test cases using JUnit framework.We have written 29 test cases for payroll program,26 test cases for calculator program,203 test cases for jdk1.4 and 255 test cases for jdk1.5We found coverage information using Code Cover tool.[14] for all the programs namely Statement coverage(SC), branch coverage(BC), loop coverage(LC) and condition coverage(CC) and prioritized the test cases using the techniques mentioned in Table 2..The total coverage for each program is as shown in Table 16. To measure the effectiveness of coverage based prioritization techniques we have calculated the various metrices viz. APSC, APBC, APLC and APCC.The details are in Table 17. To measure how quickly the prioritized test

Table 16: Coverage information provided by Code cover

$Program$	$SC\ (\%)$	$BC\ (\%)$	$LC\ (\%)$	$CC\ (\%)$
$Payroll$	95	90	67	88
$Calculator$	99	96	70	100
$Arraysof Jdk1.4$	94	80	69	70
$Arraysof Jdk1.5$	95	82	70	71

Table 17: Calculation of coverage based metrices

$Program$	Coverage Matrix (in %)			
	$APSC$	$APBC$	$APLC$	$APCC$
$Payroll$	95.1	91.2	90.1	80.1
$Calculator$	94.5	93.27	92.2	82.2
$Arraysof Jdk1.4$	95.2	93.3	91.2	91.3
$Arraysof Jdk1.5$	95.5	94.6	93.6	93.5

suite detects faults, we have injected faults using MuJava tool[16]. We have generated mutants using mutation operators as mentioned in Table 5.From the faults injected, we created fault matrix which gives the details of which test case detected what fault. From this information we calculated the metric APFD for coverage based test case prioritization techniques.APFD computation is used to measure the rate of fault detection for each prioritized test suite for each mutant group on each version. The collected score were analyzed to determine whether techniques improved the rate of fault detection. To perform Cost oriented Test case prioritization, we have prioritized test cases using prioritization techniques as shown in Table 2 by combining cost and coverage measurement using Algorithm 1 as described in section 3.3.2.We have estimated cost by measuring the actual time required to execute each test case . We considered module criticality for estimating fault severity. We estimated the fault severity based on severity values shown in Table 7.Total severity and average severity (criticality$_t$) for each test case were calculated. Then Award value is calculated .Test case which has high award value is executed first and so on. Then effectiveness of cost oriented prioritization techniques are measured using APFDc metric.Table 18 gives the complete set of metrices calculated for proving the effectiveness of various test case prioritization techniques.

From the table it is clear that the average percentage of fault detection is higher since we have the varying cost factor and considering the severity of faults.In real time applications the cost factors and the severity of faults varies always and hence APFDc is considered to be a better metric compared to APFD.

Programs	Coverage Metrics(in %)				APFD (in %)					APFDc (in %)				
	APSC	APBC	APLC	APCC	No prioritization	Stmt_total	Branch_total	loop_total	condition_total	No prioritization	Stmt_total	Branch_total	loop_total	condition_total
Payroll	95.1	91.2	90.1	80.1	80.35	87.52	82.7	81.45	82.41	76.82	94.87	86.75	80.84	84.92
Calculator	94.5	93.27	92.2	82.2	54.05	61.6	57.14	61.44	67.7	**53.86**	**84.09**	**82.48**	**61.84**	**70.71**
Arrays of jdk1.4	95.2	93.3	91.2	91.3	60.56	75.3	74.2	74.1	73.2	70.56	86.52	84.23	79.32	83.5
Arrays of jdk1.5	95.5	94.6	93.6	93.5	70.5	85.5	81.5	82.1	81.2	74.2	90.5	89.2	88.3	89.4

Table 18: Calculated metrices for the selected experimental programs

7 Conclusion and future work

We have performed coverage and cost based prioritization and examined the effectiveness of prioritization techniques in terms of coverage and rate of fault detection for some standard java programs. We considered the mutation faults to increase the test suite's rate of fault detection. From the study it is shown that Prioritization improves the rate of fault detection. In real time applications the cost factors and the severity of faults varies always and hence APFDc is considered to be a better metric compared to APFD.Thus to improve the rate of fault detection we have computed a number of metrices that predicts the quality and effectiveness of the various prioritization techniques required for regression testing. These metrices also helps the Test Manager in selecting a better prioritization Technique there by reducing the time, cost and resources during regression testing.

As a future work, we intend to apply requirement changes for performing the prioritization techniques and also considering varied cost factors like hardware cost, wages, cost of materials required for testing, earnings lost due to delays in failing to meet target release dates, and so on.

References

[1]. S. Elbaum, A. Malishevsky, and G. Rothermel, Prioritizing Test Cases for Regression Testing, Proc. Int'l Symp. Software Testing and Analysis, pp. 102-112, Aug. 2000.

[2]. S. Elbaum, A. Malishevsky, and G. Rothermel, Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization,Proc. Int'l Conf. Software Eng., pp. 329-338, May 2001.

[3] S. Elbaum, A.G. Malishevsky, and G. Rothermel, Test Case Prioritization: A Family of Empirical Studies, IEEE Trans.Software Eng., vol. 28, no. 2, pp. 159-182, Feb. 2002.

[4] J. Kim and A. Porter, A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments, Proc. Int'l Conf. Software Eng., pp. 119-129, May2002.

[5] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma,Regression Testing in an Industrial Environment, Comm.ACM, vol. 41, no. 5, pp. 81-86, May 1988.

[6] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, Prioritizing Test Cases for Regression Testing, IEEE Trans. Software Eng.,vol. 27, no. 10, pp. 929-948, Oct. 2001.

[7] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, A Study of Effective Regression Testing in Practice, Proc. Int'l Symp. Software Reliability Eng., pp. 230-238, Nov. 1997.

[8]J.-M. Kim, A. Porter, and G. Rothermel, An Empirical Study of Regression Test Application Frequency, Proc. Int'l Conf. SoftwareEng., pp. 126-135, June 2000.

[9] H. Do, G. Rothermel, On the use of Mutation faults in Empirical Assessments of Test case prioritization Techniques, IEEE Trans. Software Eng., vol. 32, no. 9, Sept. 2006

[10] Alexey G. Malishevsky_ Joseph R. Ruthru_y Gregg Rothermely Sebastian Elbaumy, Cost-cognizant Test Case Prioritization, Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering,University of NebraskaLincoln, Lincoln, Nebraska, U.S.A., 12 March 2006

[11]Zheng Li,Mark Harman,and Robert M.Hierons,Search Algorithms for Regression Test Case Prioritization, IEEE Trans. Software Eng.,vol. 33, no. 4, pp. 225-237, Apr. 2007.

[12]D. Binkley.Semantics guided regression test cost reduction. IEEE Transactions on Software Engineering,23(8):498-516, August 1997.

[13] Hyunsook Do, Gregg Rothermel, Alex Kinneer,Empirical Studies of Test Case Prioritization in a JUnit Testing Environment, Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)1071-9458,2004 IEEE

[14] www.codecover.org

[15] www.junit.org

[16] www.mujava.org

[17]L.Shanmugapriya.A.Malini,A.Askarunisha, Analysis of Java Based Coverage Testing Tools,IEEE International Advance Computing Conference,March2009.