Journal of Information Processing Systems, Vol.6, No.2, June 2010 DOI : 10.3745/JIPS.2010.6.2.235

A Regression Test Selection and Prioritization Technique

Ruchika Malhotra*, Arvinder Kaur* and Yogesh Singh*

Abstract—Regression testing is a very costly process performed primarily as a software maintenance activity. It is the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications. A regression test selection technique selects an appropriate number of test cases from a test suite that might expose a fault in the modified program. In this paper, we propose both a regression test selection and prioritization technique. We implemented our regression test selection technique and demonstrated in two case studies that our technique is effective regarding selecting and prioritizing test cases. The results show that our technique may significantly reduce the number of test cases and thus the cost and resources for performing regression testing on modified software.

Keywords—Regression Testing, Maintenance, Prioritization

1. INTRODUCTION

Software maintenance is becoming important and expensive day by day [1]. When the software is modified during maintenance phases, retesting is performed. This process of retesting the software is known as regression testing. Regression testing helps in increasing confidence as to the stability of the modified program by locating errors in the modified program, and ensuring the continued operation of the software. Regression testing is a very costly process and consumes significant amounts of resources.

During regression testing, an already designed test suite is available for reuse. A regression test selection technique may help us to select an appropriate number of test cases from this test suite. The simplest technique is to run all test cases for verifying the modified program. This is the safest technique, but it is practical only when the size of test suite is small. We may select test cases randomly to reduce the size of the test suite. Many test cases selected randomly may not have any relation with the modified program. Another technique suggests the selection of test cases that execute the modified portion of the program and the portions that are affected by these modifications. These test cases are known as modification revealing test cases. All those test cases that reveal faults in the modified program are known as fault revealing test cases. Unfortunately, we do not have any efficient technique to find fault revealing and modification revealing test cases. We may also indicate the precedence with which a test case may be addressed during regression testing. A test case with higher rank will have higher priority than a test case

Manuscript received April 12, 2010; accepted May 11, 2010.
Corresponding Author: Ruchika Malhotra
* University School of Information Technology, Guru Gobind Singh Indraprastha University, Kashmere Gate, Delhi-110403, India (ruchikamalhotra2004@yahoo.com, arvinderkaurtakkar@yahoo.com, ys66@rediffmail.com)

with lower rank.

This work is the extension of earlier regression test selection and prioritization techniques [2]. We implemented this technique and validated this technique with the help of two case studies. Unlike other techniques, our technique identifies test cases that execute the modified lines of source code at least once and selects those test cases that execute the lines of source code after deletion of lines from the execution history of the test cases The results show that the technique can significantly reduce the cost and resources for performing regression testing on modified programs.

This paper is organized as follows: The related work is summarized in Section 2. Section 3 provides background for the proposed technique.

The detailed algorithm for the proposed technique along with two case studies is given in section 4. Section 5 presents the application of the technique and the conclusions of the research are presented in section 6.

2. Related Work

Fischer et al. proposed a minimization based regression test selection technique. This technique used linear equations in order to represent relationships between basic block and test cases [3].

A safe regression test selection algorithm was proposed by Rothermal and Harrold [4]. They used control flow graphs for a program or procedure and these graphs were used to select test cases that execute modified source code from the given test suite. Wong et al. carried out a study of regression testing [5]. Chen et al. [6] and Laski and Szermer, Vokolos and Frankl [7] have also proposed safe regression test selection techniques. A hybrid technique was proposed by Wong in 1994 [5].

Two complimentary algorithms were given by [8]. Harrold and Soffa proposed a data flow coverage based regression test selection technique [9]. An empirical study was conducted by Graves et al. in order to examine the costs and benefits of various regression test selection techniques [10]. Rothermal et al. analyzed various test selection algorithms in [11]. The issues related to prioritization were addressed by Rothermal et al. [12]. Rothermal et al. described the prioritization of test cases in large software development environments [13]. Kim02 [14] proposed a prioritization technique based on historical execution data. Li07 [15] performed a empirical study using several greedy algorithms.

3. Background

Here we present the concept and types of prioritization. We also provide the basic notations used in the proposed technique in the rest of the paper.

3.1 Prioritization Criteria

The efficiency of the regression testing is dependent upon the criteria of prioritization. There are two varieties of test case prioritization viz. general test case prioritization and version specific test case prioritization. In general test case prioritization, for a given program with its test suite, we prioritize the test cases that will be useful over a succession of subsequent modified versions of the original program without any knowledge of modification(s). In version specific test case prioritization, we prioritize the test cases i.e., when the original program is changed to the modified program, versus the knowledge of the changes that have been made in the original program.

3.2 Test cases selection criteria

We consider a program P with its modified program P′ and its test suite T created to test P. When we modify P to P′, we would like to execute modified portion(s) of the source code and the portion(s) affected by the modification(s) to see the correctness of modification(s). We neither have time nor resources to execute all test cases of T. Our objective is to reduce the size of T to T′ using some selection criteria, which may help us to execute tests on the modified portion of the source code and the portion(s) affected by modification(s).

The technique is based on version specific test case prioritization where information about changes in the program is known. Hence, prioritization is focused around the changes in the modified program. We may like to execute all modified lines of source code with a minimum number of selected test cases. This technique identifies those test cases that:

(i) Execute the modified lines of source code at least once
(ii) Execute the lines of source code after deletion of deleted lines from the execution history of the test case and that are not redundant.

The technique uses two algorithms one for "modification" and the other for "deletion". The following information is available from us and has been used to design the technique:

(iii) Program P with its modified program P′.
(iv) Test suite T with test cases t1, t2, t3,…..,tn.
(v) Execution history (number of lines of source code covered by a test case) of each test case of test suite T.
(vi) Line numbers of lines of source code covered by each test case are stored in two dimensional array (t_{11}, t_{12}, t_{13},……,t_{ij}).

4. Regression Test Selection and Prioritization Technique

We propose a regression test selection and prioritization technique, which prioritizes test cases in test suite T and selects from test suite T a subset T′. The technique also prioritizes test cases of T′ and recommends using high priority test cases first and then low priority test cases and so on until time and resources are available or a reasonable level of confidence about correctness is achieved.

4.1 Modification algorithm

The "modification" portion of the technique is used to minimize and prioritize test cases based on the modified lines of source code. The "modification" algorithm uses the following information given in table 1.

Table 1. Variables used by “modification” algorithm

S.No	Variable name	Description
1	T1	It is a two dimensional array and is used to store line numbers of lines of source code covered by each test case.
2	Modloc	It is used to store the total number of modified lines of source code.
3	mod_locode	It is a one dimensional array and is used to store line numbers of modified lines of source code
4	Nfound	It is a one dimensional array and is used to store a number of lines of source code matched with modified lines of each test case.
5	Pos	It is a one dimensional array and is used to set the position of each test case when nfound is sorted.
6	Candidate	It is a one dimensional array. It sets the bit to 1 corresponding to the position of the test case to be removed.
7	Priority	It is a one dimensional array and is used to set the priority of the selected test case.

The following steps have been followed in order to select and prioritize test cases from test suite T based on the modification in the program P.

Step I: Initialization of variables

Consider a program for classification of a triangle of 42 lines of code with a test suite of 13 test cases. Its input is a triple of positive integers (say a, b, c). The program output may have one of the following words:

[Acute angled triangle, Obtuse angled triangle, Right angled triangle, Invalid triangle] Test cases are generated using data flow testing technique. Data flow testing focuses on variable definition and variable usage. The variables are defined and used (referenced) throughout the program. Hence, this technique concentrates on how a variable is defined and used at different places of the program. The execution history is given in table 2 (from definition to its usage). Table 2 also shows the inputs given and the expected output from the program. We assume that lines 5, 8, 10, 15, 20, 23, 28, 35 are modified.

Table 2. Test cases with execution history

Test case Id	A	B	C	Expected output	Execution history
T1	30	20	40	Obtuse angled triangle	8, 9, 10, 11, 12, 13
T2	30	20	40	Obtuse angled triangle	8, 9, 10, 11, 12, 13, 14, 15, 16 , 20, 21, 22
T3	30	20	40	Obtuse angled triangle	10, 11, 12, 13
T4	30	20	40	Obtuse angled triangle	10, 11, 12, 13, 14, 15, 16, 20, 21, 22
T5	30	20	40	Obtuse angled triangle	12, 13, 14, 15, 16, 20, 21, 22
T6	30	40	50	Right angled triangle	22, 23, 24, 25, 28
T7	30	20	40	Obtuse angled triangle	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21
T8	-	-	-	-	15, 16, 20, 21, 35
T9	30	10	15	Invalid triangle	5, 6, 7, 8, 9, 10, 11, 12, 14, 17, 18, 19, 20, 21
T10	30	10	15	Invalid triangle	18, 19, 20, 21, 35
T11	30	20	40	Obtuse angled triangle	24, 25
T12	30	20	40	Obtuse angled triangle	15, 16, 20, 21

The first portion of the "modification" algorithm is used to initialize and read values of variables T1, modloc, and mod_locode.

First portion of the "modification" algorithm

```
1. Repeat for i=1 to number of test cases
     a. Repeat for j=1 to number of test cases
          i. Initialize array T1[i][j] to zero
2. Repeat for i=1 to number of test cases
     a. Repeat for j=1 to number of test cases
          i. Store line numbers of line of source code covered by each test case.
3. Repeat for i=1 to number of modified lines of source code
     a. Store line numbers of modified lines of source code in array mod_locode.
```

Step II: Selection and prioritization of test cases

The second portion of the algorithm counts the number of modified lines of source code covered by each test case (nfound).

Second portion of the "modification" algorithm

```
2. Repeat for all true cases
     a. Repeat for i=0 to number of test cases
          i. Initialize array nfound[i] to zeroes
          ii. Set pos[i] =i
     b. Repeat for i=0 to number of test cases
          i. Initialize l to zero
          ii. If candidate[i] ≠ 1 then
               Repeat for k=0 to modified lines of source code
                    If t1[i][j]=mod_locode[k] then
                         Increment nfound[i] by one
                         Increment l by one
```

The status of test cases covering modified lines of source code is given in table 3.

Table 3. Test cases with Number of Matches Found

Test Cases	**Line Nos. of lines matched**	**No. of Matches (nfound)**
T1	8, 10	2
T2	8, 10, 15, 20	4
T3	10	1
T4	10, 15, 20	3
T5	15, 20	2
T6	23, 28	2
T7	5, 8, 10, 15, 20	5
T8	15, 20, 35	3
T9	5, 8, 10, 20	4
T10	20, 35	2
T11	-	0
T12	15, 20	2

Consider the third portion of “modification” algorithm. In this portion, we sort the nfound array and select the test case with the highest value of nfound as the candidate for selection. The test cases are arranged with an increasing order of priorities.

Third portion of the “modification” algorithm

```
d.  Initialize l to zero
e.  Repeat for i=0 to number of test cases
        i.   Repeat for j=0 to number of test cases
               If nfound[i]>nfound[j] then
                   t=nfound[i]
                   nfound[i]=nfound[j]
                   nfound[j]=t
                   t=pos[i]
                   pos[i]=pos[j]
                   pos[j]=t
f.  Repeat for i=0 to number of test cases
        i.   If nfound[i]=1 then
               Increment count
g.  If count = 0 then
        i.   Goto end of the algorithm
h.  Initialize candidate[pos[0]] = 1
i.  Initialize priority[pos[0]]= m+1
```

The test cases with less value have higher priority than the test cases with higher value. Hence, the test cases are sorted on the basis of number of modified lines covered as shown in table 4.

Table 4. Test cases in decreasing order of number of modified lines covered

Test Cases	Line Nos. of lines matched	No. of Matches (nfound)	Candidate	Priority
T7	5, 8, 10, 15, 20	5	1	1
T2	8, 10, 15, 20	4	0	0
T9	5, 8, 10, 20	4	0	0
T4	10, 15, 20	3	0	0
T8	15, 20, 35	3	0	0
T1	8, 10	2	0	0
T5	15, 20	2	0	0
T6	23, 28	2	0	0
T10	20, 35	2	0	0
T12	15, 20	2	0	0
T3	10	1	0	0
T11	-	0	0	0

The test case with candidate=1 is selected in each iteration. In the fourth portion of the algorithm, the modified lines of source code included in the selected test cases are removed from the mod_locode array. This process continues until there are no remaining modified lines of source code covered by any test case.

Fourth portion of the “modification” algorithm

```
i.    Repeat for i=0 to length of selected test cases
        i.  Repeat for j=0 to modified lines of source code
              If t1[pos[0]][i] = mod[j] then
              mod[j] = 0
```

Since test case T7 is selected and it covers 1 and 2 lines of source code, these lines will be removed from the mod_locode array.

mod_locode = [5, 8, 10, 15, 20, 23, 28, 35] - [5, 8, 10, 15, 20] = [23, 28, 35]

The remaining iterations of the “modification” algorithm are shown in tables 5-6.

Table 5. Test cases in Descending Order of Number of Matches found (iteration 2)

Test Cases	No. of matches (nfound)	Matches found	Candidate	Priority
T6	2	23, 28	1	2
T8	1	35	0	0
T10	1	35	0	0
T1	0	-	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T5	0	-	0	0
T9	0	-	0	0
T11	0	-	0	0
T12	0	-	0	0

mod_locode = [23, 28, 35] – [23, 28] = [35]

Table 6. Test cases in Descending Order of Number of Matches found (iteration 3)

Test Cases	No. of matches (nfound)	Matches found	Candidate	Priority
T8	1	35	1	3
T10	1	35	0	0
T1	0	-	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T5	0	-	0	0
T9	0	-	0	0
T11	0	-	0	0
T12	0	-	0	0

mod_locode = [35] – [35] = [Nil]

Hence test cases T7, T6, and T8 need to be executed on the basis of their corresponding priority (see figure 1). Out of 12 test cases, we need to run only 3 test cases for 100% code coverage of modified lines of source code. This is a 75% reduction of test cases.

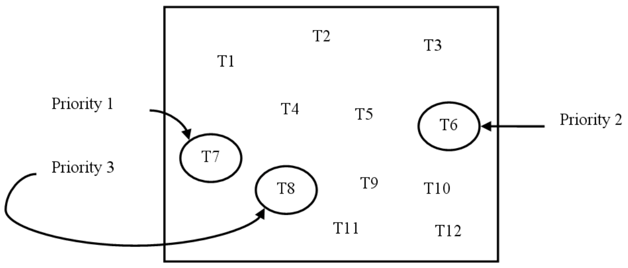

Fig. 1. Test case selection and prioritization

4.2 Deletion algorithm

The "deletion" portion of the technique is used to (i) update the execution history of test cases by removing the deleted lines of source code (ii) identify and remove those test cases that cover only those lines which are covered by other test cases of the program. The information used in the algorithm is given in table 7.

Table 7. Variables used by "modification" algorithm

S.No	Variable	Description
1	T1	It is a two dimensional array. It keeps the number of lines of source code covered by each test case i.
2	deloc	It is used to store total number of lines of source code deleted.
3	del_locode	It is a one dimensional array and is used to store line numbers of deleted lines of source code.
4	count	It is a two dimensional array. It sets the position corresponding to every matched line of source code of each test case to 1
5	match	It is a one dimensional array. It stores the total count of the number of 1's in count array for each test case.
6	deleted	It is a one dimensional array. It keeps the record of redundant test cases. If the value corresponding to test case i is 1 in a deleted array, then that test case is redundant and should be removed.

Step I: Initialization of variables

We consider a program for determination of day in a week of 118 lines of source code with a test suite of 12 test cases. Its input is a triple of day, month and year with the values in the range. The possible outputs would be the day of the week or an invalid date. The execution history (paths covered by using data flow testing technique) is given in table 8.

We assume that lines numbers 6, 28, 36, 44, 50 and 61 are modified, and line numbers 12, 55 and 27 are deleted from the source code. The information is stored as:

delloc = 3
del_locode = [12, 27, 55]
modloc = 6
mod_locode = [6, 28, 36, 44, 50, 61]

First portion of the "deletion" algorithm

```
1.  Repeat for i=1 to number of test cases
    a.  Repeat for j=1 to length of test case i
        i.  Repeat for l to number of deleted lines of source code
              If T1[i][j]=del_locode then
                  Repeat for k=j to length of test case i
                      T1[i][k]=T1[i][k+1]
                  Initialize T1[i][k] to zero
                  Decrement c[i] by one
```

Table 8. Test cases with execution history

Test case Id	Month	Day	Year	Expected output	Execution history
T1	6	15	1900	Friday	6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
T2	1	15	1900	Monday	46, 47, 48, 53, 54, 55, 56, 57, 61, 91
T3	1	15	2009	Thursday	50, 51, 52, 53, 54, 55, 56, 57, 61, 91
T4	1	15	2009	Thursday	56, 57, 61, 91
T5	2	15	2000	Tuesday	67, 68, 69, 91
T6	4	15	2009	Wednesday	74, 75, 91
T7	7	15	2009	Wednesday	89, 90, 91
T8	6	15	1900	Friday	3, 4, 5, 6, 7, 8, 9, 10, 11, 44
T9	1	15	1900	Monday	15, 16, 17, 18, 26, 37, 38, 39, 43, 44, 45, 46, 47, 48, 53, 54, 55
T10	2	15	2000	Tuesday	28, 29, 36, 43, 44
T11	2	30	2009	Invalid Date	34, 35, 36, 43, 44
T12	2	15	1900	Thursday	13, 14, 15, 16, 17, 18, 26, 27

After deleting line numbers 12, 27 and 55 the modified execution history is given in table 9.

Table 9. Modified execution history after deleting line numbers 12, 27, 55

Test case Id	Execution history
T1	6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19
T2	46, 47, 48, 53, 54, 56, 57, 61, 91
T3	50, 51, 52, 53, 54, 56, 57, 61, 91
T4	56, 57, 61, 91
T5	67, 68, 69, 91
T6	74, 75, 91
T7	89, 90, 91
T8	3, 4, 5, 6, 7, 8, 9, 10, 11, 44
T9	15, 16, 17, 18, 26, 37, 38, 39, 43, 44, 45, 46, 47, 48, 53, 54
T10	28, 29, 36, 43, 44
T11	34, 35, 36, 43, 44
T12	13, 14, 15, 16, 17, 18, 26

Step II: Identification of redundant test cases

We want to find redundant test cases. A test case is a redundant test case, if it covers only those lines which are covered by other test cases of the program. This situation may arise due to deletion of few lines of the program.

Consider the second portion of the "deletion" algorithm. In this portion, the test case array is initialized with line numbers of lines of source code covered by each test case.

Second portion of the "deletion" algorithm

```
2.  Repeat for i=1 to number of test cases
    a.  Repeat for j=1 to number of test cases
            i.  Initialize array t1[i][j] to zero
           ii.  Initialize array count[i][j] to zero
3.  Repeat for i=1 to number of test cases
    a.  Initialize deleted[i] and match [i] to zero
4.  Repeat for i=1 to number of test cases
    a.  Initialize c[i] to number of line numbers in each test case i
    b.  Repeat for j=1 to c[i]
    c.  Initialize t1[i][j] to line numbers of line of source code covered by each test case
```

The third portion of the algorithm compares lines covered by each test case with lines covered by other test cases. A two dimensional array count is used to keep the record of line number matched in each test case. If all the lines covered by a test case are being covered by some other test case, then that test case is redundant and should not be selected for execution.

Third portion of the "deletion" algorithm

```
5.  Repeat for i=1 to number of test cases
    a.  Repeat for j=1 to number of test cases
            i.  If i≠j and deleted[j]≠1 then
                    Repeat for k=1 to until t1[i][k]≠0
                        Repeat for l=1 until t1[j][l]≠0
                            If t1[i][k]=t1[j][l] then
                                Initialize count [i][k]=1
    b.  Repeat for m=1 to c[i]
            i.  If count[i][m]=1 then
                    Increment match[i] with 1
    c.  If match[i]=c[i] then
            i.  Initialize deleted[i] to 1
6.  Repeat for i=1 to number of test cases
    a.  If deleted[i] =1 then
            i.  Remove test case i (as it is a redundant test case)
```

On comparing all values in each test case with all values of other test cases, we found that test case 2, test case 4 and test case 12 are redundant test cases. These three test cases do not cover any line which is not covered by other test cases as shown in table 10.

Table 10. Redundant test cases

Test Case	Line Number of LOC	Found In Test Case	Redundant Y/N
T2	46	T9	Y
	47	T9	Y
	48	T9	Y

Test Case	Line Number of LOC	Found In Test Case	Redundant Y/N
	53	T3	Y
	54	T3	Y
	56	T3	Y
	57	T3	Y
	61	T3	Y
	91	T3	Y
T4	56	T3	Y
	57	T3	Y
	61	T3	Y
	91	T3	Y
T12	13	T1	Y
	14	T1	Y
	15	T1	Y
	16	T1	Y
	17	T1	Y
	18	T1	Y
	26	T9	Y

The remaining test cases are = [T1, T3, T5, T6, T7, T8, T9, T10, T11] and are given in table 11.

Table 11. Modified table after removing T2, T4 and T12

Test case Id	Execution history
T1	6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19
T2	46, 47, 48, 53, 54, 56, 57, 61, 91
T3	50, 51, 52, 53, 54, 56, 57, 61, 91
T5	67, 68, 69, 91
T6	74, 75, 91
T7	89, 90, 91
T8	3, 4, 5, 6, 7, 8, 9, 10, 11, 44
T9	15, 16, 17, 18, 26, 37, 38, 39, 43, 44, 45, 46, 47, 48, 53, 54
T10	28, 29, 36, 43, 44
T11	34, 35, 36, 43, 44

Now we will minimize and prioritize test case using “modification” algorithm given in section 4.1. The status of test cases covering the modified lines is given in table 12.

Table 12. Test cases with Modified Lines

Test Cases	Line Nos. of lines matched (found)	No. of matches (nfound)
T10	3	28, 36, 44
T3	2	50, 61
T11	2	36, 44
T1	1	6
T8	1	44
T9	1	44
T5	0	-
T6	0	-
T7	0	-

Test cases are sorted on the basis of no. of modified lines covered as shown in tables 13-15.

Table 13. Test cases in Descending Order of Number of Modified lines covered

Test Cases	Line Nos. of lines matched (found)	No. of matches (nfound)	Candidate	Priority
T10	3	28, 36, 44	1	1
T3	2	50, 61	0	0
T11	2	36, 44	0	0
T1	1	6	0	0
T8	1	44	0	0
T9	1	44	0	0
T5	0	-	0	0
T6	0	-	0	0
T7	0	-	0	0

mod_locode = [6, 8, 36, 44, 50, 61] – [28, 36, 44] = [6, 50, 61]

Table 14. Test cases in Descending Order of Number of Modified lines covered (iteration 2)

Test Cases	Line Nos. of lines matched (found)	No. of matches (nfound)	Candidate	Priority
T10	3	28, 36, 44	1	1
T3	2	50, 61	0	0
T11	2	36, 44	0	0
T1	1	6	0	0
T8	1	44	0	0
T9	1	44	0	0
T5	0	-	0	0
T6	0	-	0	0
T7	0	-	0	0

mod_locode = [6, 50, 61] – [50, 61] = [6]

Table 15. Test cases in Descending Order of Number of Modified lines covered (iteration 3)

Test Cases	Line Nos. of lines matched (found)	No. of matches (nfound)	Candidate	Priority
T11	1	6	1	3
T1	0	-	0	0
T8	0	-	0	0
T9	0	-	0	0
T5	0	-	0	0
T6	0	-	0	0
T7	0	-	0	0

Hence, test cases T10, T3, and T1 are need to be executed and redundant test cases are T2, T4 and T12 (as shown in figure 2).

Out of the five test cases, we need to run only 3 test cases for 100% code coverage of modified code coverage. This is 75% reduction. If we run only those test cases that are covering any

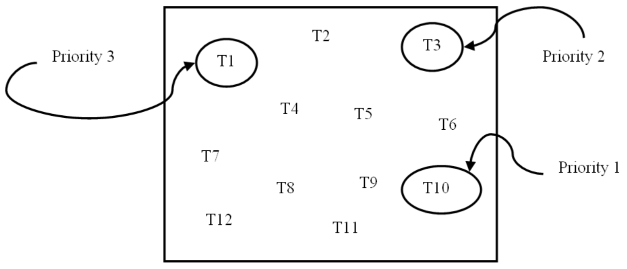

Fig. 2. Test cases selected and prioritized

modified lines, then T10, T3, T1 are selected. This technique not only selects test cases, but also prioritizes test cases. The source code of the proposed technique is given in Appendix.

5. APPLICATIONS

Test case selection and prioritization is essential for maintaining software. Every developer/tester faces this challenge in every organization. In the absence of any effective technique, the random selection of test cases may prevail and the outcome regarding the correction of the program may be illusive and sometimes becomes incorrect. The impact analysis of the changes to the program may further become difficult and time consuming. Hence, an effective technique not only reduces maintenance effort but also performs the desired impact analysis properly. Moreover, such a technique becomes the focus of maintenance activities and may help to preserve the quality of the software. The proposed regression test selection and prioritization technique can reduce the cost and time of regression testing and thereby reduce the cost of maintenance activity. Adequate regression testing will also ensure the quality and reliability of the modified software.

6. CONCLUSIONS

The goal of our work is to select and prioritize test cases for performing regression testing activity. In this work, we implement and validate a regression test selection and prioritization technique. The work is important due to the following reasons:

- The proposed technique increases confidence in the correctness of the modified program.
- The test cases selected using the proposed technique will identify and locate errors in the modified program.
- The proposed technique will help in preserving the quality and reliability of the software.
- Test cases selected by the proposed technique will ensure the software's continued operation.

The results show that the proposed regression selection and prioritization technique will help in reducing the test cases by a significant number. Therefore, the software developers and testers can use this technique in practice and this technique can reduce the cost of regression testing significantly. In future we will analyze the proposed algorithm on large programs.

REFERENCES

[1] B. Beizer, “Software Testing Techniques,” Van Nostrand Reinhold, New York, 1990.

[1] K. K. Aggarwal, Yogesh Singh, and Arvinder Kaur, “Code Coverage Based Technique for Prioritizing Test Cases for Regression Testing,” ACM SIGSOFT, Vol.29, No.5, pp.1-4, 2004.

[2] K. Fischer, F. Raji, and A. Chruscicki, “A methodology for retesting modified software,” In Proceedings of the National Telecommunications Conference B-6-3, pp.1-6, Nov., 1981.

[3] G. Rothermel and M. Harrold, “A Safe, Efficient Algorithm for Regression Test Selection,” Proceedings of International Conference on Software Maintenance,” pp.358-367, 1993.

[4] W. E. Wong, J. R. Horgan, S. London and H. Aggarwal, “A Study of Effective Regression in Practice,” Proceedings of the 8th International Symposium on software reliability Engineering, pp.230-238, Nov., 1994.

[5] Y. Chen, D. Rosenblum, and K. Vo, “Test Tube, A system for selective regression testing,” In Proceedings of the 16th International Conference on Software Engineering, 211-220, May, 1994.

[6] J. Laski and W. Szermer, “Identification of program modifications and its applications in software maintenance,” In Proceedings of the 1992 Conference on Software Maintenance (Nov.). pp.282-290, 1992.

[7] D. Binkley, “Semantics Guided Regression Test Cost Reduction,” IEEE Transactions on Software Engineering, Vol.23, No.8, pp.498-515, 1997.

[8] M.J Harrold, and M.L Soffa, “An incremental approach to unit testing during maintenance”, In Proceedings of the Conference on Software Maintenance (Oct.). pp.362-367, 1998.

[9] T. Graves, M.J. Harrold, J.M. Kim, A. Porter, and G. Rothermel, “An empirical study of regression test selection techniques,” Proceedings 20th International Conference on Software Engineering, Kyoto, Japan. IEEE Computer Society Press: Los Alamitos, CA, pp.188–197, 1998.

[10] G. Rothermel and M. Harrold, “Analysing Regression Test Selection Techniques,” IEEE Transactions on Software Engineering, Vol.22, No.8, pp.529-551, 1996.

[11] G. Rothermel, R.H. Untch, C. Chu and M.J. Harold, “Test Case Prioritization,” IEEE Transactions on Software Engineering, Vol.27, No.10, pp.928-948, Oct., 2001.

[12] A. Srivastava and J. Thiagarajan, “Effectively Prioritizing Tests in Development Environment,” Proceedings of the International Symposium of Software Testing and Analysis, Rome, 22-24 pp.97-106, July, 2002.

[13] Kim, J. M., and A. Porter, “A history-based test prioritization technique for regression testing in resource constrained environments,” In Proceedings of the 24th International Conference on Software Engineering, pp.119-129, 2002.

[14] Z. Li, M. Harman, and R. M. Hierons “Search algorithms for regression test case prioritization,” IEEE Trans. On Software Engineering, Vol.33, No.4, April, 2007.

Appendix

```
#include<stdio.h>
#include<conio.h>
void main()
{
int t1[50][50]={0};
int count[50][50]={0};
int deleted[50],deloc,del_loc[50],k,c[50],l,num,m,n,match[50],i,j;
clrscr();
for(i=0;i<50;i++){
deleted[i]=0;
match[i]=0;
}
printf("Enter the number of test cases\n");
scanf("%d",&num);
for(i=0;i<num;i++){
  printf("Enter the length of test case %d\n",i+1);
  scanf("%d",&c[i]);
  printf("Enter the values of test case\n");
  for(j=0;j<c[i];j++){
            scanf("%d",&t1[i][j]);
            }
  }

printf("\nEnter the deleted lines of code:");
scanf("%d",&deloc);

for(i=0;i<deloc;i++)
  {
            scanf("%d",&del_loc[i]);
  }
for(i=0;i<num;i++){
  for(j=0;j<c[i];j++){
            for(l=0;l<deloc;l++){
            if(t1[i][j]==del_loc[l]){
                        for(k=j;k<c[i];k++){
                        t1[i][k]=t1[i][k+1];
                        }
                        t1[i][k]=0;
                        c[i]--;
            }
  }
  }
}
printf("Test case execution history after deletion:\n");
for(i=0;i<num;i++){
  printf("T%d\t",i+1);
  for(j=0;j<c[i];j++){
  printf("%d ",t1[i][j]);
  }
printf("\n");
}
for(i=0;i<num;i++){
  for(j=0;j<num;j++){
                        if(i!=j&&deleted[j]!=1){
                        for(k=0;t1[i][k]!=0;k++){
                        for(l=0;t1[j][l]!=0;l++){
                        if(t1[i][k]==t1[j][l])
                                    count[i][k]=1;
                        }
                        }
                        }
}
```

```
for(m=0;m<c[i];m++)
  if(count[i][m]==1)
          match[i]++;
if(match[i]==c[i])
          deleted[i]=1;
}
for(i=0;i<num;i++)
if(deleted[i]==1)
printf("Remove Test case %d\n",i+1);
getch();
}


/*Program for test case selection for modified lines using regression test case selection algorithm*/

#include<stdio.h>
#include<conio.h>

void main()
{
int t1[50][50];
int count=0;
int candidate[50]={0},priority[50]={0},m=0,pos[50],found[50][50],k,t,c[50],l,num,n,index[50],i,j,modnum,
nfound[50],mod[50];
clrscr();
printf("Enter the number of test cases:");
scanf("%d",&num);
for(i=0;i<num;i++){
          printf("\nEnter the length of test case%d:",i+1);
          scanf("%d",&c[i]);
          }
for(i=0;i<50;i++)
          for(j=0;j<50;j++)
                    found[i][j]=0;
for(i=0;i<num;i++)
          for(j=0;j<c[i];j++){
                    t1[i][j]=0;
                    }
for(i=0;i<num;i++){
          printf("Enter the values of test case %d\n",i+1);
          for(j=0;j<c[i];j++){
                    scanf("%d",&t1[i][j]);
                    }
                    pos[i]=i;
  }
printf("\nEnter number of modified lines of code:");
scanf("%d",&modnum);
printf("Enter the lines of code modified:");
for(i=0;i<modnum;i++)
          scanf("%d",&mod[i]);
while(1)
{
count=0;
for(i=0;i<num;i++) {
          nfound[i]=0;
          pos[i]=i;
          }

for(i=0;i<num;i++){
l=0;
          for(j=0;j<c[i];j++){
          if(candidate[i]!=1){
          for(k=0;k<modnum;k++) {
                    if(t1[i][j]==mod[k]){
```

```
                              nfound[i]++;
                              found[i][l]=mod[k];
                              l++;
                              }
            }
            }
}
}

l=0;
for(i=0;i<num;i++)
  for(j=0;j<num-1;j++)
            if(nfound[i]>nfound[j]){
                      t=nfound[i];
                      nfound[i]=nfound[j];
                      nfound[j]=t;
                      t=pos[i];
                      pos[i]=pos[j];
                      pos[j]=t;
             }

for(i=0;i<num;i++)
            if(nfound[i]==1)
                      count++;
if(count==0)
            break;

  candidate[pos[0]]=1;
  priority[pos[0]]=++m;

printf("\nTestcase\tMatches");
for(i=0;i<num;i++) {
            printf("\n%d\t\t%d",pos[i]+1,nfound[i]);
            getch();
            }

for(i=0;i<c[pos[0]];i++)
            for(j=0;j<modnum;j++)
                      if(t1[pos[0]][i]==mod[j]){
                              mod[j]=0;
                      }

printf("\nModified Array:");
for(i=0;i<modnum;i++){
            if(mod[i]==0){
                      continue;
                      }
            else {
                      printf("%d\t",mod[i]);
                      }
            }
}

count=0;
printf("\nTest case selected.....\n");
for(i=0;i<num;i++)
            if(candidate[i]==1){
                      printf("\nT%d\t Priority%d\n ",i+1,priority[i]);
                      count++;
                      }
if(count==0){
            printf("\nNone");
            }
getch();
}
```

Ruchika Malhotra
She is an assistant professor with the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Prior to joining the school she worked as a full time research scholar and received a doctoral research fellowship from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. She received her master's and doctorate degrees in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Her research interests are in improving software quality, statistical and adaptive prediction models for software metrics, neural nets modeling, and the definition and validation of software metrics. She has published more than 33 research papers in international journals and conferences. Malhotra can be contacted by e-mail at ruchikamalhotra2004@yahoo.com.

Arvinder Kaur
She is a Reader with the School of Information Technology. She obtained her doctorate from Guru Gobind Singh Indraprastha University and her master's degree in computer science from Thapar Institute of Engg. and Tech. Prior to joining the school, she worked with Dr. B.R. Ambedkar Regional Engineering College, Jalandhar and Thapar Institute of Engg. and Tech. Her research interests include software engineering, object-oriented software engineering, software metrics, microprocessors, operating systems, artificial intelligence, and computer networks. She is also a lifetime member of ISTE and CSI. Kaur has published 22 research papers in national and international journals and conferences. Her paper titled "Analysis of object oriented Metrics" was published as a chapter in the book Innovations in Software Measurement (Shaker -Verlag, Aachen 2005). She can be reached by e-mail at arvinderkaurtakkar@yahoo.com.

Yogesh Singh
He is a professor with the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. He is also Controller of Examinations with the Guru Gobind Singh Indraprastha University, Delhi, India. He was founder Head (1999-2001) and Dean (2001-2006) of University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. He received his master's degree and doctorate from the National Institute of Technology, Kurukshetra, India. His research interests include software engineering focusing on planning, testing, metrics, and neural networks. He is coauthor of a book on software engineering, and is a Fellow of IETE and member of IEEE. He has more than 200 publications in international and national journals and conferences. Singh can be contacted by e-mail at ys66@ rediffmail.com.