Prioritizing and Optimizing the Test Cases during Regression Testing

Neha Gupta, Neha Yadav, Aruna Yadav

Abstract— Software testing is one of important phase of the software development life cycle. The software is tested using the test cases. Test cases form the building blocks of the testing process since they get into the core of the source code to find out the faulty code. Test cases are generated by considering the set of input values and their expected outputs. This expected output is again compared with the output after feeding the same values to the software under test. If the expected output is same then the software is working right else there is some problem. After testing the entire software if any change is made to the software the system is tested again which is known as regression testing. To reduce the bulk during regression testing tester need to prioritize and optimize the test cases to yield quick and appropriate results. The paper proposes an approach to optimize the test cases and providing efficient output as compared to the previous approach

Index Terms— Regression testing, Test case prioritization, Optimization ,minimization of test cases,software testing,retesting

1 INTRODUCTION

THIS covers the idea of the test case optimization during the Regression Testing. Software testing is one of the very important phases of the software development Life cycle. The software is tested on all the requirements being given to them as per the priority by the customers. The software is tested using the test cases. Test cases form the building blocks of the testing process since they get into the core of the source code to find out the faulty code. Test cases are generated by considering the set of input values and their expected outputs. This expected output is again compared with the output after feeding the same values to the software under test. If the expected output is same then the software is working right else there is some problem.

After testing the entire software if any change is made to the software the system is tested again which is known as regression testing. It tests the software with the same test cases to check whether the change affects the requirements. Regression testing is considered to be much costlier than any other testing performed because once the software goes under a change it becomes a challenge for the tester to test the entire software again. To reduce the bulk during regression testing tester need to prioritize and optimize the test cases to yield quick and appropriate results. Regression testing is the process of executing the previous test cases on the changed program to see whether the changes are adversely affecting the function performed by the program or not Regression Testing is performed to locate the errors, to preserve the quality of the software and to increase the confidence in the correctness of the modified program.

- *Neha Gupta is currently pursuing Masters degree program in Computer Science Engineering in Jamia Hamdard ,Hamdard UniversityNew Delhi,India and affiliated to KIET, Ghaziabad.*
- *Aruna Yadav and Neha Yadav are Asistant Professor in Department of Computer Science & Engineering, KIET Ghaziabad,India.They have 5years of experience in teachingand research and published various research papers . aruna.yadav21@gmail.com nehayadav1508@gmail.com*

2 RELATED WORK

Gregg R.et. al. [1] describe several techniques for using test execution information to prioritize test cases for regression testing, including: 1) techniques that order test cases based on their total coverage of code components, 2) techniques that order test cases based on their coverage of code components not previously covered, and 3) techniques that order test cases based on their estimated ability to reveal faults in the code components that they cover.. The data also shows, however, that considerable room remains for improvement.

Eric W. et al. [2] describe regression testing is usually performed by running some, or all, of the test cases created to test modifications in previous versions of the software. Many techniques have been reported on how to select regression tests so that the number of test cases does not grow too large as the software evolves.

Yoo S. et.al.[3] discuss test case prioritization seeks to order test cases in such a way that early fault detection is maximized. This paper surveys each area of minimization, selection and prioritization technique and discusses open problems and potential directions for future research. Xuan L.et.al. [4] conduct survey of current research on regression testing and current practice in industry and also try to find out whether there are gaps between them. Gaurav D et.al.[5] presented the various types of regression testing techniques their classifications presented by various researchers , explaining selective and prioritizing test cases for regression testing in detail.

Elbaum E. et.al.[6], this analysis shows that test suite granularity significantly affects several cost- benefits factors for the methodologies considered, while test input grouping has limited effects. Further, the results expose essential tradeoffs affecting the relationship between test suite design and regression testing cost-effectiveness, with several implications for practice.

Kapfhammer M. et.al.[7] advocates a way forward involving a mutually beneficial increased sharing of the inputs, outputs, and procedures used in experiments. Mark H et.al.[8] presents several examples of costs and values that could be incorporated into such a Multi Objective Regression Test Optimization (MORTO) approach. Lijun H et.al.[9] that their techniques can achieve significantly higher rates of fault detection than existing techniques. Hsu Y. et.al.[10] show how mints can be used to instantiate a number of different test-suite minimization problems and efficiently find an optimal solution for such problems using different solvers.

3 BACKGROUND

TEST CASE PRIORITIZATION

Test case prioritization [1], [2] is a technique of ordering the test cases according to a particular condition (a "fitness number"). Test case prioritization defined by Rothermal et al. [1] is as follows:Given: P, a test suite; PP, the set of permutations of P; f,a function from PP to the real numbers.

Problem: Find P' such that

$(\vee P'' (P'' \epsilon PP) (P'' \neq P' [f(P') > (P'')])$

Here, PP is the set of all prioritizations of P and f is defined as a function which when applied to any of the possible prioritizations yield a result.Test case prioritization is performed based on some criterion and it is mandatory since re-execution of the test cases becomes challenging. So if few of the test cases are left out then the most effective test cases are executed.

4 PRIORITIZING AND OPTIMIZING THE TEST CASES DURING REGRESSION TESTING

4.1 Test Case Creation

The test cases are created for C program by using the line preprocessor directive i.e., __LINE__. The preprocessor directive generates the line number which gets executed when the input values are provided to the C program. The creation of the test cases can be explained with the help of the example.

Considering an example which is a C program which can find the area of the two dimensional figures. It finds the area of the square, rectangle and triangle. The choice is given to the user as '1' and '2'. Choice '1' finds the area of the square and rectangle and Choice '2' finds the area of different
types of rectangles.

For example if we input the choice as 1 then the sides which the user can input are 'a' and 'b' whereas choice '2' needs input as 'a', 'b' and 'c'. The output to the program are the area of the two dimensional figures and the line of code executed i.e., for choice '1' and 'a' and 'b' 2 and 8 respectively the output is 16 and the line of code executed are <12,13, 14, 15, 16, 17, 19, 20, 21>.

4.2 GENERATING THE LINE NUMBERS CHANGED

The program is tested once and the changes are made to the program. The section 2 compares the old program and the new program and generates the set of the lines which are changed. In the example the line numbers generated by the Section 2 are <13, 14, 15, 19, 26, 28, 33, 34, 36>.

4.3 ALGORITHM FOR PRIORITIZING AND OPTIMIZING TEST CASES

The most important section of the entire approach is Section 4 in which test suite is reduced, the test cases covering only the changed lines are provided and the entire program can tested using the optimized test cases. The section 4.3 can be explained on the basis of the Table 4.2.

number : total number of test cases.
length[] : the number of elements in each test case.
test[][] : two dimensional array storing the elements of each testcase.
Modnum : the number of modified lines in the source code.
mod[] : one dimensional array storing the lines which are modified.
ncommon[] : number of matching elements between test[][] and mod[].
common[][] : two dimensional array storing the values of the line of code matching with the each test case.
Count: counter variable

ISSN 2229-5518

Table 4.1:The variables used in the pseudocode

```
// First section of the pseudo code:
 Initialization of variables

1. Repeat for i=0 to number of test cases
   a. Repeat for j=0 to number of test cases
      i. Initialize array test cases[i][j] to zero
2. Repeat for i=0 to number of test cases
   a. Repeat for j=0 to number of test cases
      i. Store line numbers of line of source code
       covered by each test case in array common[i][j]
3. Repeat for i=0 to number of modified lines of
   source code
   a. Store line numbers of modified lines of source
      code in array mod[i].
//Second section of the pseudo code:
Comparison between test[ ][ ] and mod []
 4.  Repeat for all true cases
   a. Repeat for i=0 to number of test cases
        i.  Initialize array ncommon[i] to zero
   b. Repeat for j=0 to number of test cases
        i.  Repeat for k=0 to modified lines of source code
     If  testcase[i][j]=mod[k]
Then   Increment common[i] by 1
 common[i][j]=test[i][j]
```

Table 4.2: Test cases with the line of code covered

Test Case Id	Inputs				Expected Output	Line of Code Covered
	Choice	a	b	c		
T1	1	2	8		16	12, 13, 14, 15, 16, 17, 19, 20, 21
T2	1	4	4		16	12, 13, 14, 15, 16, 17, 18, 21
T3	2	2	4	8	0; Invalid Triangle	12, 13, 14, 21, 22, 23, 24, 26, 27, 28, 29
T4	2	2	2	2	1.7; Equilateral Triangle	12, 13, 14, 21, 22, 23, 24, 25
T5	2	6	6	8	17.8	12, 13, 14, 21, 22, 23, 24, 30, 31, 32
T6	1	2	2		4	12, 13, 14, 15, 16, 17, 19, 20, 21
T7	2	8	8	6	22.24	12, 13, 14, 21, 22, 23, 24, 30, 31, 32
T8	2	8	6	8	22.24	12, 13, 14, 21, 22, 23, 24, 30, 31, 35, 36
T9	2	6	8	8	22.24	12, 13, 14, 21, 22, 23, 24, 30, 31, 33, 34
T10	1	8	2		16	12, 13, 14, 15, 16, 17, 19, 20, 21

```
// Third  section of the pseudo code:
Elimination of the common test cases
5. Repeat for i=0 to number of test cases
Initialize count to zero
Repeat for j=i+1 to number of test cases
Set count to zero
  If ncommon [i]>ncommon [j] then
       Repeat for k=0 to number of test cases
        Repeat for l=0 to number of test cases
   If common[i][k]=[j][l] and common[i][j]!=0
                Increment count by 1
If count = common[j] and count to zero then
      Repeat for m=0 to number of test cases
            Set common[i][j] to zero
            Set   common[i][j] to zero

// Fourth section of the pseudo code :
Output depicting the priority of test cases
 6. Initialize count to zero
 7. Repeat for i=0 to number of test cases
    a.  Initialize count=0
    b.Repeat for j=1 to number of test cases
     i.Initialize count=0
     ii. If ncommon[i]!=0 and  ncommon[j]!=0 and  i!=j
       Repeat for k=0 to number of test cases
       Repeat for 0 to number of test cases
 If common[i][k]==common[j][l] and common[i][k]!=0)
            Increment count by 1
                    iii. If count=ncommon[j] and count!=0
                      Repeat for m=0 to number of test cases
                          Initialize common[j][m]=0
                            Initialize ncommon[j]=0
```

4.3.1 Initialization of variables

In the first section the program takes the total number of test cases and the values of line of code covered by the each test case as the input set and the modified lines of the source code as the second input.The test cases are stored in the variable 'number' and the number of modified lines in the variable 'modnum'.The values of the line of code covered by each test case are stored in the two dimensional array 'test []' and the modified lines are stored in the single dimensional array 'mod []'.The output of the first section initializes two 2 dimensional arrays 'test []' and'common[]' and a single dimensional array mod[].

4.3.2 Comparison between test [][] and mod []

The second section of the algorithm compares the array mod[] with the every row of the two dimensional array test[][].The array 'ncommon []' stores the number of common elements obtained after performing the comparison between the mod[] and test[][]. The common values between the mod[] and each row of the 2 dimensional array are stored in are stored in the two dimensional array 'common[][]

Test CaseId	Common[][]	Ncommon[]
T1	13, 14, 15, 19	4
T2	13, 14, 15	3
T3	13, 14, 26, 28	4
T4	13, 14	2
T5	13, 14	2
T6	13, 14, 15, 19	4
T7	13, 14	2
T8	13, 14, 36	3
T9	13, 14, 33, 34	4
T10	13, 14, 15, 19	4

Table 4.3 Output of pseudocode 4.3.2

4.3.3 Elimination of the common test cases

The third section of the algorithm eliminates test cases from the 2 dimensional array common[][] which have the same values.The two test cases with the same values need not to be executed so eliminating one can help in minimising the time of the regression testing.The ncommon[] value with respect to each test case is considered for identifying the test cases having same number of values.The output of the section three is such that the 2 dimensional array assigns 0 values to one of the common test case after comparison.

Test Case Id	Common[][]	Ncommon[]
T1	13, 14, 15, 19	4
T2	13, 14, 15	3
T3	13, 14, 26, 28	4
T4	13, 14	2
T5	0	0
T6	0	0
T7	0	0
T8	13, 14, 36	3
T9	13, 14, 33, 34	4
T10	0	0

Table 4.4 Output of pseudocode 4.3.3

4.3.4 Output: Gives the only required test cases

The fourth section of the algorithm considers that the test case covering the maximum number of elements is prior to any other test case.

Test case Id	Common[][]	Ncommon[]
T1	13, 14, 15, 19	4
T2	0	0
T3	13, 14, 26, 28	4
T4	0	0
T5	0	0
T6	0	0
T7	0	0
T8	13, 14, 36	3
T9	13, 14, 33, 34	4
T10	0	0

Table 4.3 Output of pseudocode 4.3.4

5 RESULTS AND CONCLUSION

The algorithm provides the minimal set of test cases as an output. These test cases cover the entire test suites. The following examples are considered to compare both the approaches.The following examples are tabulated with the input values, output values and the line of code covered. Each of the examples consists of the set of generated test cases and their respective outputs. Line of code covered are the line number executed for each input values. The modified lines are generated for each of the example by making changes to the original program and then comparing the line numbers of each program. The final output of the program or the optimized test cases are shown after every table.

ISSN 2229-5518

EXAMPLE 1

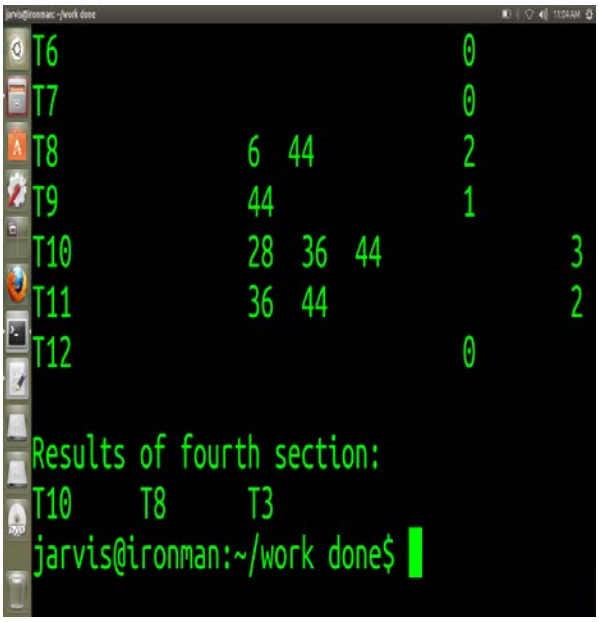
The example consists of the total 12 test cases.

Input: The input values provided by the user are month, day, year.
Output: The output of the algorithm is the day of the week.

The Section 2 of the approach generates the lines which were changed after testing the program once. The modified lines are <6, 28, 36, 44, 50, 61>.

Test Case Id	Month	Day	Year	Expected Output	Line of Code Covered
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

The output of the algorithm or the minimum number test cases required to test the entire program are T10, T8, T3.



REFERENCES

[1] Rothermel Gregg, Untch Roland H., Chu Chengyun, Harrold Mary Jean, 2000, Prioritizing Test b Cases For Regression Testing, International Symposium on Software Testing and Analysis, pp102-112, 2000.

[2] Wong Eric W., Horgan J.R., London Saul, Agrawal Hira, 1997, A Study of Effective Regression Testing in Practice. IEEE International Symposium on Software Reliability Engineering, pp 264-274, 1997.

[3] Yoo, S., Harman, M., 2007, Regression Testing Minimisation, Selection and Prioritisation : A Survey, Wiley Interscience, 24 September 2007.

[4] Lin Xuan, Regression Testing in Research and Practice, Computer Science and Engineering Department University of Nebraska, Lincoln.

[5] Duggal Gaurav, Suri Bharti, 2008, Understanding Regression Testing Techniques, Second Conference on Opportunities and Challenges in Information Technology, 2008.

[6] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsk, Praveen Kallakuri, Xuemei Qiu,2004, On Test Suite Composition and Cost-Effective Regression Testing, ACM Transactions on Software Engineering and Methodology, vol. 13, issue 3, pp. 277-331, July 2004.

[7] Kapfhammer M. Gregory, 2011, Empirically Evaluating Regression Testing Techniques: Challenges, Solutions, and a Potential Way Forward, IEEE 4th International Conference, pp. 99-102, 21-25 March 2011.

[8] Harman Mark, 2011, Making the case for MORTO: Multi objective regression test optimization, The 1st International Workshop on Regression Testing, pp. 111–114, 2011.

[9] Mei Lijun, Zhang Zhenyu , Chan W. K., Tse T. H., 2009, Test Case Prioritization for Regression Testing of Service Oriented Business Applications, Proceedings of the 18th International Conference on World Wide Web, ACM Press, New York, 2009.

[10] Hsu Y. Hwo and Orso Alessandro, 2009, MINTS: A General Framework and Tool for 7Supporting Test-suite Minimization, Proceedings of the 31st IEEE and ACM SIGSOFT International Conference on Software Engineering, pp. 419–429, May 2009.

[11] Malhotra Ruchika, Kaur Arvinder, Singh Yogesh, 2010, A Regression Test Selection and Prioritization Technique, Journal of Information Processing Systems, vol.6, no.2, June 2010 .