PRIORITIZE REGRESSION TEST CASES

[1]Avinash Gupta, [2]Dharmender Singh Kushwaha
[1,2]MNNIT Allahabad
Email:[1]avinashg.mnnit@gmail.com,[2]dsk@mnnit.ac.in

Abstract—Test suites can be reduced to a smaller suite that guarantees equivalent coverage, termed as test suite minimization. Test case prioritization techniques prioritize and schedule test cases in an order that attempts to maximize some desired objective like achieving code coverage at the fastest rate in order to minimize the regression testing. The proposed approach is computationally simple requiring lesser number of computations. It also identifies and eliminates those program statements that have been tested in the previous test sessions. This achieves complete regression testing in fewer numbers of sessions.

I. INTRODUCTION & RELATED WORK

Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix [1, 2, and 6]. However, as software evolves, the test suite tends to become enormous, thus posing constraints to execute the entire test suite [7]. This limitation leads to the need of techniques that reduces the effort required for regression testing. Three different techniques have therefore been proposed for test suite reduction. These are prioritization, selection, minimization of test suite.

A test suite minimization lowers the cost by reducing a test suite to a minimal subset that maintains equivalent coverage of original set with respect to particular test adequacy criterion [8].

As mentioned by S. Yoo, M. Harman [10], test case selection, or the regression test selection problem, is essentially similar to the test suite minimization problem: both problems are about choosing a subset of test cases from the test suite.

Test case prioritization is the process of scheduling test cases in an order to meet some performance goal [9]. The test suite may contain test cases on higher priority which may not be able to detect the errors [3]. Hence, several techniques have been proposed for prioritizing the existing test cases to accelerate the rate of fault detection in regression testing. Some of these approaches are Coverage-based Prioritization [9], Interaction Testing, Distribution-based Approach [5], Requirement-based Approach, and the Probabilistic Approach [4]. All these approaches apart from probabilistic approach referred above consider prioritization as an unordered, independent and one-time model. They do not take into account the performance of test cases in the previous regression test sessions, such as the number of times a test case revealed faults [10]. History Based Approach (HBA) has been applied to increase the fault detection ability of the test suite. Kim and Porter [4] considered the problem of prioritization of test cases as a probabilistic approach and defined the history-based test case prioritization. Khalilian et al. [3] proposed an extension of history-based prioritization proposed in [4], and modifies the equation given by Kim and Porter [4], to have dynamic coefficients. The priority is calculated using the mathematical equation by computing the coefficients of the equation from the historical performance data. Avinash et al. [11] extends the

approach proposed by Khalilian et al. in [3] by prioritizing the modified lines.

In this paper, we propose a new approach which is an extension of the Avinash et al. [11]. Unlike in [11], require extensive computation of parameters, the proposed approach requires less number of computations.

The rest of the paper is organized as follows. In Section 2, we present the proposed approach and implementation. Section 3 describes performance analysis and comparison results. We conclude the paper and discuss future work in Section 4.

II. PROPOSED APPROACH & IMPLEMENTATION

The proposed approach – prioritize regression test cases, although an extension of the history based approach presented in [3, 11], applies a new set of prioritization equations that maximizes the execution of yet to be executed test case and eliminates already executed test cases. In contrast to the existing approaches, the proposed approach applies the prioritization equation on each modified line of the code as against each test case. The proposed approach also ensures that those test cases are selected for each modified line such that the test case has the maximum coverage among all the test cases which contain the modified line of code.

Our proposed approach has been implemented in a 'C' program and the history is being stored in text format in text files. The history contains all the test cases and parameters such as number of executions of test case, number of times fault detected by test case, number of times each line has been delayed execution. The test cases contain the number of all the lines traversed along the line of execution of the test case. The parameters have been stored in the form of arrays, where each index represents a line in the code. The proposed approach in this paper includes the steps shown in Figure 1.

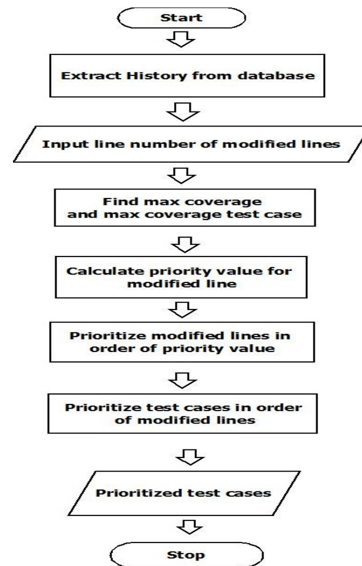

Figure 1: Flow chart of proposed algorithm

Step1: Extract History From Database

In this phase the following parameter values of the previous session are extracted from the database:-

h_k[]=0 if test case has been executed for a statement in test session k-1, otherwise set to (h_{k-1} + 1).

mod_locode[] = an index is set to 1 if the corresponding line has been modified else to 0.

del_locode[] = an index is set to 1 if the corresponding line has been deleted else to 0.

PR_{k-1}[] = the value at an index indicates the priority value of the corresponding line in last session.

PR_k[] = the value at an index indicates the priority value of the corresponding line in present session.

Step2: Input Modified Lines

The modified lines are taken as input from the user through a well defined interface of the program. Any new test cases are also entered through this interface.

Step3: Find max coverage and max coverage test case

If the modified line say, mi is present in a testcase T_i , then

max coverage of line mi = max (no. of lines in Ti /total number of lines) * 100

for all Ti in which mi is found.

Ti is the max coverage test case for mi, if Ti has max code coverage.

Step4: Calculate Priority Value for Modified Value

For each modified line, the priority value is calculated using Eq. (1)[11].

$$PR_k = (\alpha . h_k + \beta . PR_{k-1})/k \quad (1)$$

In the proposed approach, value of β is initialized as 1 in the 1st session and for subsequent sessions; it is set to 0 for fault detected or 1 otherwise. Likewise value of 'α' is initially set to 1 and is increased by 0.2 in subsequent sessions.

In Eq. (1), h_k is the test cases execution history. In Eq. (1), PR_0 is defined for each test case as the percentage of code coverage of the test case. The presence of PR_0 will be helpful in refining the ordering of the test cases in the first session.

Step 5: Prioritize Modified Lines

Modified lines are prioritized by their corresponding PR_k, in descending order. If the modified lines m1, m2 and m3 have PR_k values as -

PR_k [m1] = 10.56, PR_k [m2] = 54.56, PR_k [m3] = 9.64

Hence ordering would be -

m2, m1 , m3.

Step 6: Prioritize Test cases in order of modified lines

Max coverage test case for, say

m1 = T2, m2 = T1, m3 = T3

Hence ordering of test cases in order of the respective modified line m2, m1 and m3 would be - T1, T2, T3.

Step 7: Output prioritized the test cases

The final output for session k is T1, T2, T3.

After prioritizing, the test cases are executed. Let us assume that only 40% of all the test cases prioritized are able to get executed. Out of all the test cases executed, there are certain test cases which detect fault, and after debugging a fault is detected. Then, the parameters are updated in the following manner-

- h_k = 0 if test case has been executed for a statement in test session k-1, otherwise set to (h_{k-1} + 1).
- Value of β is set to 0 for fault detected or 1 otherwise.
- Value of 'α' is increased by 0.2 in subsequent sessions.
- Eliminate the statements where faults are detected.
- Select 40% from the reduced set of test cases. The database is updated with all these modifications.

Case Study:

The proposed approach is demonstrated with an example here. We are considering a small 'C' program as shown in Figure 2 and its modified version in Figure 3. The program in Figure 2 calculates the value of mathematical equations. The equation consists of variables x and y whose value depends on the value of a, b, c, d and e.

```
1
2 public class TestProgram {
3       int x=0;
4       int y=0;
5
6 int takeInput(int a,int b, int c, int d, int e)
7 {
8       if(a>0)
9       {
10              x =2;
11      }
12      else
13      {
14              x=5;
15              if(b>0)
16                      y = x+1;
17              else
18                      y =x-1;
19      }
20      if(c>0)
21      {
22              if(d>0)
23              {
24                      if(e>0)
25                      {
26                              return(1/(b+2));
27                      }
28                      else
29                      {
30                              return(1/(y+4));
31                      }
32              }
33              else
34              {
35                      return(1/(x+5));
36              }
37      }
38      else
39      {
40              return(1/(x+5));
41      }
42}
43}
```

Figure 2: Sample Program

```
1
2 public class TestProgram {
3       int x=0;
4       int y=0;
5
6 int takeInput(int a,int b, int c, int d, int e)
7 {
8       if(a<0)
9       {
10              x =2;
11      }
12      else
13      {
14              x=5;
15              if(b>0)
16                      y = x+1;
17              else
18                      y =x-1;
19      }
20      if(c>0)
21      {
22              if(d>0)
23              {
24                      if(e>0)
25                      {
26                              return(1/(b-1));
27                      }
28                      else
29                      {
30                              return(1/(y-4));
31                      }
32              }
33              else
34              {
35                      return(1/(x-5));
36              }
37      }
38      else
39      {
40              return(1/(x-5));
41      }
42}
43}
```

Figure 3: Modified Sample Program

The program in Figure 3 is modified at line numbers 8, 26, 30, 35 and 40. The modified lines are each shown with a dark underline in Figure 3. The changes in each of the modified lines are shown in Table 1. These modifications in the program will introduce divide by zero error in the program.

Table 1: Changes in the Sample program

Line no.	Original line	Modified line
8	a>0	a<0
26	(1/(b+2))	(1/(b-1))
30	(1/(x+4))	(1/(x-4))
35	(1/(x+5))	(1/(x-5))
40	(1/(x+5))	(1/(x-5))

The Control Flow Graph (CFG) for the sample program and the modified sample program is same because in Figure 3, neither the branch condition has changed nor has a new line been added. Based on branch coverage, we have the following test cases:

T1 - 8 9 10 11 20 38 39 40 41 42 43

T2 - 8 12 13 14 15 17 18 19 20 21 22 33 34 35 36 37 42 43

T3 - 8 12 13 14 15 16 19 20 21 22 23 24 25 26 27 42 43

T4 - 8 12 13 14 15 17 18 19 20 21 22 23 24 25 26 27 42 43
T5 - 8 9 10 11 38 39 40 41 42 43
T6 - 8 12 13 14 15 16 19 38 39 40 41 42 43
T7 - 8 12 13 14 17 18 19 20 21 22 23 28 29 30 31 32 37 42
These test cases are kept in the 'testcases.txt' file.

Now, for session k=1:

Step 1: Extract history from database

For each modified line, the value of parameters, namely h_k, PR_k and PR_{k-1} is set to 0 as shown in the table. PR_{k-1} is test case priority in current session. PR_k is test case priority in next session.

Step 2: Input line numbers of modified lines

As is mentioned in Figure 3, the line numbers of modified lines are 8, 26, 30, 35 and 40. The line numbers of modified lines are entered into the program via the input interface of the program.

Step 3: Find max coverage and max coverage test case

Since, Code coverage of a test case T= (no. of lines in the test case / total number of lines in the program)* 100
Hence, max code coverage for line 26= 36%
Since, Test case T4 is having the max code coverage value of 36 among all the test cases containing the line 26
So, max code coverage test case for line 26 = T4
Similarly max code coverage test case for line number 8, 30, 35 and 40. The results are:
Max code coverage test case for line 8 = T2, and max code coverage = (18 / 50) * 100 = 36%
Max code coverage test case for line 30 = T7, and max code coverage = (18 / 50) * 100 = 36%
Max code coverage test case for line 35 = T2, and max code coverage = (18 / 50) * 100 = 36%
Max code coverage test case for line 40 = T6, and max code coverage = (13 / 50) * 100 = 26%

Step 4: Calculate priority values for modified lines

For session k=1, the status of all the parameters is as shown in the Table 2.

Table 2: Status of parameters before session k=1

Line No.	h_k	PR_k	PR_{k-1}
8	0	0	36
26	0	0	36
30	0	0	36
35	0	0	36
40	0	0	26

Now, Substituting the values of h_k, α=0, β=1 and PR_{k-1} in Eq. (1) from Table 2, to calculate the corresponding value of PR_k for lines 8, 26, 30, 35 and 40. We get: PR_k [8] = 36, PR_k [26] = 36, PR_k [30] = 36, PR_k [35] = 36, PR_k [40] = 26.

Step 5: Prioritize the modified lines in order of priority value

Based on the priority values calculated in step 4, modified lines in order of priority as per PR_k values are: 8, 26, 30, 35, 40.

Step 6: Prioritize the test cases in order of modified lines

Max code coverage test case for Line 8 is T2, Line 26 is T4, Line 30 is T7, Line 35 is T2, Line 40 is T6, So ordering the test cases in the same order as their respective modified lines are, we get:T2, T4, T7, T2, T6. Removing the repeated test case T2 from the fourth place, we get T2, T4, T7, T6.

Step 7: Output prioritized the test cases

The final output for session k=1 is T2, T4, T7, T6.

After the end of session 1, 40% of all the test cases are executed i.e. test cases T2, T4, T7 are executed in the order of priority as given by the final output of session 1. After the execution of the test cases it is found that all the test cases fail i.e. detect faults.

T2 detects fault at statement 8 and 35, T4 at 8 and 26 and finally T7 detects fault at statement 30.

Session 2: (For session k=2)

It is evident that fault at statement 40 is yet not detected. Eliminating the statements where fault have been detected in session 1 from the set of statement number 8,26,30, 35 and 40 leaves only statement 40. Hence, one has to initiate the next session. The values of the various parameters for these sessions are illustrated in the Table 3.

Table 3: Status of parameters before session k=2

Line No.	h_k	PR_k	PR_{k-1}
40	1	0	26

In Table 3, the values for PR_k in session k=1 become the respective values of PR_{k-1} in session k=2 for each line. Since, max code coverage test case for line number 40 i.e. T6 did not execute in session 1, the value of h_k for line 40 is increased from 0 to 1 before the start of session 2. Similarly, the value of 'β' is set to 1 and the value of 'α' is increased by 0.2. Eliminate the

statements where faults are detected. Hence, we now proceed as we did in session k=1.

Now, Substituting the values of h_k=1, α=1.2, β=1 and PR_{k-1}=26 in Eq. (1), we get PR_k [40] = 13. Based on the priority values calculated in step 4, modified lines in order of priority as per PR_k values are: 40. Since, Max code coverage test case for Line 40 is T6. Therefore final output for session k=2 is T6.

Test case covering statement 40 i.e. T6 is executed and all the faulty statements of the code have been found in the second session itself. Thus the proposed approach is able to detect all the faults in two sessions as compared to 3 sessions required by approaches proposed in [11] and [3].

III. Performance Analysis:

Proposed approach is illustrated by three programs of Java, C and C++ based platforms. The references of which are given in Table 5.

Table 5: Project References

Program	LOC	Platform	URL/Journals
Branch Coverage Sample Program	50	Java	An improved method for test case prioritization by incorporating historical test case data [11]
Bank Account	52	Java	Automated Behavioural Based Regression Testing [10]
Payroll Managment System	300	C++	http://www.softwareandfinance.com/forums/index.php?topic=407.0

The proposed approach is compared with the one proposed by Khalilian et al. [3] and Avinash et al. [11] by applying both the prioritization mechanisms on the programs listed in Table 5. Five faults were seeded in each of the programs. Multiple sessions of regression test were followed by the proposed approach as well as by Khalilian et al. [3] and Avinash et al. [11] approaches.

Table 6: Proposed Approach Results

Program	Line No. Modified	Session No.	Faulty Lines Detected by Avinash et al. [11] Approach	Faulty Lines Detected by Khalilian et al. [3] Approach	Faulty Lines Detected by the Proposed Approach
Branch coverage sample program	8, 26,30, 35, 40	S1	26, 30, 35	26, 30	8,26,30,35
		S2	No Faults Found	35, 40	40
		S3	40, 8		
Bank Account	6,17,22, 24,27	S1	17, 22,24, 27	17, 22, 24, 27	17, 22,24, 27
		S2	No Faults Found	No Faults Found	6
		S3	6	6	
Payroll Management System	70,118, 124, 207,231	S1	124, 118, 231, 207	124, 118, 231	124, 118, 231, 207
		S2	70	70	70
		S3		207	

Results in Table 6 shows the faults detected in each session by the 3 approaches. This is followed by a comparison of the total number of sessions required to find all the faults in the proposed approach as compared to Khalilian et al. [3] and Avinash et al. [11] approaches as illustrated in Table 7.

After analysing the results, it is found that the number of faults detected in program segments per session is more than or equal to for the proposed approach than the other 2 approaches. At the same time the number of sessions required to discover all faults in the proposed approach is always lesser as illustrated in figure 3.

Table 7: Comparison of the Proposed Approach with Avinash and Khalilian Approach

Projects	No. of sessions required in Avinash et al. [11] Approach	No. of sessions required in Khalilian et al. [3] Approach	No. of sessions required in the Proposed Approach
Branch Coverage Sample Program	3	2	2
Bank Account	3	3	2
Payroll Management System	2	3	2

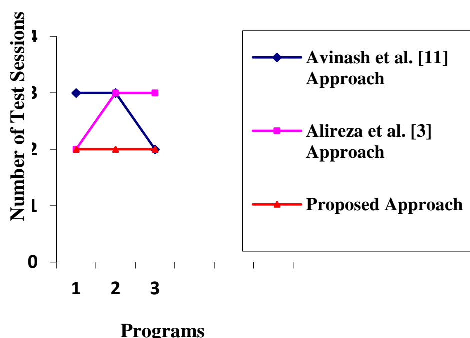

Figure 3: Comparison Chart of Proposed approach

IV. Conclusion & Future Work

The proposed approach for prioritization and reduction of test cases for regression testing is computationally simple requiring lesser number of computations. It also identifies those program statements that have been tested in the previous test sessions, that are eliminated from the set of statements over which test cases need to be run. After analysing the results, it is found that the number of faults detected in program segments per session is more than or equal to for the proposed approach than the other approaches. The proposed approach consumes at least 33% lesser test sessions as compared to other existing approaches. It is noteworthy that the proposed approach requires execution of just 1 equation as compared to 3 by the other two approaches for prioritization.

In future work, we may consider other factors such as severity of fault detected, which may help to refine the process of prioritization. The program for the proposed algorithm takes line number of modified lines as input manually. Experiments may be done on more programs to analyse the results in different perspective.

References

[1] H.-Y. Hsu and A. Orso. Mints: A general framework and tool for supporting testsuite minimization. In Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pages 419–429. IEEE, 2009.

[2] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pages 137–146. IEEE, 2010.

[3] A. Khalilian, M. Abdollahi Azgomi, and Y. Fazlalizadeh. An improved method for test case prioritization by incorporating historical test case data. Elsevier Science of Computer Programming, 78(1):93–116, 2012.

[4] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, pages 119–129. IEEE, 2002.

[5] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on, pages 442–453. IEEE, 2003.

[6] R. Mall. Fundamentals of software engineering. PHI Learning Pvt. Ltd., 2009.

[7] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage based test sets. In In Proc. Twelfth Int'l. Conf. Testing Computer Softw. Citeseer, 1995.

[8] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In Software Maintenance, 1998. Proceedings. International Conference on, pages 34–43. IEEE, 1998.

[9] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. Software Engineering, IEEE Transactions on, 27(10):929–948, 2001.
[10] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability, 22(2):67–120, 2012.
[11] Avinash Gupta, Nayaneesh Mishra, Dushyant Kumar Singh and Dharmender Singh Kushwaha, "Test Cases Reduction through Prioritization Technique", International conference on Advance in Computing, Communication and Information Technology, (CCIT-14), London, June, 01-02, 2014.