A Greedy Approach for Coverage-Based Test Suite Reduction

Preethi Harris[1] and Nedunchezhian Raju[2]
[1]Faculty of Information Technology, Sri Ramakrishna Engineering College, India
[2]Faculty of Computer Science and Engineering, Sri Ranganathar Institute of Engineering and Technology, India

Abstract: *Software testing is an activity to find maximum number of errors which have not been discovered yet with optimum time and effort. As the software evolves the size of the test suite and grows with new test cases being added to the test suite. However, due to time and resource constraints rerunning all the test cases in the test suite is not possible, every time the software is modified, in order to deal with these issues, the test suite size should be manageable. In this paper a novel approach is presented to select a subset of test cases that exercise the given set of requirements with for data flow testing. In order to, express the effectiveness of the proposed algorithm, both the existing Harrold Gupta and Soffa (HGS) and Bi-Objective Greedy (BOG) algorithms were applied to the generated test suites. The results obtained from the proposed algorithm were compared with the state-of-art algorithms. The results of the performance evaluation, when compared to the existing approaches show that, the proposed algorithm selects near optimal test cases that satisfy maximum number of testing requirements without compromising on the coverage aspect.*

Keywords: *Software testing, test cases, test suite, requirements, coverage and adequacy criterion.*

Received April 2, 2012; accept August 26, 2012, published online April 17, 2014

1. Introduction

Software testing is a vital activity in the development of software to find bugs as early as possible. The objective of software testing is to detect faults in the program and therefore, provide more assurance for customers on the quality of the software. Any software that is developed and put into use may be subjected to addition or modification of existing features. With a tremendous number of possible test cases available as software evolves, testers have no means to control the size of the test suite. The literature survey [10, 12] throws light upon the fact that software testing consumes a greater chunk of the development cost. With software projects also being subjected to time and resource constraints, ways to address test suite reduction has become a topic of interest among researchers. During test case generation or after creating the test suite, the effectiveness of the test process can be improved if a minimal subset of test cases could be determined to exercise all the test requirements as the original test suite. Apparently, the lesser the number of test cases, the lesser time it takes to test the program which consequently reduces the computational effort of running the entire test suite.

However, another important issue to be addressed during test suite reduction is the coverage aspect. Also, coverage-based reduction techniques should ensure that majority of the execution paths of the given program are exercised. The general implication from the previous research work [15, 18] is that test case(s) that do not add to the coverage of a test suite are more likely to be ineffective in satisfying the specified requirements. From the literature survey it can be inferred that test suite reduction approaches significantly reduce the size of the test suite [2, 3, 16]. However, how far the reduced test suite obtained satisfies the test metric(s) under consideration is an important issue to be addressed. In fact some potential drawbacks observed in test suite reduction survey involves random selection of test case in the event of a tie (two or more test cases satisfying the same set of requirements), complex mathematical operation for test suite reduction, quality of test case(s) [7, 14] etc., Thus, the trade-off between coverage and optimal test case selection is vital in test suite reduction.

In this paper a new algorithm for Test Suite Reduction called Coverage Based Test Suite Reduction (CBTSR) has been proposed. The contributions of this paper include the following:

- Identifying an optimal representative test set comprising of test cases which are related to the given testing objective.
- Applying data flow testing to generate test cases and requirements to examine the physical structure of the program and locate sub-paths traversed by variables.
- Using the proposed CBTSR algorithm for test suite reduction.

- Performing a set of empirical studies on ten subject programs. Then comparing the relative performance and effectiveness of the proposed reduction algorithm with the state-of-art Harrold Gupta and Soffa (HGS) [7] and Bi-Objective Greedy (BOG) [14] algorithms.

The rest of the paper contents are organized as follows: Section 2 describes the problem statement. In section 3 a detailed outlook of the proposed test suite reduction algorithm and section 4 describes the results and discussion. Finally section 5 presents the concluding remarks.

2. Test Suite Reduction Problem

According to the definition of test suite problem given [7, 10]:

- A test suite T of test cases $\{t_1, t_2, t_3, \ldots, t_n\}$, known as universal test suite.
- A set of testing requirements $\{r_1, r_2, \ldots, r_m\}$ that must be covered to provide the desired coverage for the program under consideration.
- Subsets $\{T_1, T_2, \ldots T_m\}$ of T known as test sets where each test set is associated with r_i, such that any one test case(s) belonging to T_i satisfies r_i.

The objective of test suite minimization problem is to find the representative set (reduced test suite) T_{rs} that exercises the same set of requirements as those exercised by the original test suite T.

2.1. Background

The problem of finding the representative set is equated to the set-cover problem [10]. The set-cover problem has been shown to be NP complete [7] in HGS algorithm. Nevertheless, there has been some research work [7, 14] in the area of computing optimally-minimized test suites. Most of the other research works in test suite minimization have however relied on heuristics for computing near-optimal solutions [2, 3, 11, 16, 17]. Several approaches have been proposed in literature [1, 2, 4, 7, 8, 9, 11, 14, 16, 17] for addressing test suite reduction issues. In practice test suite reduction approaches generally focus on removing obsolete and redundant test cases from the universal test suite [14]. The objective of test suite minimization in software testing is to retain the most effective test cases only [14, 15]. Further, these test cases should be capable of satisfying the most number of test requirements and consequently also expose the faults in existence. Along with the test suite reduction techniques, usage of coverage aspects is also vital. Coverage criteria [7, 8, 9, 11] such as branch coverage, statement coverage, data flow coverage, MC/DC and call stack coverage to name a few, exercise greater assurance to the quality, reliability and stopping rules [4, 5, 6, 12, 18] for test engineers.

HGS algorithm proposed by Harrold *et al.* [7] has drawn a lot of attention towards test suite reduction. This algorithm uses the concept of cardinality (number of occurrence of a test case in each test set) to reduce the test suite size. It begins the test suite minimization process by selecting singleton test cases (test cases with cardinality one) and proceeds to the next higher cardinality test cases. Similarly the recently proposed BOG algorithm by Saeed and Alireza [14] uses complex matrix operations to reduce the test suite size. However, a potential drawback of the HGS algorithm is the random selection of test cases during test suite reduction in the event of a tie. Further, in BOG algorithm the orders in which test sets are subjected to reduction adversely affect the values of the representative set. Hence, from the literature survey it is quite apparent that there is a need for research work to focus on issues arising while optimizing the test suite size. In the next section the CBTSR algorithm is described with an example.

3. Test Suite Reduction Algorithm

3.1. Related Concepts

The number of requirements R may be finite or infinite. However, from a pragmatic point of view, it is assumed that R is finite. For each requirement, $r_i \subset R$, there is a test case t_j in the input domain that satisfies it. As a result, a finite test suite T also exists. The variables m and n are used to denote the size of R and T, respectively. The Boolean matrix A of size $m \times n$ is used to describe the satisfaction relation between requirements and test sets such that $\forall\, r_i \in R$ and $\forall\, t_j \in T$ Equation 1:

$$a_{ij} = \begin{cases} 1 & \text{if } r_i \text{ is covered by } t_j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Where, for i $= 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

The sum vector S represents the count of all "1" in the i^{th} row of a_{ij}. The representation of vector S is denoted in Equation 2:

$$s = \begin{bmatrix} \sum a_{1n} \\ \sum a_{2n} \\ \vdots \\ \sum a_{mn} \end{bmatrix} \quad (2)$$

Thus, the sum vector values can be further simplified and represented as in Equation 3:

$$S = \{c_i\}, i \rightarrow 1 \text{ to } m \quad (3)$$

Where, $c_i = \sum_{j=1}^{n} a_{ij}$

In the process of test suite reduction, the mapping f: $T \rightarrow R$ can be defined as a Boolean function. The coverage relationship expressed as a requirement matrix can be considered as the satisfaction relationship among test requirements and test cases in the optimal representative set selection problem. Thus, the boolean function simplification problem can be equated to the optimal representative set selection problem.

3.2. CBTSR Algorithm

The algorithm CBTSR shows the generation of the reduced test suite through simple mathematical operations. In the algorithm the following assumptions were made: Let n denote the number of test cases in a test set and m denote the number of test requirements. The other related issues are: Each test set T_i consists of test cases corresponding to a requirement. The reduction process in the proposed CBTSR approach begins with the construction of test case requirement matrix *'A'*. This matrix maps the test cases with the testing requirements. An association between a test case and requirement is indicated by one or zero otherwise. In the matrix each i^{th} row denotes the requirement coverage and each j^{th} column denotes the test case overlap with the requirement(s). The algorithm first includes all the test cases t_js that occur as a single element in the test set T_is (singleton test case), to the temporary set T_s. Then:

Algorithm 1: CBTSR

```
Input: Test cases in the given test sets along with requirements.
Test Sets: $T_1,T_2, ...,T_m$.
Associated requirements: $r_1,r_2,..., r_m$
Test cases: $t_1,t_2,...,t_n$.
Output: Reduced test suite.
 $T_{rs}$ a representative set of $T_1, T_2,...,T_m$.
Begin
{
A: is a boolean matrix, 1-covered and 0 – uncovered.
sel_tc:= {}: selected test cases returned by the subroutine select_optimal()
list_t: list of test cases
list_r: list of requirements
$T_s$:={}: selected singleton test cases
$T_{temp}$:={}: temporarily selected test cases
algorithm CBTSR
{
  Begin
      list_t : =all $t_j \in T$ //Contains all the test cases
      list_r : =all $r_i \in R$ //Contains all the requirements construct
      the matrix A; //matrix denoting relationship between
      requirements and test cases
        for each $r_i$ do
            construct the vector S //Vector consisting of sum of
            the elements from row   1..m of matrix A
           $T_s := \cup T_i$;   //Assign test set(s) with row sum= 1
          to the temporary set $T_s$
          update list_t:= remove all $t_j$ selected;
         // Update by removing all the marked test cases
         update list_r:= remove all $r_i$ selected;
        //Update by removing all the marked requirements
        endfor
      //Consider unmarked requirements where $i \rightarrow 1$ to m and select the
       Corresponding  test set
       for each  $T_i$ such that there exists $r_i$ do
          sel_tc=select_optimal(list_r, list_tc);
          //Invoke the subroutine by passing the list of test
          cases and requirements
          update list_t:= remove all $t_j$ selected;
            // Update by removing all the marked test cases
           update list_r:= remove all $r_i$ selected;
           //Update by removing all the marked requirements
           $T_{temp} := \cup$ { sel_tc}; // distinct test cases with
           highest coverage value
       endfor
    $T_{rs} = T_{temp} \cup T_s$ // union of all distinct optimal test cases
  end
}
end CBTSR
```

Subroutine 1: select_optimal (list_r,list_t)

```
/*selects test sets to be included in the representative set */
Input: unmarked test cases and requirements
Output: Representative set
S: integer vector denoting number of requirements covered by a test sets
T: test set with highest coverage value
max(): returns S vector row(s) having highest sum value
  Begin
   {
      reconstruct the vector S; // sum of requirements covered
      by test cases
      if  max(S) > 0
         return (T=max(S)); // return the row with the highest
         value to the variable sel_tc
      else
         return; // return to the main program
    }
  end
}
end select_optimal
```

The corresponding occurrences of the requirements r_is and test cases t_js in the test case requirement matrix are then reset to the value zero as represented in Equations 4 and 5. Equation 4 resets all the elements in column *'j'* to the value zero and Equation 5 resets all the elements in row i to the value zero. This is followed by removing the test case and requirement from the lists: *list_t* and *list_r*. Then, the subroutine *select_optimal()* is recursively called to select the remaining test cases:

$$A\left[All, t_j\right] = 0 \tag{4}$$

$$A\left[\ r_i\ \right] = 0 \tag{5}$$

Another subroutine *max()* when invoked returns the row(s) with the maximum test case(s) covering the given requirement, from the vector S. The t_js marked in row(s) returned are added to another temporary set T_{temp}. Again the corresponding requirement r_is and t_js are reset to the value zero and the details of the same are also updated in the lists. The algorithm recursively selects test cases and updates the test case requirement matrix and lists containing the selected test cases/requirements, until the vector S returns value(s) greater than zero. Finally, the distinct test cases in T_{temp} are combined with test cases in T_s to generate the representative set T_{rs}. The worst case run time for the proposed algorithm CBTSR constitutes the time to mark the test case requirement matrix $o(n \bullet m)$. Computing the vector S and selecting test case(s) to be added to the representative set requires $O\left(\frac{m(m+1)}{2}\right)$. Under these assumptions the complexity of the proposed algorithm becomes $O\left(m^2\right)$.

3.3. Concept Illustration

With a majority of programs written to handle data, variable utilization becomes vital. In such scenarios, the concept of data flow testing can be used to examine the variables. It includes the definition and assignment of variable(s) throughout the program. The paths from points where each variable is defined to points where it is referenced are called definition-use pairs or DU-pairs.

The test cases are generated based on all possible flow of data from the declaration to the assignment. Similarly the DU-pairs obtained are used as a criterion to assess the coverage of requirements by the representative set, T_{rs}.

To illustrate the proposed CBTSR algorithm and the state-of-art algorithms like HGS and BOG, a hypothetical program is listed below:

Program 1: Odd_Even

```
1.  program odd_even
2.  var  n
3.  input(n)
4.    if (n>0)
5.       print (“Number positive”+n)
6.    else
7.       print (“Number negative”+n)
8.       neg_no=n
9.       n=num_convert(neg_no)
10.      print “Natural  Number determined”
11.  if (n%2==0)
12.      print (“Number even”+n)
13.  else
14.      print (“Number odd”+n)
15.      print “Type of Number determined”
16. End program
```

The test cases are generated based on all possible flow of data from the declaration to the assignment. Similarly the DU-pairs obtained are used as a criterion to assess the coverage of requirements by the representative set. In the control flow graph Figure 1 the statements of the form ‘*n:=*’ represents definition of n and ‘*:=n*’ uses of the variable *n*. Through data flow analysis the DU-pairs are generated to serve as requirements for testing the odd/even program. For the program odd_even, there are two defining nodes in Statements 3, Statement 9 and six usage nodes in Statements: 4, 5, 7, 8, 11, 12, 14, thus, leading to four DU-paths as indicated in Table 1.

From the values in Table 1, every execution path is considered to be analogous to a test case. From the DU-paths the corresponding DU-pairs are tabulated as indicated in Table 2. These DU-pairs serve as bench marks for constructing the test sets T_i with the associated test cases t_j. Table 2 depicts the test sets for the sample program considered. Thus, the universel test suite $T=\{t_1, t_2, t_1, t_2, t_3, t_4, t_1, t_3, t_2, t_4, t_3, t_4, t_3, t_4, t_3, t_4\}$ was generated for the sample program considered.

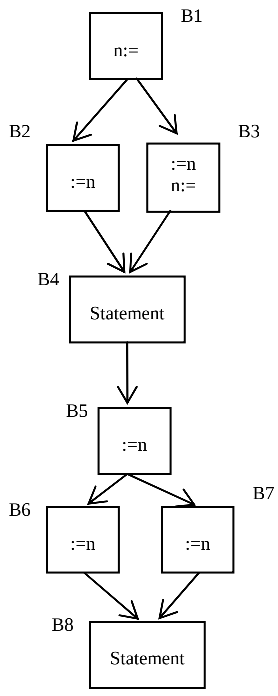


Figure 1. Control Flow Graph for odd_even program.

Table 1. Testing information obtained through data flow analysis.

S.no	**Testcase,t_j**	**Execution path**	**DU-pair(s) in the execution path**
1	t_1	B_1,B_2,B_4,B_5,B_6,B_8	(B_1,B_2) $(B_1,B_5)(B_1,B_6)$
2	t_2	B_1,B_2,B_4,B_5,B_7,B_9	(B_1,B_2) (B_1,B_5) (B_1,B_7)
3	t_3	B_1,B_3,B_4,B_5,B_6,B_8	(B_1,B_3) (B_3,B_5) $B_3,B_6)$
4	t_4	B_1,B_3,B_4,B_5,B_7,B_8	(B_1,B_3) (B_3,B_5) $B_3,B_7)$

Table 2. Test sets generated for the DU pairs.

S.no	**Test Sets, T_i**	**DU-pair**	**Test cases in T_i**
1	T_1	(B_1,B_2)	$\{t_1,t_2\}$
2	T_2	(B_1,B_5)	$\{t_1,t_2, t_3,t_4\}$
3	T_3	(B_1,B_6)	$\{t_1,t_3\}$
4	T_4	(B_1,B_7)	$\{t_2,t_4\}$
5	T_5	(B_1,B_3)	$\{t_3,t_4\}$
6	T_6	(B_3,B_5)	$\{t_3,t_4\}$
7	T_7	(B_3,B_6)	$\{t_3\}$
8	T_8	(B_3,B_7)	$\{t_4\}$

The reduction process for the proposed CBTSR, HGS and BOG algorithms begins with the construction of test case requirement matrix as in Figure 2. In the matrix A, each row represents the requirement r_i and each column the test case t_j.

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2. Initial test case requirement matrix for the sample program.

In the proposed CBTSR algorithm, from the vector *S* row 7 and row 8 have a row count of one. These rows represent singleton test cases with cardinality one that corresponds to DU pair *(*B_3*,* B_6) and (B_3*,* B_7). These DU pairs are represented by test cases t_3 and t_4. The two test cases are selected into the temporary set T_s. The lists: *list_t* and *list_r* are updated by removing these test cases and requirements. The rows 7 and 8 covering the requirements and the columns representing test cases t_3 and t_4 are reset to zero in the test case requirement matrix. Then the function *select_optimal()* is invoked. This determines the row count of the test case requirements matrix, followed by recomputing the values of the vector *'S'*. Then, the maximum value is determined from the vector *'S'*, which is correlated with the requirement and the corresponding test cases satisfying the requirements. The test cases are then added to the temporary set T_{temp} and the process iterates till the vector *'S'* contains values greater than zero. Finally, the values of temporary sets T_s and T_{temp} are combined to generate the representative set consisting of $\{t_1, t_2, t_3, t_4\}$.

In the HGS algorithm the cardinality of all the eight test sets i.e two, four, two, two, two, two, one, one are computed. Then, test sets with cardinality one (singleton test cases) are added to representative set i.e, t_3 and t_4. This was followed by marking the occurance of these test cases in the test sets, T_2, T_3, T_4, T_5, T_6, T_7 and T_8. The algorithm, then proceeds with the next higher cardinality which is two. The only test set available is $\{t_1, t_2\}$. Since, there is a tie involved, the first test case t_1 in the test set T_1 was chosen at random into the representative set.

In the BOG algorithm, the test case requirement matrix is constructed and multiplied by its transposed matrix to generate the multipled matrix. The maximum value of the diagonal element in the multiplied matrix is five. Then, the sum of the elements in the i^{th} row of the multiplied matrix, except for the diagonal element is computed. From this computation the test case with the maximum value is determined as t_3 and place in a list called maxList. Similarly the test case with the minimum value is determined as t_1 and place in a list called minList. Both the maxlist and minlist values are then subjected to intersection operation to determine if there were any common values. As there are no common values, a subroutine to determine the optimal test case is invoked. This subroutine returns the test case to the representative set as t_1. Then, the diagonal elements of the multiplied matrix are updated for unselected test cases. This process iterates and finally the representative set generated consists of $\{t_1,t_3,t_4,t_5\}$.

The results from Table 3 show that when HGS algorithm is used, one DU-pair *(*B_1*,* B_7*)*, which determines if the natural number is odd was not selected in the representative set T_{rs}. Further the test cases selected into the representative set, obtained using BOG algorithm also did not select the DU-pair *(*B_1*,* B_7*)*. Hence, the actual objective to determine whether a positive number is odd or even could only be partially tested using the test cases in the representative set, of both HGS and BOG algorithms Table 3. However, retesting the sample program using the test cases in the representative set T_{rs} of the proposed CBTSR algorithm provides the desired coverage of all DU-pairs Table 3 and also satisfies all the requirements to determine whether a number is odd or even.

Table 3. Representative set obtained for odd/even program.

Algorithm(s)	**|T|**	**|DU|**	**T_{rs}**	**DU-pair(s) Not covered by The representative Set**
HGS	16	8	$\{t_1,t_3,t_4\}$	(B_1,B_7)
BOG			$\{t_1,t_3,t_4\}$	(B_1,B_7)
CBTSR			$\{t_1,t_2,t_3,t_4\}$	None

The next section describes the metrics used to analyse the resultant representative set and analysis of results using the proposed CBTSR and state-of-art algorithms.

4. Experiments and Analysis

An empirical study was conducted to evaluate the proposed CBTSR algorithm and state-of-art algorithms, using ten program units each consisting of 11 to 24 lines of coding that cover a wide range of applications. The program description along with the lines of coding is shown in Table 4. The selection of test cases was done using Rapps and Weyuker data flow criterion [13]. Each program considered for experimentation used DU-pair(s) that were hand-instrumented. All the test suite reduction approaches considered in this work had been implemented using Java.

Table 4. Description of program units.

Program No.	**Program Name**	**LOC**	**#DU-Pairs**
Pgm 1	Odd_Even	14	4
Pgm 2	Num_Digits	14	5
Pgm 3	Calc_cost	12	5
Pgm 4	Compute_1	16	5
Pgm 5	Valid Pin	18	5
Pgm 6	Sum_Digits	13	5
Pgm 7	Max_Val	17	6
Pgm 8	Prime_Num	20	6
Pgm 9	Prod_Discount	19	7
Pgm 10	Triangle_type	14	7

The following test metrics are used to determine the performance of the proposed CBTSR approach and state-of-art algorithms:

- The percentage of Requirement Coverage (RCov) is defined in Equation 6:

$$RCov = \frac{|\ R_{cov}\ |}{|\ R_{tot}\ |} \times 100 \qquad (6)$$

Where, | R_{tot} | denotes the total number of test requirements under consideration during test selection and |R_{cov}| is the number of requirements satisfied by the test cases in the representative set. Higher *RCov* means better requirement coverage by the representative set T_{rs}.

- The percentage of test Suite Size Reduction (SSR) [14, 15] is defined as in Equation 7:

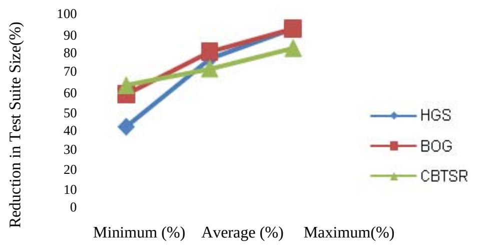
$$SSR = \frac{|T| - |T_{rs}|}{|T|} \times 100 \tag{7}$$

Where, $|T|$ denotes the number of test cases in the original test suite and $|T_{rs}|$ the number of test cases in the representative set. Optimal SSR with better *RCov* is desirable.

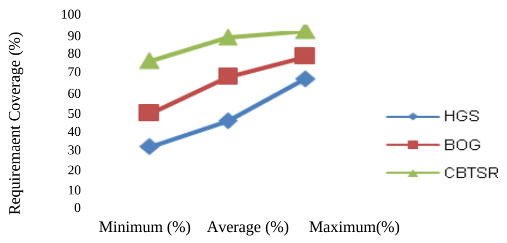
- *Test SSR*: For the subject program considered, the size metric was evaluated for the proposed CBTSR and state-of-art algorithms using Equation 6. From the results obtained as shown graphically in Figure 4, it could be inferred that CBTSR provided minimized test suites ranging between 66.67% to 84.62%. Further the average test SSR obtained for CBTSR was 74.77% which was slightly less than HGS (79.5%) and BOG (82.9%).
- *RCov*: The next metric evaluated was the requirement coverage by the representative sets. The observations made during experimentation showed that the requirement coverage was consistently high. Though the average test SSR was high when using HGS and BOG algorithms, the average *RCov* was marginally less when compared to the proposed CBTSR approach. Thus, the average values of the test metrics considered during performance evaluation for the proposed CBTSR algorithm was better than the state-of-art algorithms Figure 4.

From the experiments conducted, the observations made are summarized as follows:

- HGS algorithm focused on the cardinality of test sets to construct the representative set. In the HGS algorithm the recursive function for test suite minimization was invoked atleast once, to break the ties among equally important test cases. In this algorithm such recursions slowed down the minimization process. Further in the case of a tie between test cases, random test case selection also altered the coverage of requirements.
- When BOG algorithm was used, the order in which test sets were assigned for reduction was important. Further, more computations were also involved in selecting optimum test cases using the helper function. Once all the diagonal values of the matrix became zero the test suite minimization process stopped though there were other non-diagonal values. A major bottleneck of this algorithm were the elaborate computations involved in determining the test cases to be selected into the representative set.
- The proposed CBTSR test suite reduction algorithm removed redundant test cases introduced during program development and retained only the most effective test cases that contributed to the requirement coverage. The test cases that were retained could also provide the maximum requirement coverage. From Figure 4 it is quite apparent that when CBTSR algorithm was used, the test cases in the representative set T_{rs} satisfied more number of requirements and thus subsequently offered better coverage by traversing more DU-paths in a given program.



a) Test suite reduction.



b) Requirement coverage.

Figure 4. Test suite size reduction and requirements coverage using CBTSR, HGS and BOG algorithm.

5. Conclusions

The contributions of this work focus on improving the effectiveness of software testing downstream as unit testing. Although, there has been some existing work in this area, in the present work attempts have been made to reduce the size of the test suite using a simple approach that focuses on the test metrics: Size and requirement coverage. The proposed CBTSR algorithm generated a reduced test suite iteratively using simple matrix operations. The performance evaluations of the proposed CBTSR approach show that:

1. CBTSR algorithm offered consistently better *RCov* than the state-of-art algorithms HGS and BOG.
2. However, the average test SSR of the proposed CBTSR (74.77%) was marginally less than the state-of-art algorithms HGS (79.5%) and BOG (82.9%).

Thus, from the observations made in this work it can be inferred that CBTSR reduces the size of the test suite by retaining test cases that offer maximum *RCov*. In future this work may be extended for another test metric to determine the fault detection capability.

References

[1] Agrawal H., “Efficient Coverage Testing using

Global Dominator Graphs," *in Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, pp. 11-20, 1999.

[2] Black J., Melachrinoudis E., and Kaeli D., "Bi-Criteria Models for All-Uses Test Suite Reduction," *in Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, pp. 106-115, 2004.

[3] Chen T. and Lau M., "Dividing Strategies for the Optimization of a Test Suite," *Information Process Letters*, vol. 60, no. 3, pp. 135-141, 1996.

[4] Errol L. and Brian M., "A Study of Test Coverage Adequacy in the Presence of Stubs," *Journal of Object Technology*, vol. 4, no. 5, pp. 117-137, 2005.

[5] Hutchins M., Foster H., Goradia T., and Ostrand T., "Experiments on the Effectiveness of Dataflow and Control-based Test Adequacy Criteria," *in Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, pp. 191-200, 1994.

[6] Horgan J. and London S., "ATAC: A Data Flow Coverage Testing Tool for C" *in Proceedings of Symposium on Assessment of Quality Software Development Tools*, New Orleans, Louisiana, pp. 2-10, 1992.

[7] Harrold M., Gupta R., and Soffa M., "A Methodology for Controlling the Size of a Test Suite," *ACM Transactions in Software Engineering and Methodology*, vol. 2, no. 3, pp. 270-285, 1993.

[8] Jeffrey D. and Gupta N., "Test Suite Reduction with Selective Redundancy," *in Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, pp. 549-558, 2005.

[9] Jones J. and Harrold M., "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195-209, 2003.

[10] Naresh C., *Software Testing Principles and Practices*, Oxford University Press, India, 2011.

[11] Offutt A., Pan J., and Voas J., "Procedures for Reducing the Size of Coverage-Based Test Sets," *in Proceedings of the 12th International Conference on Testing Computer Software*, Washington, USA, pp. 111-123, 1995.

[12] Paul A. and Jeff O., *Introduction to Software Testing*, Cambridge University Press, 2008.

[13] Rapps S. and Weyuker E., "Selecting Software Test Data using Data Flow Information," *IEEE Transactions on Software Engineering*, vol.11, no. 4, pp. 367-375, 1985.

[14] Saeed P. and Alireza K., "On the Optimization Approach towards Test Suite Minimization," *International Journal of Software Engineering and its Applications*, vol. 4, no. 1, pp. 15-28, 2010.

[15] Scott M. and Atif Memon, "Fault Detection Probability Analysis for Coverage-Based Test Suite Reduction," *in Proceedings of the 21st IEEE International Conference on Software Maintenance*, Paris, France, pp. 335-344, 2007.

[16] Tallam S. and Gupta N., "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *in Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering,* Lisbon, Portugal, pp. 35-42, 2005.

[17] Xue-ying M., Bin-kui S., Cheng-qing Y., "A Genetic Algorithm for Test-suite Reduction," *in Proceedings IEEE International Conference on Systems, Man and Cybernetics,* Hawaii, USA, pp. 133-139, 2005.

[18] Yi W, Manuel O., and Bertrand M., "Is Coverage a Good Measure of Testing Effectiveness?" available at: http://se.inf.ethz.ch/people/wei/papers/is_coverage_a_good_measure_of_testing_effectiveness.pdf, last visited 2010.

Preethi Harris obtained her BE in computer science engineering, ME in software engineering and PhD in Information Communication Engineering. Her teaching and industry experience spans over 15 years. She has published a number of papers in software testing in National and International conferences. She has conducted a Faculty Development Programmes and has also received funds from AICTE, New Delhi to conduct a seminar.

Nedunchezhian Raju is currently working as a Principal. He has more than 20 years of experience in research and teaching. He obtained his BE, ME and PhD degrees in computer science and engineering. Recently, he has obtained AICTE grant for conducting research in data mining. His research interests include knowledge discovery, soft computing, distributed computing and information security. He has published 2 books, 45 research journal papers and 23 conference papers. He is a reviewer for a few international journals/ conferences. He is also the life member of ISTE and ACCS.