Computing and Informatics, Vol. 34, 2015, 1113–1132

TEST SUITE REDUCTION USING HGS BASED HEURISTIC APPROACH

Angelin Gladston, H. Khanna Nehemiah

Ramanujan Computing Centre, Anna University
Chennai – 600 025, Tamilnadu, India
e-mail: nehemiah@annauniv.edu

P. Narayanasamy, A. Kannan

Department of Information Science and Technology, Anna University
Chennai – 600 025, Tamilnadu, India

Abstract. Regression testing is performed throughout the software lifecycle to uncover the faults as early as possible and to ensure that changes do not have any adverse effect in the software that is operational. Test suites once developed are reused and updated frequently. As the software evolves, test cases in the test suite may become redundant. The reason behind this is that the requirements covered by newly added test cases may also be covered by the existing test cases. This redundant nature of test suite increases the cost of executing the same. Further, resource and time constraints impose the necessity to develop techniques to minimize test suites by removing redundant test cases. Few heuristic approaches have been used to solve the test suite minimization problem. Even though solutions exist, still the redundancy of test case remains. In order to solve this problem, this paper proposes two *Harrold-Gupta-Soffa (HGS)* based heuristic algorithms namely, *Non Redundant HGS* and *Enhanced HGS*. The former utilizes the redundant strategy available with *Greedy, Redundant, Essential (GRE)* to get rid of redundancy, whereas the latter selects a test case for higher cardinalities based on overall coverage of unmarked associated testing sets and thus arrives at reduced, non-redundant test suite. The experiments show that the proposed algorithms always select smaller size of test suite, compared to the existing *HGS* heuristics.

Keywords: Test suite, test suite minimization, HGS, GRE, non redundant HGS, enhanced HGS

1 INTRODUCTION

The goal of software testing is to develop bug free software, satisfying functional and non-functional requirements. This process involves executing a program with test cases and recording the actual results in order to uncover faults, thus improving the quality of the product. The test suite comprises a set of test cases, each of which is made up of the input, called test data, and the expected output. Software testers typically maintain a variety of test suites to be used for the software testing. Each test case created either manually or automatically for a software, [1, 2] exercises certain requirements of the software. A requirement is some entity in the software that can be exercised by a test case, it may be either white-box, which deals with the code itself, or black-box, which deals with the specification of the software. Such requirements may include coverage of statements, decisions, definition-use pairs, paths, or coverage of special input values and output values derived from the specification. A test case is often created specifically to cover a certain requirement or a set of requirements, since exercising more unique requirements implies that more of the software is being tested.

As software grows and evolves [3], its test suite also grows. Many test cases will be required to test the new or changed functionality that has been introduced to the software. Since time progresses, some test cases in a test suite are likely to become redundant with respect to a particular coverage criterion, as the specific coverage requirements exercised by those redundant test cases are also exercised by other test cases in the test suite. The property of a test case being redundant is relative to a specific set of coverage requirements. Intuitively, the more the test cases are used, there is a possibility for more requirements to be satisfied. Test suite usually undergoes a process of expansion, as new test cases are inserted into the test suite. This results in a test suite, containing enormous test cases which render regression testing, that is time consuming.

An efficient way to conduct regression testing is to find a minimal subset of test cases which exercises all the test requirements as the original set. A suitable subset could be found during the test case generating or after creating the test suite. Another important point in reducing test suites is the maintenance of them. Whenever there are less test cases in a test suite, tests take the less time for their execution as well as for the maintenance. This process of finding a minimal subset is called test suite reduction or test suite minimization, and the resulting suite is called the representative set.

The goal of test suite minimization is to find a minimal subset of test cases in a suite that exercises the same set of coverage requirements as the original suite. The key idea behind minimization techniques is to remove the test cases in a suite that have become redundant in the suite with respect to the coverage of some particular set of program requirements. Test suite minimization problem as defined by Yoo et al. [4] is, given a test suite T, a set of test requirements r_1, r_2,..., r_n that must be satisfied to provide the desired adequate testing of the program, and subsets T_1, T_2, ..., T_n, one associated with each of the r_i such that any one of the test

cases t_j belonging to T_i satisfies r_i, the problem is to find a representative set, T', of test cases from T that satisfies all r_i. In this paper, two new approaches for the test suite reduction namely, *Non Redundant HGS* algorithm and *Enhanced HGS* algorithm are proposed. These two tailored *HGS* algorithms are aimed at reducing the test suite. They address the redundant test case issue and get rid of redundant test cases.

The remaining of this paper is organized as follows: Section 2 presents an overview of test suite minimization problem and work related to the application of heuristics, *GRE* and *HGS*. Section 3 describes *HGS* and *GRE* algorithms used in this work for evaluation. Section 4 describes the proposed *Non Redundant HGS* algorithm and *Enhanced HGS* algorithm for test suite minimization. Section 5 provides the implementation details. Section 6 reports the results and discussion and Section 7 concludes the work.

2 RELATED WORK

Jeffrey et al. [5] extended the *HGS* heuristic so that certain test cases were selectively retained, based on a secondary set of testing requirements. When a test case was marked as redundant with respect to the first set of testing requirements, then it was checked for redundancy with respect to the second set of testing requirements. If not, the test case was selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used a branch coverage as the first set of testing requirements and all-uses coverage information obtained by data-flow analysis as the second set of testing requirements. Seven programs from the Siemens suite [8], namely, tcas – altitude separation, totinfo – info accumulator, schedule – priority scheduler, schedule2 – priority scheduler, printtokens – lexical analyzer, printtokens2 – lexical analyzer and replace – pattern substitute, Space program, which is an array definition language interpreter along with four *Java* programs, namely, bst, to remove from a binary search tree, avl, to insert into an avl tree, heap, to delete min from a binary heap and sort, to quick sort an array were used. The results showed that, though the fault detection capability was better compared to single-criterion versions of *HGS* heuristic, they produced larger test suites.

Zhong et al. [6] compared four typical test suite reduction techniques. They are *heuristic H, heuristic GRE*, *Genetic* algorithm based approach and *Integer Linear Programming (ILP)* based approach for test suite reduction. The techniques namely *heuristic H*, finds the essential test cases, the *Greedy-Redundant-Essential (GRE)* algorithm, which comprises three strategies: *greedy* strategy which selects test cases that satisfies most of the unsatisfied requirements, the *essential* strategy selects all essential test cases and *1-to-1 redundancy* strategy removes all the 1-to-1 redundant test cases, *Genetic* algorithm, which is based on the concept of natural evolution, where reproduction and selection operations are applied to populations to get optimal solutions and *ILP*, which is based on multi-objective test suite reduction. They

used six $'C'$ programs and reduced their test suites using all four techniques. On an average, the difference in the size of the resulting representative set among these techniques is less than 1 %. To conclude, *heuristic H* should be the first. In *heuristic GRE* and *ILP*-based approach when tie occurs in removing the redundant test cases, a test case was chosen randomly. As a result, some test cases which may be very effective in detecting faults were removed.

Tallam et al. [7] identified that the early selection of test cases made by the *greedy* algorithm can eventually be rendered redundant by the test cases subsequently selected. Hence they extended the *greedy* approach by introducing the delayed *greedy* approach. *Concept lattice*, a hierarchical clustering, based on the relation between test cases and testing requirements was constructed and used for reducing the test cases using a delayed *greedy* approach. Programs in the Siemens test suite and Space program [9] were used for the study. In the empirical evaluation, on an average for a branch coverage, the test suites minimized by this delayed *greedy* were smaller than those minimized by the *greedy* approach and *HGS* in 35 % and 64 % of the cases, respectively. Thus the delayed *greedy* selected a minimal cardinality subset of the test suite.

McMaster et al. [10] proposed a test suite minimization technique based on call-stack coverage. A test suite is represented by a set of unique maximum depth call-stacks. Its minimized test suite is a subset of the original test suite where the execution generates the same set of unique maximum depth call-stacks. Once the call-stack coverage information is collected, the *HGS* heuristic is applied. Studies showed the trade-off between reduction in the test suite size and the loss of fault detection effectiveness.

Lin et al. [11] presented a novel approach called *Reduction with tie-breaking (RTB)* for a test suite reduction. *RTB* approach uses another coverage criterion to break the ties in the minimization process. They integrated the *RTB* approach with *GRE* and *HGS* into *Modified GRE (M_GRE)* and *Modified HGS (M_HGS)* and showed the improvement in the fault detection effectiveness with a negligible increase in the size of the suite. The subject programs were Siemens programs and the Space program. On an average, the suite size reduction of *HGS* and *M_HGS* were 85.2 % and 84.94 % respectively. Similarly, the suite size reduction for *GRE* and *M_GRE* were on an average, 91.83 % and 91.84 % respectively. On the whole, compared with the *GRE* and *HGS*, the proposed approaches had almost the same or better ability in reducing the test suites for the selected subject programs.

Jeya Mala et al. [12] applied Hybrid Genetic Algorithm (HGA) for improving the quality of test cases. In their approach, they selected effective test cases that have higher mutation score and path coverage from a near infinite number of test cases. Hence, the final test set size has been reduced which in turn reduced the total time needed in testing. Gu et al. [13] defined the multi-objective test suite reduction problem under selective-form regression testing and developed the *Hierarchical Alignment of Two-dimensional Spectra (HATS)* algorithm to solve the problem. The *HATS* conforms to the heuristic *greedy* search framework. A weighting factor α, balancing two objectives, namely, to reduce the test suite size and to

avoid coverage of irrelevant test requirements, is defined. The experiments showed that with proper setting of the factor α, HATS reduced the test suite size with a cut down in the coverage of irrelevant requirements and less compromise in the fault detection ability. To conclude, among the various algorithms, namely *HGS*, *GRE*, *heuristic H*, *Greedy*, *Genetic* and *ILP* based approach being utilized for test suite reduction, *HGS* had been widely used. The utilization of *HGS* and the existence of redundant test cases are taken up as issues to be addressed in this work.

3 EXPERIMENTAL DESIGN

The *HGS* and *GRE* algorithms as mentioned in the related work section, have been used to solve the test suite minimization problem. Though *HGS* [11, 15] minimizes the test suite, still there is redundancy [7]. To overcome this problem we devised two approaches, and the existing *HGS* and *GRE* algorithms described in this section are used for the evaluation.

3.1 HGS Algorithm

The input to the *HGS* algorithm is the requirements matrix. Using the requirements matrix the associated testing sets are generated. The algorithm, first considers the associated testing sets of cardinality one, and places the entire test cases belonging to the associated testing sets of cardinality one, into the representative set. Then the associated testing sets of cardinality two are considered, and the test case, which occurs the maximum number of times among all associated testing sets of cardinality two, is added into the representative set. Whenever a test case is selected the corresponding test case and the satisfied associated testing sets are marked. This process is repeated up to a higher cardinality until all the requirements are satisfied. When examining the cardinality for selecting a test case, a tie may occur because several test cases occur more than once. In that case, test cases are selected arbitrarily. The same is illustrated in Section 4.3.

3.2 GRE Algorithm

In *GRE*, three strategies – namely Essential, Redundant and Greedy – are applied until all the requirements are satisfied. A test case is regarded as essential if a requirement is satisfied by that test case alone. In contrast, a test case is said to be 1-to-1 redundant if there exists a test case, such that the set of requirements covered by a test case is also covered by another test case. First, the Essential strategy selects the test cases which are essential and adds them to the representative set. Then, the Redundant strategy reduces the test cases which are redundant. Finally, when there are requirements yet to be covered, the Greedy strategy, which selects the test cases that satisfy most of those requirements, is applied.

4 PROPOSED ALGORITHMS

The two proposed algorithms are *Non Redundant HGS* and *Enhanced HGS*. We integrated the redundant strategy of *GRE* into *HGS*, called *Non Redundant HGS* to get rid of redundant test cases in the minimized test suite. Further we have tailored *HGS* into *Enhanced HGS* which selects out reduced set of test cases, considering the coverage of test cases, rather than based on maximum number of occurrence among the chosen cardinality. Coverage of test case is the number of times the test case occurs among all unmarked associated testing sets irrespective of their cardinality. Thus, by considering coverage, *Enhanced HGS* results in non-redundant, minimized test suite.

4.1 Non Redundant HGS Algorithm: Proposed Approach I

Non Redundant HGS algorithm takes the requirements matrix as input. Associated testing sets and their cardinality which is the number of test cases in associated testing set are computed and used. When examining the test cases of a particular cardinality, a tie may occur because several test cases occur in more than one associated testing sets, and in that case test cases are selected arbitrarily. *Non Redundant HGS* selects test cases using the same approach as that of *HGS*. The redundant test cases [7] are removed in two steps, namely step 4 and step 6 as illustrated in the Algorithm 1. Step 4 is realized in line numbers 17 to 21. Step 6 is realized in line numbers 40 to 46. The redundant strategy of *GRE* is utilized and the steps involved in *Non Redundant HGS* are as follows:

Step 1: Initialization.

Step 2: Place all test cases of associated testing sets of cardinality one in the representative set.

Step 3: Mark the corresponding testing sets.

Step 4: When the requirement set of a test case is a subset of another test case, then remove it from the associated testing sets.

Step 5: Compute *RS* according to the heuristic for sets of higher cardinality.

Step 6: Check for redundant test cases and remove from representative set.

Step 7: When all associated testing sets are marked, *RS* will have the reduced test cases, and the algorithm terminates.

Non Redundant HGS takes as input the associated testing sets in the line 1 and the requirement sets in the line 2, both derived from the requirements matrix. Maximum_Cardinality and Current_Cardinality are the variables declared in the line 4 for considering the cardinality one by one and the algorithm returns the selected test cases in the representative set, RS as in the line 3. List in the line 5 supplies the list of t_i belonging to associated testing sets of chosen cardinality to the SelectTest function in the Algorithm 2, which returns the Chosen_Test that can be

one of $t_1, t_2, \ldots, t_n$ as in the line 6. The line 7 declares Marked as a boolean array initialized as false and the line 8 declares May_Reduce as a boolean variable. Two functions, namely Max(), which returns the maximum of a set of numbers and Cardinality(), which returns the cardinality are introduced in the lines 9 and 10. Step 1, initialization assigns Maximum_Cardinality with Max(Cardinality(T_i)), which returns the maximum cardinality among the cardinalities of all $T_i's$ in the line 11. Step 2 places all test cases of associated testing sets of cardinality one in the representative set and is realized in the line 12 of the Algorithm 1. It adds all single cardinality T_i into the representative set. Step 3 marks the corresponding testing sets in the lines 13 to 15. All T_i in representative set are marked so that unmarked T_i can be considered next. Current cardinality under consideration is the one which is set in the line 16. Step 4 in the lines 17 to 21 removes the test case from the associated testing sets, when the requirement set of a test case is a subset of another test case. The line 17 finds a requirement set which is a subset of any other requirement set R_j and the line 19 removes the redundant test cases from the associated testing sets.

Step 5 computes RS according to the heuristic for sets of higher cardinality as realized in the lines 22 to 39. Current cardinality is incremented in the line 23. All t_i belonging to all unmarked T_i of current cardinality are included in the List using the lines 24 and 25. The SelectTest function in the Algorithm 2, which implements the selection method of a test case from all the test cases belonging to the current cardinality, is invoked in the line 27. This function selects the next t_i to be included in RS as shown in the line 28. The function from the lines 1 to 15 in the Algorithm 2, finds the number of times t_i occurs in associated testing sets of chosen cardinality, called count as shown in the lines 3 and 4. The line 6 chooses the test cases, which occur maximum number of times in the associated testing sets of chosen cardinality and puts them in a TestList. It finds the number of test cases in the TestList and when there is only one test case having maximum cardinality then that test case is added to the representative set as in the lines 7 and 8. When there are many test cases having maximum cardinality, then any one test case, as in the lines 10 and 11, is added to the representative set. Then, as in the line 13, the next higher cardinality is taken into consideration. The line 28 in the Algorithm 1 adds the chosen test case into representative set.

Step 5 utilizes May_Reduce to indicate that the current cardinality has been considered by setting it as false. All associated testing sets having a chosen test are marked. May_Reduce is used in the line 36 and if it is true, it finds the maximum cardinality among those of all unmarked associated testing sets. This gets repeated until Current_Cardinality becomes Maximum_Cardinality. Step 6 checks for redundant test cases in the representative set, and removes them from the representative set. Here S is the representative set excluding t_i as shown in the line 41 and R_s is the requirement set of S as in the line 42. The lines 44 and 45 check for redundant test cases in the representative set, and remove the redundant test cases from the representative set. When all associated testing sets are marked, RS will have the reduced test cases, and the algorithm terminates in Step 7.

```
1  Input Data: T_1, T_2, ..., T_n: associated testing sets for r_1, r_2, ..., r_n
         respectively, containing test cases from t_1, t_2, ..., t_n
2  Data: R_1, R_2, ..., R_n: set of requirements satisfied by t_1, t_2, ..., t_n
3  Result: RS: a representative set of test cases selected from T_1, T_2, ..., T_n
4  Maximum_Cardinality, Current_Cardinality: 1..n
5  List: list of t_i belonging to associated testing sets of chosen cardinality
6  Chosen_Test: one of t_1, t_2, ..., t_n
7  Marked: array [1..n] of boolean, initially false
8  May_Reduce: boolean
9  Max(): returns the maximum of a set of numbers
10 Cardinality(): returns the cardinality
11 Maximum_Cardinality ← Max(Cardinality(T_i))
12 RS ← RS ∪ T_i, for all Cardinality(T_i) = 1
13 foreach T_i such that T_i ∩ RS ≠ Φ do
14 |  Marked[i] ← true
15 end
16 Current_Cardinality ← 1
17 foreach R_i such that R_i ∩ R_j = R_i do
18 |  foreach T_i where t_i ∈ T_i do
19 |  |  T_i ← T_i − t_i
20 |  end
21 end
22 while Current_Cardinality ≤ Maximum_Cardinality do
23 |  Current_Cardinality ← Current_Cardinality + 1
24 |  while there are T_i such that Cardinality(T_i) = Current_Cardinality and
   |    not Marked[i] do
25 |  |  List ← all t_i ∈ T_i where Cardinality(T_i) = Current_Cardinality and
   |  |    not Marked[i]
26 |  end
27 |  Chosen_Test ← SelectTest (Current_Cardinality, List)
28 |  RS ← RS ∪ {Chosen_Test}
29 |  May_Reduce ← False
30 |  foreach T_i such that Chosen_Test ∈ T_i do
31 |  |  Marked[i] ← true
32 |  |  if Cardinality(T_i) = Maximum_Cardinality then
33 |  |  |  May_Reduce ← true
34 |  |  end
35 |  end
36 |  if May_Reduce then
37 |  |  Maximum_Cardinality ← Max(Cardinality(T_i)), for all i where
   |  |    Marked[i] ← false
38 |  end
39 end
```

```
40 foreach t_i ∈ RS do
41 |   S = RS − t_i
42 |   R_s = R_s ∪ R_k, ∀t_k ∈ S
43 end
44 if R_i ∩ R_s = R_i then
45 |   RS ← RS − t_i
46 end
```

Algorithm 1: Non redundant HGS algorithm

```
1  Function SelectTest(Size, List)
2  Count array[1..n]
3  foreach t_i in List do
4  |   compute Count[t_i], the number of unmarked T'_i s of cardinality Size
   |     containing t_i
5  end
6  Construct TestList consisting of tests from List for which Count[i] is
     maximum
7  if Cardinality(TestList) = 1 then
8  |   return(the test case in TestList)
9  else
10 |   if Size = Maximum_Cardinality then
11 |   |   return(any test case in TestList)
12 |   else
13 |   |   return(SelectTest(Size + 1, TestList))
14 |   end
15 end
```

Algorithm 2: Function SelectTest for Non Redundant HGS

4.2 Enhanced HGS Algorithm: Proposed Approach II

In *Enhanced HGS* algorithm the selection of test case for all higher cardinalities is altered. Rather than choosing a test case based on its maximum number of occurrences among all the associated testing sets belonging to that cardinality, we confine to the selection of test cases based on its maximum occurrence in unmarked associated testing sets called the coverage. Coverage $[t_i]$ returns the count of unmarked associated testing sets having t_i irrespective of their cardinality, which is the number of yet uncovered requirements covered by t_i because each associated testing set corresponds to a requirement. Thus coverage of t_i is its coverage among the unmarked associated testing sets. The steps involved in *Enhanced HGS* as realized in the Algorithm 3 are as follows:

Step 1: Initialization.

Step 2: Place all test cases of associated testing sets of cardinality one in the representative set.

Step 3: Mark the corresponding testing sets.

Step 4: Compute RS according to the new heuristic for sets of higher cardinality.

Step 5: When all associated testing sets are marked, RS gives the reduced test cases and the algorithm terminates.

Steps 1, 2 and 3 of Enhanced HGS algorithm realized in the Algorithm 3 are the same as that of Non Redundant HGS. Step 4 computes representative set according to the new heuristic for sets of higher cardinality based on Coverage. The new SelectTest function in the Algorithm 4 realized in the the lines 1 to 15 implements the selection method of a test case, based on the maximum coverage of test cases irrespective of their cardinality. This function selects the next t_i to be included in the representative set, based on the maximum occurrence of t_i in all the unmarked associated testing sets irrespective of their cardinality, called a coverage as realized in the line 6. Coverage function finds the number of times t_i occurs among all unmarked associated testing sets irrespective of their cardinality as in the lines 3 and 4. TestList is constructed by finding and putting test cases with a maximum coverage. When all associated testing sets are marked, RS gives the reduced test cases and the algorithm terminates in step 5.

4.3 Illustration

To illustrate *HGS, Non Redundant HGS* and *Enhanced HGS*, consider the requirements matrix in Table 1 where the seven rows represent seven test cases and seven columns represent seven requirements. The requirements satisfied by each test case are marked. Table 2 illustrates the test cases and their corresponding requirement set. R_i represents the requirement set of test case t_i. For example, $\{r_1, r_2\}$ is the requirement set of test case t_1. Table 3 illustrates the requirements and their corresponding associated testing sets. T_i represents the associated testing set of requirement r_i. For example, $\{t_1, t_5\}$ is the associated testing set of requirement r_1.

$t_i \backslash r_i$	r_1	r_2	r_3	r_4	r_5	r_6	r_7
t_1	*	*					
t_2			*		*		
t_3			*	*			
t_4			*	*			
t_5	*					*	
t_6		*			*		*
t_7				*	*		

Table 1. Requirements matrix

Thus, an associated testing set is the set of test cases, which covers a given requirement and the cardinality of an associated testing set is the number of test

```
1  Input Data: T_1, T_2, ..., T_n: associated testing sets for r_1, r_2, ..., r_n
        respectively, containing test cases from t_1, t_2, ..., t_n
2  Data: R_1, R_2, ..., R_n: set of requirements satisfied by t_1, t_2, ..., t_n
3  Result: RS: a representative set of test cases selected from T_1, T_2, ..., T_n
4  Maximum_Cardinality, Current_Cardinality: 1..n
5  List: list of t_i belonging to associated testing sets of chosen cardinality
6  Chosen_Test: one of t_1, t_2, ..., t_n
7  Marked: array [1..n] of boolean, initially false
8  May_Reduce: boolean
9  Max(): returns the maximum of a set of numbers
10 Cardinality(): returns the cardinality
11 Maximum_Cardinality ← Max(Cardinality(T_i))
12 RS ← RS ∪ T_i, for all Cardinality(T_i) = 1
13 foreach T_i such that T_i ∩ RS ≠ Φ do
14 |   Marked[i] ← true
15 end
16 Current_Cardinality ← 1
17 while Current_Cardinality ≤ Maximum_Cardinality do
18 |   Current_Cardinality ← Current_Cardinality + 1
19 |   while there are T_i such that Cardinality(T_i) = Current_Cardinality and
   |     not Marked[i] do
20 |   |   List ← all t_i ∈ T_i where Cardinality(T_i) = Current_Cardinality and
   |   |     not Marked[i]
21 |   end
22 |   Chosen_Test ← SelectTest (Current_Cardinality, List)
23 |   RS ← RS ∪ {Chosen_Test}
24 |   May_Reduce ← False
25 |   foreach T_i such that Chosen_Test ∈ T_i do
26 |   |   Marked[i] ← true
27 |   |   if Cardinality(T_i) = Maximum_Cardinality then
28 |   |   |   May_Reduce ← true
29 |   |   end
30 |   end
31 |   if May_Reduce then
32 |   |   Maximum_Cardinality ← Max(Cardinality(T_i)), for all i where
   |   |     Marked[i] ← false
33 |   end
34 end
```

Algorithm 3: Enhanced HGS algorithm

cases in the associated testing set. For example, $T_1 = \{t_1, t_5\}$ is the associated testing set of requirement r_1. Associated testing set T_1 has two test cases, hence the cardinality of T_1 is two. Each column in the requirements matrix in Table 1

```
1  Function SelectTest(Size, List)
2  Coverage array[1..n]
3  foreach t_i in List do
4  |  compute Coverage[t_i], the number of unmarked T_i's containing t_i
   |    irrespective of their cardinality.
5  end
6  Construct TestList consisting of tests from List where Coverage[i] is
     maximum
7  if Cardinality(TestList) = 1 then
8  |  return(the test case in TestList)
9  else
10 |  if Size = Maximum_Cardinality then
11 |  |  return(any test case in TestList)
12 |  else
13 |  |  return(SelectTest(Size + 1, TestList))
14 |  end
15 end
```

Algorithm 4: Function SelectTest for Enhanced HGS

represents a requirement. All test cases covering a requirement are marked with one and others with zero. Similarly, for all requirements, the corresponding test cases are marked with ones and others with zeros. Cardinality is computed from this requirements matrix. Cardinality has to be computed for every associated testing set, that is, for every set of test cases covering a particular requirement, which means for every column in the requirements matrix. Test cases covering a particular requirement are marked with one. Hence, the number of entries in a column is the cardinality of the corresponding associated testing set. Similarly, number of entries in every column is counted and thus the cardinality of all associated testing sets is arrived at.

In the requirements matrix in Table 1, there are seven columns, each belonging to one associated testing set, because it has entries for the corresponding test cases covered by that particular requirement, for which the associated testing set belongs. First column and second column have two entries, hence the cardinality of associated testing sets, T_1 and T_2, is two. Similarly, third, fourth and fifth columns have three entries each, hence the cardinality of associated testing sets, T_3, T_4 and T_5, is three. Sixth column and seventh column have one entry each, hence the cardinality of associated testing sets, T_6 and T_7, is one. Basically *HGS* algorithm as well as the proposed algorithms, *Non Redundant HGS* algorithm and *Enhanced HGS* algorithm selects test cases using heuristic based on cardinality. The algorithms consider the associated testing sets of a given cardinality and the test case that participates in an unmarked associated testing set is selected. The selected test cases are added to a set called representative set, *RS*.

Test Case t_i	Requirement Set R_i
t_1	$\{r_1, r_2\}$
t_2	$\{r_3, r_5\}$
t_3	$\{r_3, r_4\}$
t_4	$\{r_3, r_4\}$
t_5	$\{r_1, r_6\}$
t_6	$\{r_2, r_5, r_7\}$
t_7	$\{r_4, r_5\}$

Table 2. Test cases and their requirement sets

Requirement r_i	Associated Testing Set T_i
r_1	$\{t_1, t_5\}$
r_2	$\{t_1, t_6\}$
r_3	$\{t_2, t_3, t_4\}$
r_4	$\{t_3, t_4, t_7\}$
r_5	$\{t_2, t_6, t_7\}$
r_6	$\{t_5\}$
r_7	$\{t_6\}$

Table 3. Requirements and their associated testing sets

HGS algorithm first considers associated testing sets of cardinality one. The associated testing sets of requirements, r_6 and r_7 that is, T_6 and T_7 have cardinality one. Hence representative set, *RS* becomes $\{t_5, t_6\}$ and the corresponding test case and associated testing sets are marked. Next, cardinality two is considered and the associated testing sets of requirements r_1 and r_2 that is, T_1 and T_2 come under this category. Among these, the test case which occurs maximum number of times, among the associated testing sets of cardinality two, is selected and added to *RS*, provided the corresponding associated testing sets are unmarked. But T_1 and T_2 are already marked. Thus *RS* remains the same $\{t_5, t_6\}$. Cardinality three is considered next. Associated testing sets of cardinality three are T_3, T_4 and T_5 which correspond to the requirements r_3, r_4 and r_5. Among these test cases, t_2, t_3, t_4 and t_7 are the test cases which occur a maximum number of times among the selected associated testing sets. Here a tie occurs, and any test case can be chosen arbitrarily, considering the first test case is selected and *RS* becomes $\{t_5, t_6, t_2\}$. There are no more associated testing sets with higher cardinalities but T_4 is not yet marked. Hence a test case which covers T_4 has to be added. The test cases, which participate in T_4 are t_3, t_4 and t_7. Among these, the first one, t_3 is added to *RS*, making it $\{t_5, t_6, t_2, t_3\}$ as shown in Table 5. In this *RS*, t_2 is a redundant test case. The test case t_2 covers the requirement r_3 and r_5. The test case t_3 also covers the requirement r_3 and the test case t_6 also covers the requirement r_5. Hence, t_2 is a redundant test case with respect to the testing requirement. This is the shortcoming with *HGS*.

Non Redundant HGS uses the same selection process incorporated in the SelectTest function as that of *HGS*, but checks for redundant test cases in two steps.

First, in step 4 of *Non Redundant HGS* redundant test cases are removed from the associated testing sets. The requirement sets of any two test cases are compared with each other. If they match, then the later test case is removed from all associated testing sets where it is participating. Here, the requirement sets of test case t_3 and t_4 are the same. Hence, t_4 is a redundant test case and t_4 is removed from all associated testing sets where it is participating. Thereby t_4 is removed from the associated testing sets T_3 and T_4. Second, in step 6 of *Non Redundant HGS*, redundant test cases are removed from the *RS*. In the representative set, RS, t_2 has become a redundant test case because of the test cases t_3 and t_6. It is in this step 6, t_2 gets removed and *RS* becomes $\{t_5, t_6, t_3\}$ as shown in Table 5. Thus, *Non Redundant HGS* differs from *HGS*. The steps involved in the three algorithms are illustrated in Table 4.

Step	HGS	Non Redundant HGS	Enhanced HGS
Cardinality 1	$\{t_5, t_6\}$	$\{t_5, t_6\}$	$\{t_5, t_6\}$
Cardinality 2	$\{t_5, t_6\}$	$\{t_5, t_6\}$	$\{t_5, t_6\}$
Cardinality 3	$\{t_5, t_6, t_2\}$	$\{t_5, t_6, t_2\}$	$\{t_5, t_6, t_3\}$
For T_4	$\{t_5, t_6, t_2, t_3\}$	$\{t_5, t_6, t_2, t_3\}$	
Redundancy check		$\{t_5, t_6, t_3\}$	

Table 4. Step-by-step illustration

Enhanced HGS algorithm first considers associated testing sets of cardinality one and all test cases belonging to associated testing sets of cardinality one is included in *RS*. Thus *RS* becomes $\{t_5, t_6\}$. Next, associated testing sets of cardinality two is considered. Among the test cases belonging to associated testing sets of cardinality two, the test case which occurs a maximum number of times, among all unmarked associated testing sets irrespective of their cardinality, that is, the test case which occurs in a maximum number of unmarked associated testing sets, is added to *RS*. Here, associated testing sets of cardinality two are already marked. Hence, *RS* remains the same $\{t_5, t_6\}$. In the next step, among the associated testing sets with cardinality three, that is T_3, T_4 and T_5, both T_3 and T_4 are unmarked and the corresponding test cases are t_2, t_3, t_4 and t_7. Among these, t_3 and t_4 are having maximum coverage of two, i.e. they participate in two unmarked associated testing sets. Hence, *RS* becomes $\{t_5, t_6, t_3\}$ as shown in Table 5. Here a tie occurs and any test case can be chosen arbitrarily. Even when t_4 is chosen, the algorithm will arrive at a reduced, non-redundant *RS*. Thus *Enhanced HGS* differs from *HGS* in selecting a test case. Further, it results in a *RS* which is not redundant.

Algorithm	Representative set
HGS	$\{t_5, t_6, t_2, t_3\}$
Non Redundant HGS	$\{t_5, t_6, t_3\}$
Enhanced HGS	$\{t_5, t_6, t_3\}$

Table 5. Representative set using various algorithms

5 EXPERIMENTS

A PL/SQL procedure named *FIRST_RUN*, used in a *Payroll Management System* that was developed using Oracle 10g was used for the study. The system consists of thirty three relations. The *FIRST_RUN* procedure populates the earnings_trial_run relation and deductions_trial_run relation with new values. The earnings_trial_run relation is meant for storing the earnings of employees applicable in a month and deductions_trial_run relation is meant for storing the deductions of employees applicable in a month. These two relations are used to make a report generation easy. The earnings_trial_run relation is populated with earnings from earnings relation. The deductions_trial_run relation is populated with various deductions that are drawn from the sixteen relations: employee_gpf which stores employee provident fund details, employee_cps which stores employee contributory pension scheme details, deduction_specific which covers deduction details of additional house rent, electricity charge, cooperative society, vehicle maintenance, as well as employees' recreation club, variable_ded_std which captures details of a family benefit fund, mediclaim, government health insurance scheme, as well as the professional tax, cps_recovery which stores employee contributory pension scheme recovery details, spf which stores special provident fund details, court_recovery, income_tax_deduction, gpf_loan, house_loan, loan_sanction, rop which stores recovery of over payment details, bank_loan, licpolicy, pli which stores postal life insurance policy details and rec_deposit which stores details of recurring deposits.

The details of the program used for the study are given in Table 6. In the study, testing requirements for the white-box testing criteria, namely a branch coverage was used for reducing the test suites. Five different branch coverage adequate test suites were created to allow varying levels of redundancy. Requirements matrix for the test cases was generated manually, which maps the test cases with the corresponding requirements covered by it.

Program	Lines of Code	Average Test Suite Size
FIRST_RUN	512	142

Table 6. Program used for the study

Requirements matrix for the *FIRST_RUN* has 142 rows, that are test cases and 92 columns, which are requirements. The test cases are reduced using *Non Redundant HGS* algorithm and *Enhanced HGS* algorithm with respect to a branch coverage. *GRE* presented by Chen et al. [14] for test suite minimization and *HGS* presented by Harrold et al. [15] are used for evaluating the *Non Redundant HGS* algorithm and *Enhanced HGS* algorithm.

6 RESULT ANALYSIS

The parameter Suite Size Reduction (SSR) given by $SSR = |T| - |T_{min}|/|T|$, where $|T|$ is the number of test cases in the original test suite and $|T_{min}|$ is the number

of test cases in the minimized or reduced test suite, is used for the analysis. The proposed algorithms are devised to solve the redundancy problem in *HGS* heuristic technique. Table 7 as well as the illustration in Section 4.3 clearly depict that both *Non Redundant HGS* and *Enhanced HGS* improve upon existing *HGS* algorithm making it more effective in arriving at a non-redundant reduced test suite.

Test Suite	Algorithm	Test Cases		% of SSR
		Before Reduction	After Reduction	
1	GRE	142	32	77.46
	HGS	142	36	74.65
	Non Redundant HGS	142	31	78.17
	Enhanced HGS	142	31	78.17
2	GRE	142	41	71.13
	HGS	142	47	66.90
	Non Redundant HGS	142	42	70.42
	Enhanced HGS	142	41	71.13
3	GRE	142	38	73.24
	HGS	142	45	68.31
	Non Redundant HGS	142	45	68.31
	Enhanced HGS	142	39	72.54
4	GRE	142	38	73.24
	HGS	142	45	68.31
	Non Redundant HGS	142	38	73.24
	Enhanced HGS	142	38	73.24
5	GRE	142	39	72.53
	HGS	142	47	66.90
	Non Redundant HGS	142	44	69.01
	Enhanced HGS	142	38	73.23

Table 7. Test suite reduction using *GRE*, *HGS*, *Non Redundant HGS* and *Enhanced HGS* for five different test suites

Table 7 shows the test cases before and after the reduction using *GRE, HGS, Non Redundant HGS* and *Enhanced HGS* for the five different test suites. Each execution result is that of a different test suite. The nature of the test suite affects the performance of the algorithm. In execution 3, for that particular third test suite *GRE* reduces one more test case, achieving 38 whereas *Enhanced HGS* achieves 39. But for the test suite 1 and 5 *GRE* has reduced to 32 and 39, respectively, whereas *Enhanced HGS* has achieved 31 and 38, respectively. Hence, on an average, SSR rate of *Enhanced HGS* is better than that of *GRE.*

Table 8 shows the average size of each original test suite and the average size of each reduced test suite and the average SSR of all four approaches, namely *GRE, HGS, Non Redundant HGS* and *Enhanced HGS*. Average suite size reduction percentage of *Non Redundant HGS* is 71.82 % and that of *Redundant HGS* is 73.66 %, which are better when compared to that of *HGS*, 69.01 %. Moreover, the performance of *Enhanced HGS* and *Non Redundant HGS* have improved a lot compared

Algorithm	Test Cases		% of SSR
	Before Reduction	After Reduction	
GRE	142	38	73.51
HGS	142	44	69.01
Non Redundant HGS	142	40	71.82
Enhanced HGS	142	37	73.66

Table 8. Average experimental results for suite size reduction

with that of original *HGS* making it more effective by removing redundant test cases. Figure 1 shows that, compared to *HGS*, both the proposed algorithms, *Non Redundant HGS* and *Enhanced HGS* always produce much reduced test suite. Moreover, the problem cited with *HGS*, i.e. the existence of redundant test cases, is removed in *Non Redundant HGS* as well as in *Enhanced HGS*. In *Non Redundant HGS*, redundant test cases are removed after selection and *Enhanced HGS* gets rid of redundant test cases while selecting test case itself. Thus the proposed techniques significantly reduce the test suite size and outperform the existing *HGS* technique by resulting in a reduced non-redundant test suite.

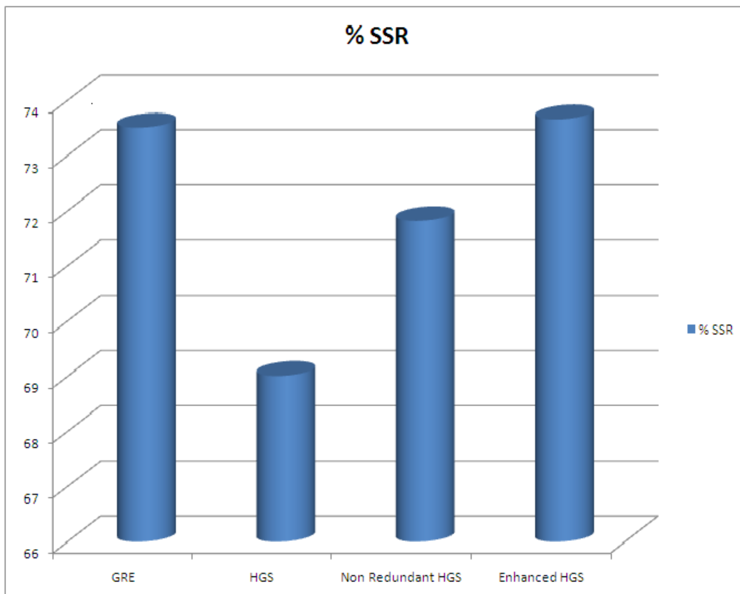

Figure 1. Percentage of suite size reduction vs. reduction technique

7 CONCLUSION

The *Enhanced HGS* and *Non Redundant HGS* algorithms select a minimal subset of a test suite that covers all the requirements covered by the original test suite. This minimal subset is much smaller compared to the existing *HGS*. Further, these techniques improve upon the *HGS* heuristics by removing redundancy in the selected test cases. In the experiments, the percentage of the suite size reduction obtained for *Enhanced HGS* and *Non Redundant HGS* shows that they consistently produce a smaller test suite compared to the existing *HGS* heuristics. The results show that the redundancies in the representative set available in the existing *HGS* heuristic are removed by the proposed approaches and the suite size reduction rate is increased.

REFERENCES

[1] Bulbul, H. I.—Bakir, T.: XML-Based Automatic Test Data Generation. Computing and Informatics, Vol. 27, 2008, No. 4, pp. 681–698.

[2] Khamis, A. M.—Girgis, M. R.—Ghiduk, A. S.: Automatic Software Test Data Generation for Spanning Sets Coverage Using Genetic Algorithms. Computing and Informatics, Vol. 26, 2007, No. 4, pp. 383–401.

[3] Tun, T. T.—Trew, T.—Jackson, M.—Laney, R.—Nuseibeh, B.: Specifying Features of an Evolving Software System. Software – Practice and Experience, Vol. 39, 2009, pp. 973–1002, DOI: 10.1002/spe.923.

[4] Yoo, S.—Harman, M.: Regression Testing Minimization, Selection and Prioritization: A Survey. Software Testing, Verification and Reliability, Vol. 22, 2012, No. 2, pp. 67–120, DOI: 10.1002/stvr.430.

[5] Jeffrey, D.—Gupta, N.: Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction. IEEE Transactions on Software Engineering, Vol. 33, 2007, No. 2, pp. 108–123, DOI: 10.1109/TSE.2007.18.

[6] Zhong, H.—Zhang, L.—Mei, H.: An Experimental Study of Four Typical Test Suite Reduction Techniques. Journal of Information and Software Technology, Vol. 50, 2008, No. 6, pp. 534–546, DOI: 10.1016/j.infsof.2007.06.003.

[7] Tallam, S.—Gupta, N.: A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. ACM SIGSOFT Software Engineering Notes, Vol. 31, 2006, No. 1, pp. 35–42. DOI: 10.1145/1108768.1108802.

[8] Hutchins, M.—Foster, H.—Goradia, T.—Ostrand, T.: Experiments on the Effectiveness of Dataflow- and Control Flow-Based Test Adequacy Criteria. Sixteenth International Conference on Software Engineering (ICSE '94), Italy, 1994, pp. 191–200, DOI: 10.1109/ICSE.1994.296778.

[9] Vokolos, F. I.—Frankl, P. G.: Empirical Evaluation of the Textual Differencing Regression Testing Technique. Fourteenth International Conference on Software Maintenance, USA, 1998, pp. 44–53, DOI: 10.1109/ICSM.1998.738488.

[10] McMaster, S.—Memon, A. M.: Call-Stack Coverage for GUI Test Suite Reduction. IEEE Transactions on Software Engineering, Vol. 34, 2008, No. 1, pp. 99–115, DOI: 10.1109/TSE.2007.70756.

[11] Lin, J.-W.—Huang, C.-Y.: Analysis of Test Suite Reduction with Enhanced Tie-Breaking Techniques. Journal of Information and Software Technology, Vol. 51, 2009, No. 4, pp. 679–690, DOI: 10.1016/j.infsof.2008.11.004.

[12] Jeya Mala, D.—Ruby, E.—Mohan, V.: A Hybrid Test Optimization Framework – Coupling Genetic Algorithm with Local Search Technique. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 133–164.

[13] Gu, Q.—Tang, B.—Chen, D.-X.: A Test Suite Reduction Technique for Partial Coverage of Test Requirements. Chinese Journal of Computers, Vol. 34, 2011, No. 5, pp. 879–888, DOI: 10.3724/SP.J.1016.2011.00879.

[14] Chen, T. Y.—Lau, M. F.: A New Heuristic for Test Suite Reduction. Journal of Information and Software Technology, Vol. 40, 1998, No. 5, pp. 347–354, DOI: 10.1016/S0950-5849(98)00050-0.

[15] Harrold, M. J.—Gupta, R.—Soffa, M. L.: A Methodology for Controlling the Size of a Test Suite. ACM Transactions on Software Engineering and Methodology, Vol. 2, 1993, No. 3, pp. 270–285, DOI: 10.1145/152388.152391.

Angelin Gladston is a Research Scholar in Ramanujan Computing Centre, Anna University, Chennai, Tamilnadu, India. She is working as Assistant Professor at the Department of Computer Science and Engineering, Anna University, Chennai. Her research interests include software engineering, software testing and data mining.

H. Khanna Nehemiah is working as Associate Professor in Ramanujan Computing Centre, Anna University, Chennai, Tamilnadu, India. His research interests include software engineering, databases, data mining and medical image processing.

P. Narayanasamy is working as Professor at the Department of Information Science and Technology, Anna University, Chennai, Tamilnadu, India. His research interests include networks, mobile computing and software engineering.

A. Kannan is working as Professor in the Department of Information Science and Technology, Anna University, Chennai, Tamilnadu, India. His research interests include software engineering, databases, data mining and artificial intelligence.