International Journal of Software Engineering and Its Applications
Vol. 4, No. 1, January 2010

On the Optimization Approach towards Test Suite Minimization

Saeed Parsa and Alireza Khalilian

School of Computer Engineering
Iran University of Science and Technology, Tehran, Iran
parsa@iust.ac.ir, khalilian@comp.iust.ac.ir

Abstract

Regression testing is a critical activity which occurs during the maintenance stage of the software lifecycle. However, it requires large amounts of test cases to assure the attainment of a certain degree of quality. As a result, test suite sizes may grow significantly. To address this issue, Test Suite Reduction techniques have been proposed. However, suite size reduction may lead to significant loss of fault detection efficacy. To deal with this problem, a greedy algorithm is presented in this paper. This algorithm attempts to select a test case which satisfies the maximum number of testing requirements while having minimum overlap in requirements coverage with other test cases. In order to evaluate the proposed algorithm, experiments have been conducted on the Siemens suite and the Space program. The results demonstrate the effectiveness of the proposed algorithm by retaining the fault detection capability of the suites while achieving significant suite size reduction.

Keywords: *Software regression testing, testing criteria, test suite minimization, test suite reduction, fault detection effectiveness.*

1. Introduction

Software regression testing is a critical activity in the maintenance phase of evolving software. However, it requires large amounts of test cases to test any new or modified functionality within the program [1]. Re-running all existing test cases together with the new ones is often costly and even infeasible due to time and resource constraints. To address this problem, the research community proposed techniques to optimize regression testing [2], [3], [4], [5], [6], [7], [8]. Re-running test cases that do not exercise any changed or affected parts of the program makes extra cost and gives no benefit. An effective technique is to permanently discard such redundant or obsolete test cases and retain the most effective ones to reduce the excessive cost of regression testing [6]. Such technique attempts to find a minimal subset of test cases which satisfy all the testing requirements as the original set does [9]. This subset could be found during the test case generation or after creating the test suite. Apparently the less the number of test cases the less time it takes to test the program. This consequently improves the effectiveness of the test process. This technique is commonly known as *test suite reduction* or *test suite minimization* in the literature and the resulting suite is called *representative set* [3].

Almost all the previous test suite reduction techniques could significantly reduce the size of the test suites [10]. But an important issue deals with how well these reduced suites can be compared with their corresponding un-reduced suites using criteria rather than the suite size criterion. Since the purpose of test case execution is to detect faults

in the software, one measure of the suite quality is its fault detection capability. In fact, a potential drawback observed in test suite reduction studies is that permanent removal of test cases from a test suite may highly decrease the fault detection effectiveness of the remaining suite. Thus, the tradeoff between the time required to execute and manage test suites and their fault detection effectiveness should be considered when applying test suite reduction techniques [2].

In this paper, we propose a new algorithm for test suite minimization. The proposed algorithm greedily selects an optimum test case into the reduced suite until all testing requirements are satisfied. An optimum test case should satisfy two objectives simultaneously. First, it must satisfy the maximum number of unmarked requirements. Second, it must have the minimum overlap in requirements coverage with other test cases. The first objective attempts to select effective test cases in fault detection. The second one attempts to remove redundancy from the test suite and selects unique test cases in terms of requirements coverage. The proposed algorithm has two main features: First, it achieves significant suite size reduction and improves their fault detection effectiveness compared to other approaches. Second, the reduction process is based on the information of each program which can be obtained easily and accurately.

In order to evaluate the applicability of the proposed approach, we conducted experiments on the *Siemens* suite and the *Space* program. We also implemented the well-known *H* algorithm [3], to compare the results of our algorithm with those of *minimizing* test suites using the *H* algorithm.

The rest of the paper is organized as follows: Section 2 discusses the background of the test suite reduction techniques. Section 3 contains the outline of the proposed approach. Section 4 describes the empirical studies and the obtained results. Finally, conclusions are mentioned in section 5.

2. Background

In this section, first we describe the Test Suite Reduction problem. Then, an outline of the previous studies in this area will be presented.

2.1. Test suite reduction problem

The first formal definition of test suite reduction problem introduced in 1993 by Harrold et al. [3] as follows:

Given. $\{t_1, t_2,\ldots, t_m\}$ is test suite T from m test cases and $\{r_1, r_2,\ldots, r_n\}$ is set of test requirements that must be satisfied in order to provide desirable coverage of the program entities and each subsets $\{T_1, T_2,\ldots, T_n\}$ from T are related to one of r_is such that each test case t_j belonging to T_i satisfies r_i.

Problem. Find minimal test suite T' from T which satisfies all r_is covered by original suite T.

Generally the problem of finding the minimal subset T', $T' \subseteq T$ which satisfies all requirements of T, is NP-complete [10], because we can reduce the *minimum set-cover* problem to the problem of test suite minimization in polynomial time. Thus, researches use heuristic approaches to solve this problem. One heuristic method proposed by Harrold et al.

[3], tries to find the smallest representative set that provides the same coverage as the entire test suite does.

2.2. Related work

Related work in the context of test suite reduction can be classified into two main categories: The works in which a new technique is presented [3], [4], [5], [6], [7], [9], [11] and empirical studies on the previous techniques [1], [2], [10], [12]. The works which propose a new approach commonly include heuristic algorithms [11], genetic algorithm-based techniques [13] and approaches based on integer linear programming [14]. In a recent study [10], four typical test suite reduction techniques have been evaluated and compared on 11 subject programs. Based on the results, this study suggests that the heuristic H must be the first choice when selecting from the reduction techniques.

Testing criteria are defined in order to help the selection of subsets of the input domain to be covered during testing. For example, a code coverage criterion provides test suite adequacy with respect to coverage of the program entities and also provides a check on its quality. Assuming testing criterion *C* which satisfies by the test suite *T*, a test case, *t*, is *redundant* if the suite *T*-{*t*} also satisfies *C* [5]. Therefore, removing those test cases which are redundant with respect to some specific criteria preserves test suite's adequacy with respect to that criteria. In previous empirical studies [10] researchers commonly apply various code coverage criteria in their reduction techniques. The results of empirical studies [2], [12] show that the more percentage of the test suite size is reduced the more percentage of faults will be lost.

Usually a test suite reduction technique attempts to remove *redundancy* among test cases and retain the most *effective* ones into the test suite [6]. Effective test cases are those that are capable of satisfying the most requirements as well as exposing the most of the existing faults. Note that the more requirements are satisfied, the more execution paths within the program would be exercised which yields the all kinds of the faults to be exposed.

3. The proposed approach

Our approach to test suite reduction has been motivated by the following issue: An appropriate test suite reduction technique should select test cases that are both unique in exercising execution paths and effective in fault detection. The first objective attempts to remove as much redundancy from the test suite, and the second one seeks for satisfying the main purpose of software testing which is fault detection.

The proposed approach to test suite reduction uses test case-requirement matrix. This matrix shows the mappings between test cases and testing requirements. The elements consist of 1's and 0's which indicate for satisfying or dissatisfying the requirements by test cases respectively. The general idea of the proposed algorithm is as follows: At first the test case-requirement matrix is multiplied by its transposed matrix. The resultant is a square matrix of size $n * n$, such that n is the number of test cases. Each diagonal element of this matrix shows the number of unmarked requirements covered by the corresponding test case. Each non-diagonal element in the ith row and the jth column shows the number of requirements coverage in which the ith and the jth test cases overlap. Then, the algorithm *greedily* selects an optimum test case until a same

coverage of testing requirements is achieved. A test case is optimum which satisfies the maximum number of unmarked requirements (the maximum diagonal element) and simultaneously having the minimum overlap in requirements coverage with other test cases (the minimum value obtained by adding the all non-diagonal elements at the corresponding row). We call our algorithm *Bi-Objective Greedy* (*BOG*). The pseudocode description of the proposed algorithm is shown in the Figure 1. The inputs of this algorithm is a test suite with *m* test cases, a set of *n* testing requirements and also the *m*n* test case-requirement matrix. Besides, two arrays of Boolean values namely *marked* and *selected* are considered. The first array keeps the cumulative requirements coverage of the reduced test suite. The second one keeps the selection of test cases. The proposed algorithm consists of three main steps and a helper function which is described in the following.

Step 1: In this step, the test case-requirement matrix is multiplied by its transposed matrix and is kept in a matrix called *multiplied*. Using this matrix, a vector called *sumColumns* will be computed which will indicate the number of requirements coverage overlap of a test case with others.

Step 2: In this step, the algorithm repeatedly selects an optimum test case until all testing requirements are satisfied. In each of its iteration, first the test cases with the maximum diagonal values from *multiplied* matrix are selected into the *maxList*. Also, test cases with the minimum values in the *sumColumns* are selected into the *minList*. Then, a test case is selected from the intersection of these lists. If they are disjoint sets, a function, *selectOptimumTestCase*, is invoked to select a near optimal test case.

Step 3: In this step, the *selected* vector is updated with respect to the selected test case and cumulative coverage of the reduced suite is updated. Moreover, the diagonal elements of the *multiplied* matrix are updated for unselected test cases. If during the update process, the value of an element becomes zero, it is redundant and will be removed from further considerations.

The helper function: This function (Figure 2) is used when the intersection of the lists *maxList* and *minList* is empty. It selects a near optimal test case from the union of these sets. To achieve this, the function should first determine the optimum test case from each of the two sets. An optimum test case in the *maxList* should have the minimum respective element in the vector *sumColumns*. Oppositely, an optimum test case in the *minList* should have the maximum respective diagonal element in the *multiplied* matrix. Then, either of the two optimum test cases is selected as the near optimal test case based on which has the less distance. The distance of the optimum test case in *maxList* is the difference of its respective element in the *sumColumns* with the respective element of *sumColumns* for an arbitrary test case in *minList*. The distance of the optimum test case in *minList* is the difference of its respective diagonal element in the *multiplied* matrix with the respective diagonal element of *multiplied* for an arbitrary test case in *maxList*.

Worst-Case Runtime Analysis: Let *nt* denotes the number of test cases in the original test suite and *m* denotes the number of testing requirements. In the first step, the computation of the transposed matrix and the vector *sumColumns* needs the O(*nt.m.nt*) and O(*nt.nt*) time respectively. The second step needs O(*nt*) time to compute each of two lists and their intersection. Since at most *nt* test cases may be selected, the order of this step is O(*nt.nt*). The

```
define: requirement: set of coverage requirements for minimization: r1,r2,…,rn
input:
   t1,t2,…,tm: all test cases present in the test suite
   cv[m,n]: coverage matrix representing requirement coverage of each test case,
            1 for covered and 0 for uncovered
output:  RS: a reduced suite of test cases from the test pool.

declare:
    nextTest: one of test cases
    marked: array[1..n] of boolean, representing the covering if requirements, initially FALSE
    selected: array[1..m] of boolean, representing the selection of test cases, initially FALSE
    sumColumns: array[1..m] of integer
    multiplied: matrix[1..m][1..m] of integers
    maxList, minList, interSection: list of ti's

algorithm TestSuiteReduction
begin
 STEP 1:        multiplied := multiplication of the cv[m,n] * cv^T[n,m];  // initialization
                foreach ti do compute sumColumns[i], sum of the elements in the ith row
                          of the multiplied matrix, except for the diagonal element;
 STEP 2:        while there exists ri such that marked[i] == FALSE do
                   maxList := all ti for which the selected[i] == FALSE and
                              multiplied[i,i] is the maximum;
                   minList := all ti for which the selected[i] == FALSE and
                              sumColumns[i] is the minimum;
                   interSection := maxList ∩ minList;
                   if Card(interSection) == 0 then
                      nextTest := SelectOptimumTestCase(maxList, minList,
                                                        multiplied, sumColumns);
                   else if Card(interSection) == 1 then
                      nextTest := the test case in the interSection;
                   else
                      nextTest := any test case in the interSection;
                   endif
 STEP 3:           RS := RS ∪ {nextTest};
                   selected[nextTest] := TRUE;
                   multiplied[nextTest,nextTest] := 0;
                   foreach rj ∈ requirements where cv[nextTest,rj] == TRUE do
                             marked[i] := TRUE;
                   foreach ti for which the selected[i] == FALSE and
                           multiplied[nextTest,ti] > 0 do
                             multiplied[ti,ti] := the number of unmarked requirements
                                                  covered by the ti;
                             if multiplied[ti,ti] == 0 then
                                selected[ti] := TRUE;
                   endfor
                endwhile
                return RS;
end TestSuiteReduction
```

Figure 1. The pseudocode description of the proposed algorithm

third step needs O(m) time to mark requirements and O($nt.m$) time for updating the matrix. Under these assumptions, the total runtime of the proposed algorithm becomes O($nt.nt.m$).

```
function SelectOptimumTestCase(maxList, minList, multiplied, sumColumns)
declare:
    testCase: selected test case either from maxList or from minList
    minMin, minMax, maxMin, maxMax, minListDistance, maxListDistance: integer
begin
    minMax := the minimum value of sumColumns[t_i] for each t_i in the maxList;
    maxMax := the value of the mulplied[t_i] for one of the t_i in the maxList;
    maxMin := the maximum value of the mulplied[t_i] for each of the t_i in the minList;
    minMin := the value of the sumColumns[t_i] for one of the t_i in the minList;
    minListDistance := maxMax – maxMin;
    maxListDistance := minMax – minMin;
    if minListDistance < maxListDistance then
        testCase := the t_i from the minList for which multiplied[t_i] == maxMin;
    else
        testCase := the t_i from the maxList for which sumColumns[t_i] == minMax;
    endif
    return testCase;
end SelectOptimumTestCase
```

Figure 2. A helper function to select a near optimal test case from two sets of test cases

4. Empirical studies

In order to evaluate the proposed algorithm and compare the results with prior studies, we conducted an empirical study. In this section, we describe this study.

4.1. Subject programs, measures and analysis tools

Our studies have been conducted on the eight C programs as subjects (Table 1). *Siemens* suite includes seven programs developed by the researchers at Siemens Corporation for experiments with control-flow and data-flow test adequacy criteria [15]. These programs are associated with several faulty versions. Each faulty version of each program contains a single fault seeded in it. For each program there is a test pool which contains test cases developed for different black-box and white-box testing objectives. The eighth subject is the *Space* program which is a real one [16].

To investigate the effectiveness of our approach, we implemented the bi-objective greedy algorithm. Moreover, the *H* algorithm [3] has been implemented to compare the results of this approach with those of the proposed approach, since it is reported [10] as the best choice among current reduction techniques. We measured the following from the experiments:

1. The *percentage suite size reduction* $= \frac{|T| - |T_{red}|}{|T|} \times 100$, where $|T|$ is the number of test cases in the original test suite and $|T_{red}|$ is the number of test cases in the reduced test suite.

2. The *percentage fault detection loss* $= \frac{|F| - |F_{red}|}{|F|} \times 100$, where $|F|$ is the number of distinct faults exposed by original test suite and $|F_{red}|$ is the number of distinct faults detected by the reduced suite.

Besides, SAS 9.1.3 [17] was used to create box plots. Box plot diagrams are commonly used to visualize the empirical results in test suite reduction studies.

Table 1. Subject programs used in the experiments

Program name	LOC	Faulty versions	Test pool size	Program name	LOC	Faulty versions	Test pool size
tcas	148	41	1608	print-tokens	402	7	4130
totinfo	346	23	1052	print-tokens2	483	10	4115
schedule	299	9	2650	replace	516	32	5542
schedule2	297	10	2710	space	6218	38	13585

4.2. Experiment setup and results

Our experiments follow a setup similar to that used by Rothermel et al [2] and Jeffrey [16]. For each program, we created branch coverage adequate test suites for six different suite ranges named as B, B1, B2, B3, B4 and B5. For each suite range, we first selected *X* * *LOC* test cases randomly from the test pool and added to the test suite, where *X* is 0, 0.1, 0.2, 0.3, 0.4 and 0.5 respectively and *LOC* is the number of lines of code for each program. Then, randomly-selected test cases are added into the test suite as necessary so long as each test case increased the cumulative branch coverage of the suite, until the test suite becomes adequate with respect to branch coverage. In this way, the developed test suites have various types and varying levels of redundancy exist between them. For each program, we created 1000 such branch coverage adequate test suites in each suite size range. In order to gather branch coverage information of test cases, all programs were hand-instrumented.

Both the *H* algorithm and the *BOG* algorithm were applied to the generated suites with respect to branch coverage as testing criterion. The results of this experiment are shown in the columns labeled H and BOG in Table 2. The values in each row of the table are average values for 1000 suites in each range.

In this table, $|T|$ indicates for the original suite size, $|F|$ for the number of faults exposed by the original suite, $|T_{red}|$ for the reduced suite size, $|F_{red}|$ for the number of faults exposed by the reduced suite size, %Size Reduction for the percentage suite size reduction and %Fault Loss for the percentage fault detection loss. The box plot in the Figure 3 shows the distribution of the percentage of size reduction (SR) and percentage fault detection loss (FL) in the largest suite size range (B5) for each program. In this figure, boxes are paired such that, white pair of boxes shows percentage of size reduction and gray pair of boxes shows the percentage of fault detection loss. In each pair, left side box indicates for our algorithm and the right side one indicates for the H algorithm.

Suite Size Reduction: Table 2 and the Figure 3 show that the average suite size reduction is high for all programs. The results also show that both the H algorithm and the BOG algorithm could reduce the suites to the same extent. This indicates the effectiveness of the proposed algorithm in determining redundant test cases. Moreover, suite size reduction increases for larger suites. The reason is that the high number of test cases provides more opportunities for the algorithm to select among test cases.

Fault Detection Loss: The results show that the average fault detection loss has been improved except for *schedule2*. In addition, the amount of fault loss for the *tcas* and *schedule2* is relatively high among other programs. For the *tcas* program, this may be due to simplicity of this program. Many of the test cases satisfy the same branches and will be

removed since they are redundant. But these test cases exercise unique execution paths with respect to some other testing criteria. Hence, using different or fine-grained criteria would result to significant improvements in fault loss. For the *schedule2*, the fault loss is high simply due to low number of faults. The fault loss for the *Space* program is less than others. This indicates the effectiveness of the proposed approach for real programs.

Table 2. The results of the experiments comparing the H algorithm and the proposed algorithm

program	Suite range	$\|T\|$	$\|F\|$	$\|T_{Red}\|$		$\|F_{Red}\|$		%Size Reduction		%Fault Loss	
				H	BOG	H	BOG	H	BOG	H	BOG
ptok	B	17.86	3.81	8.28	8.38	3.44	3.46	52.58	52.01	8.34	7.92
	B1	29.46	3.99	8.16	8.27	3.46	3.47	69.33	68.90	11.73	11.33
	B2	48.29	4.29	8.15	8.30	3.44	3.48	78.39	78.03	17.64	16.74
	B3	65.27	4.45	8.08	8.22	3.43	3.49	82.54	82.28	20.29	19.09
	B4	86.42	4.69	8.05	8.19	3.42	3.45	85.99	85.81	24.49	24.03
	B5	105.46	4.96	8.01	8.18	3.43	3.48	87.76	87.56	28.35	27.38
ptok2	B	13.41	8.47	8.01	8.29	8.17	8.22	39.08	37.30	3.46	2.96
	B1	28.79	8.87	7.63	7.96	8.14	8.19	67.11	66.24	7.96	7.12
	B2	51.68	9.24	7.13	7.44	8.20	8.27	79.47	78.75	10.88	10.16
	B3	75.1	9.39	6.90	7.28	8.22	8.25	84.06	83.34	12.23	11.90
	B4	99.73	9.53	6.64	7.01	8.29	8.29	87.74	87.19	12.78	12.73
	B5	128.01	9.68	6.41	6.85	8.32	8.26	90.76	89.73	13.89	14.10
replace	B	23.13	13.67	13.63	14.26	10.99	11.35	40.52	37.81	19.04	16.52
	B1	38.36	15.49	13.52	14.36	11.01	11.42	61.15	58.95	27.92	24.93
	B2	59.55	17.11	13.40	14.23	10.90	11.32	71.33	65.75	34.78	32.35
	B3	84.94	18.64	13.18	14.13	10.97	11.49	78.28	76.94	39.69	36.85
	B4	108.98	19.45	12.99	14.05	10.75	11.19	81.67	80.42	42.96	40.59
	B5	135.34	20.48	12.79	13.91	10.87	11.20	84.75	83.66	45.08	43.41
sched	B	8.8	3.60	6.30	6.37	3.06	3.09	27.27	26.56	13.41	12.63
	B1	18.82	4.57	6.19	6.27	2.89	3.04	6046	60.93	34.36	33.40
	B2	33.19	5.33	6.01	6.11	2.96	3.02	74.66	74.32	43.03	41.84
	B3	46.76	5.74	5.96	6.04	3.02	3.07	79.81	79.54	46.02	45.07
	B4	61.67	5.97	5.86	6.01	2.92	3.12	83.30	82.43	50.04	45.88
	B5	76.22	6.10	5.84	5.95	2.85	3.00	85.56	85.31	51.96	49.45
sched2	B	7.96	2.35	5.33	5.37	2.02	2.05	31.42	30.82	12.31	10.49
	B1	17.75	2.79	5.19	5.25	2.11	2.15	63.81	63.44	22.52	20.87
	B2	32.08	3.44	5.09	5.13	2.10	2.14	77.03	74.92	34.49	32.19
	B3	45.82	3.97	4.97	5.04	2.16	2.11	81.84	81.67	39.96	40.54
	B4	60.60	4.50	4.83	4.9	2.17	2.11	85.23	85.05	46.14	46.82
	B5	73.65	4.81	4.78	4.84	2.12	2.12	87.35	87.19	50.21	49.35
space	B	154.52	32.03	119.06	121.25	31.24	31.34	22.90	21.49	2.46	2.14
	B1	362.25	32.59	117.07	120.38	31.07	31.26	60.84	59.85	4.72	4.05
	B2	639.92	33.09	114.80	118.88	30.85	31.10	73.82	73.03	6.67	5.94
	B3	941.91	33.51	112.94	117.33	30.57	30.90	80.96	80.36	8.66	7.68
	B4	1272.76	33.81	111.50	116.06	30.39	30.74	84.75	84.24	9.99	8.98
	B5	1558.41	34.01	110.69	115.31	30.22	30.63	86.88	86.45	11.02	9.83
tcas	B	5.70	7.13	5.00	5.00	6.44	6.31	11.12	11.12	8.32	10.17
	B1	9.22	8.70	5.00	5.00	6.39	6.27	39.53	39.53	23.04	24.81
	B2	15.16	11.21	5.00	5.00	6.37	6.29	57.70	57.70	37.41	38.48
	B3	21.27	13.61	5.00	5.00	6.58	6.63	66.06	66.06	44.95	45.20
	B4	28.29	15.54	5.00	5.00	6.57	6.80	71.99	71.99	51.19	50.66
	B5	35.70	17.28	5.00	5.00	6.42	6.59	76.35	76.27	56.33	55.50
totinfo	B	8.92	13.66	5.56	5.65	12.75	12.77	36.16	35.20	6.37	6.26
	B1	19.99	15.41	5.38	5.50	12.83	12.93	66.37	65.73	15.93	15.34
	B2	36.10	16.87	5.27	5.40	12.96	13.21	78.17	77.67	22.17	20.80
	B3	52.38	17.89	5.19	5.28	13.11	13.51	82.66	82.34	25.55	23.42
	B4	69.18	18.50	5.12	5.19	13.14	13.79	86.08	85.87	27.72	24.46
	B5	90.76	19.37	5.06	5.14	12.96	13.99	89.50	89.35	32.06	26.94

To determine whether the degradation in fault detection loss observed for the proposed algorithm over the H algorithm is statistically significant, we conducted a *hypothesis test for the difference of the two means* [18]. The samples are the number of distinct faults exposed by each of the 1000 reduced test suites for suite size range B5 using the H algorithm and the proposed algorithm. We considered the *null hypothesis* that there is no difference in the mean number of the exposed faults by the two algorithms. We used a reference table of critical values presented in [18]. Table 3 shows the resulting *z values* computed for the hypothesis test along with the percentage confidence with which we may reject the null hypothesis. Note that for z values greater than 4.0, we can reject the null hypothesis with the confidence over than 99.99 percent. Thus, the difference in the mean number of faults exposed by the H algorithm and the BOG algorithm is statistically significant.

Table 3. Computed z value and the corresponding percentage of confidence for rejecting the null hypothesis for each program

Program Name	Computed z value	Percentage of confidence for rejecting the null hypothesis
printtokens	1.56	>88%
printtokens2	-1.29	>80%
replace	2.89	>99.5%
schedule	2.56	>98.9%
schedule2	0.15	<50%
space	5.71	>99.99%
tcas	1.50	>86.6%
totinfo	8.84	>99.99%

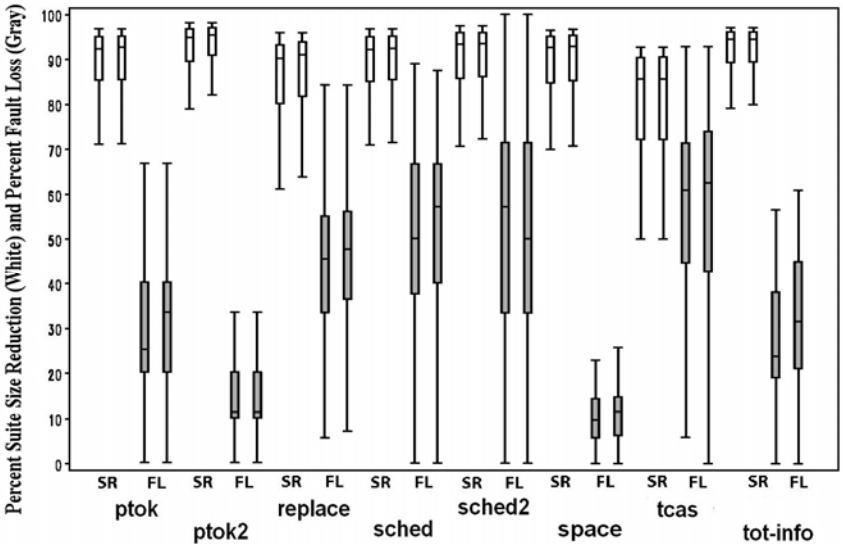

Figure 3. The boxplot for the percentage suite size reduction and percentage fault detection loss

To further aid in experimental analysis, Tables 4 through 11 are presented for each program. To save space, data is presented for suite size B5 only. Each table presents a matrix in which each of 1000 suites for the given subject program are plotted, comparing the BOG-minimized suites to their corresponding H-minimized counterparts. The first column of each row in a matrix represents the sign of the difference in number of test cases in the BOG-minimized suite over the corresponding H-minimized suite:

$$|T|_{diff} = |Tred|_{BOG} - |Tred|_{H} \quad (1)$$

The first row of each column represents the sign of the difference in number of faults detected by the BOG-minimized suite over the H-minimized counterpart:

$$|F|_{diff} = |Fred|_{BOG} - |Fred|_{H} \quad (2)$$

Each entry in a matrix is a test suite count. Entry x at a row/column in a table indicates that among 1000 suites for suite size range B5 of the given subject program, there were x suites such that the BOG-minimized suites contains less, equal or more test cases and detected less, equal or more faults than the corresponding H-minimized suites. Column and row test suite

sums are provided in the rows and columns labeled with a "∑". The following observations are made from Tables 4 through 11:

1. In all subject programs except for printtokens2 and schedule2, there are far more suites with increased fault detection than decreased fault detection, when going from the H techniques to the BOG technique. This shows that the BOG technique, while not always improving the fault detection of suites, has a much greater likelihood of increasing fault detection effectiveness than of decreasing it.
2. Programs schedule, schedule2, printtokens, and printtokens2 have a relatively larger number of suites in which the fault detection effectiveness remained unchanged in going from H to BOG. This is most likely due to the fact that these four programs have the fewest number of faulty versions available, so there are fewer opportunities for detecting new distinct faults with these four programs.
3. In programs printtokens, schedule, schedule2, tcas, and totinfo have a high portion (nearly all) of suites which remained unchanged in their size reduction when going from the H to BOG. For printtokens2, about half of the suites and for replace about one third of suites remained unchanged or more reduced by BOG than H. For space, almost all suites have a few more (Table 2 and Figure 3) number of test cases when using BOG technique than H. All these suggest that the BOG technique has a very high likelihood of minimizing suites than of reducing them.

Table 4. BOG vs H sign of the difference in number of test cases / detected faults matrix: printtokens. This matrix shows the number of test suites for suite size range B5. "∑" represents a sum of test suite counts

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	1	1	1	3
= 0	25	767	42	834
> 0	9	119	35	163
∑	35	887	78	1000

Table 5. BOG vs H matrix: printtokens2. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	86	61	104	251
= 0	91	75	99	265
> 0	198	132	154	484
∑	375	268	357	1000

Table 6. BOG vs H matrix: replace. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	2	1	0	3
= 0	66	102	75	243
> 0	220	174	360	754
∑	288	277	435	1000

Table 7. BOG vs H matrix: schedule. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	4	4	1	9
= 0	131	569	175	875
> 0	12	40	64	116
∑	147	613	240	1000

Table 8. BOG vs H matrix: schedule2. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	2	10	11	23
= 0	163	596	130	889
> 0	18	34	36	88
∑	183	640	177	1000

Table 9. BOG vs H matrix: space. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	1	2	0	3
= 0	5	8	4	17
> 0	236	331	413	980
∑	242	341	417	1000

Table 10. BOG vs H matrix: tcas. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	0	0	0	0
= 0	411	113	476	1000
> 0	0	0	0	0
∑	411	113	476	1000

Table 11. BOG vs H matrix: totinfo. This table is organized the same way as Table 4

$\lvert T\rvert_{diff}$ \ $\lvert F\rvert_{diff}$	< 0	= 0	> 0	∑
< 0	5	3	1	9
= 0	177	271	494	942
> 0	12	7	30	49
∑	194	281	525	1000

4.3. Threats to validity

In this section, we describe the potential threats to validity of our study.

Threats to construct validity. In our study, the measurement for the percent of fault loss assumes simple model for cost which treats all faults as equally severe. But in practice, faults have wide range of severity from less critical to more critical.

Threats to internal validity. The most important issue deals with hand-instrumentation of code which we have done for obtaining branch coverage of test cases. To validate the correctness of this process, the instrumentation was re-verified. Another issue is composition of the test suites. However, we utilized the process of creating suites which was employed in previous studies [2], [16].

Threats to external validity. Siemens programs are widely used in software testing studies. However, these programs are limited and their faults are known. Moreover, they are not real programs and the faults are hand-seeded. Space program is a big and real one, used to our case study. But it is one of such a program we used.

4.4. Discussion

One of the main objectives of the proposed algorithm is to select effective test cases in fault detection. However, testers are not aware of faults and their severities before testing. Thus, they have to estimate faults behavior through some heuristic methods. One way to do this is using the extent of code coverage by each test case. In other words, coverage of testing requirements can provide a reasonable estimate of fault detection capability [8].

The proposed algorithm requires the coverage information of test cases for a single criterion. Unlike some existing approaches, the required information can be gathered accurately and full automatically. Moreover, our algorithm exactly determines test cases once they become redundant by selecting a test case into the reduced suite. Thus, it is possible to use other criteria [16] to improve fault detection loss.

9. Conclusions

We have presented a new algorithm for test suite minimization. The new algorithm considers test suite minimization as an optimization problem with two objectives. The first objective is fault detection capability which should be maximized. The second objective is number of test cases which should be minimized. The new algorithm is indeed applicable and effective in reducing suites with significant suite size reduction and improved fault detection capability. This observation is evidenced by the results obtained from the experiments similar to prior studies which compare our algorithm with the best existing approach on typical benchmarks. In future, we are going to conduct experiments on available real C programs as well as java benchmarks.

References

[1] G. Rothermel, M.J. Harrold, J. von Ronne, C. Hong, “Empirical Studies of Test-Suite Reduction”, Journal of Software Testing, Verification, and Reliability, 12(4), 2002, pp. 219-249.
[2] G.Rothermel, M.J. Harrold, J. Ostrin, C. Hong, “An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites”, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, 1998.
[3] M.J. Harrold, R. Gupta, M.L. Soffa, “A Methodology for Controlling the Size of a Test Suite”, ACM Transactions on Software Engineering Methodologies 2**,** 1993, pp. 270-285.
[4] T.Y. Chen, M. F. Lau, “Heuristics toward the Optimization of the Size of a Test Suite” Proc. 3rd Int’l Conf. on Softw. Quality Management. Vol. 2, Seville, Spain, April 1995, pp. 415-424.
[5] J.A. Jones, M.J. Harrold, “Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage”, IEEE Trans. Softw. Eng. 29, 2003, pp. 195-209.
[6] S. McMaster, A. Memon, “Call-Stack Coverage for GUI Test Suite Reduction”, IEEE Trans. Softw. Eng. 34**,** 2008, pp. 99-115.
[7] S.Tallam, N. Gupta, “A concept analysis inspired greedy algorithm for test suite minimization”, Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, Lisbon, Portugal, 2005.
[8] D. Leon, A. Podgurski, “A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases”, Proceedings of the 14th International Symposium on Software Reliability Engineering, IEEE Computer Society, 2003.

[9] Z. Chen, B. Xu, X. Zhang, C. Nie, "A novel approach for test suite reduction based on requirement relation contraction", Proceedings of the 2008 ACM symposium on Applied computing. ACM, Fortaleza, Ceara, Brazil, 2008.
[10] H. Zhong, L. Zhang, H. Mei, "An experimental study of four typical test suite reduction techniques", Inf. Softw. Technol. 50, 2003, pp. 534-546.
[11] T. Y. Chen, M. Lau, "A new heuristic for test suite reduction", Information and Software Technology 40 (5-6), 1998.
[12] W.E. Wong, J.R. Horgan, S. London, A.P. Mathur, "Effect of test set minimization on fault detection effectiveness", Softw. Pract. Exper. 28**,** 1998, pp. 347-369.
[13] N. Mansour, K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing", Journal of Software Maintenance **11,** 1999, pp. 19-34.
[14] J. Black, E. Melachrinoudis, D. Kaeli, "Bi-Criteria Models for All-Uses Test Suite Reduction", Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, 2004.
[15] G. Rothermel, S. A. Elbaum, Kinneer, H. Do, "Software-artifact infrastructure repository", http://www.cse.unl.edu/~galileo/sir.
[16] D. Jeffrey, N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction", IEEE Trans. Softw. Eng. 33, 2007, pp. 108-123.
[17] SAS 9.1.3 Documentation, SAS/GRAPH 9.1 Reference, http://support.sas.com/documentation/onlinedoc/91pdf/index_913.html.
[18] J. E. Freund, Mathematical Statistics, 5th ed., Prentice-Hall, 1992.

Authors

Saeed Parsa received his B.Sc. in mathematics and computer science from Sharif University of Technology, Iran, his M.Sc. degree in computer science from the University of Salford in England, and his Ph.D. in computer science from the University of Salford in England. He is an associate professor of computer science at Iran University of Science and Technology. His research interests include software engineering, soft computing and algorithms.

Alireza Khalilian received his B.Sc. in computer engineering from Azad University, North Branch, Iran, and his M.Sc. degree in software engineering from Iran University of Science and Technology, Iran. His research interests include software engineering, software testing, software debugging, and object-oriented modeling and design. He has published several papers in prestigious international conferences and journals including TAP, CEE-SET, ASEA, ICCSIT, CSICC, and ICEE. He has been the recipient of several honors and awards, including the Outstanding Teacher Award from the Computer Engineering Department of the University of Applied Science and Technology, and he has won the Outstanding Student Award in December 2000 as an undergraduate student.