Mutual Information Gain based Test Suite Reduction

Meenu Dave, PhD
Jagan Nath University
Jaipur

Rashmi Agrawal
Jagan Nath University
Jaipur

ABSTRACT

The test suite optimization during test case generation can save time and cost. The paper presents an information theory based metric to filter the redundant test cases and reduce the test suite size while, maintaining the coverage of the requirements and with minimum loss to mutant coverage. The paper propose two versions, RR and RR2. RR filters test cases for each requirement, where as, RR2 filters till the target coverage is achieved. The paper suggests the time and phase for the implementation of the algorithms, based on results. The results show that the proposed algorithms are effective at optimizing the testing process by saving time and resource.

General Terms

Software Testing, Test suite generation

Keywords

Information Theory, Optimization, Mutual Information Gain, Test suite size reduction, test data generation

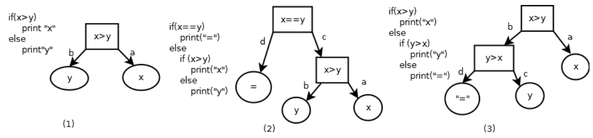
1. INTRODUCTION

Software systems go through a number of changes in during their evolution phase (deletion, addition, debugging,modifications or change in requirements) [1], [2]. The initial versions of the software are tested with test suite (TS) and stored along with the respective version for the future. As the software evolves, new versions are released and test cases are generated to execute these modifications [1]. Testing is an expensive process consuming a lot of time, space and resource. As the new test cases are added, the test suite size increases, putting addition burden on costs (time, human efforts, resource allocation, data storage). The cost of testing is directly proportional to the size of a TS [3]. To save time and cost, one option is to optimize the test suite by identifying and eliminating redundant or irrelevant (like losing coverage, fault detecting capability) test cases. Optimization of TS is a continuous process. *The TS optimization is the process of removal of test cases to reduce test suite size, without losing the coverage of requirements or reduction in fault detecting capability.*
The requirements of coverage are a set of rules, known as *adequacy criteria* or *testing criteria*, to be covered by test cases [3] [4]. Test suites are quantified based on their code coverage. The requirements can be based on the coverage of structural elements of the program (e.g. statement coverage, branch coverage, decision coverage) or the flow of information or fault detection capability(e.g. killing mutants in mutation testing)or a combined technique two or more criteria. The proposed heuristic is independent of the adequacy criteria and syntactically independent.
The objective of the proposed algorithms and techniques is to identify and remove redundant test cases to save time during the maintenance of software [5]. The number of test cases may be huge when generated automatically as compared to manual [3]. In that case, the TS optimization becomes a critical part of maintenance. This paper frames the test suite size and execution minimization as an optimization problem.
Generally the optimization of test suite is associated with regression testing. Test suite generation along with proposed algorithms, can bring efficiency to testing and bringing down the maintenance cost.In our paper we propose to move the optimization to the initial stage of testing, when the test cases are generated, saving time and space from the very beginning. The test suite is divided into two sub suites,the reduced and the redundant. Whenever the new test cases are generated, they are checked for redundancy in both the sub set, and if they exist in any of the sets, they are ignores, else they are added. Testing has many stages, however, traditionally a testing phase consists of a) test case generation b) execution c) selection d) maintenance. With the implementation of the proposed algorithm, first phase (test data generation) will have following stages a) test data search b) test data execution c) analysis of output d) analysis of execution profile e) optimization.
A number of test suite reduction algorithms and techniques have been proposed [2],[3], [6], [7]. This paper presents a TS size reduction technique that efficiently generates a reduced TS with full coverage. The heuristic designed is based on Information theory metrics [8], [9] to identify and eliminate the redundant test data.

Summary of steps. a) Generate a random pool of test data b) record the execution profile c) calculate the redundancy for each test case by implementing the proposed algorithms d) add the selected to the reduced TS e) analyze the coverage.
The paper is organized as follows. Section II covers the basics of unit testing, evolution of test cases with version and generation of diagnostic matrix from execution profile, a brief introduction to the basic concepts of information theory, measurements, and finally assumptions. Section III shows the proposed algorithms along with description. Section IV covers the related work. Section V is the details of the experimental set up. Section VI shows the results of the experiments and their discussion. Section VII concludes the paper.

2. BACKGROUND

Testing is a process in which a program, is executed with some real environment like data, to know the run-time behavior of program by analyzing the output. If the output differs from the expectations, it is marked as fault in system. The aim of testing is to ensure that the program is behaving as expected and reveal unidentified errors. Testing is performed at all the stages of development (requirements, specification, code, integration, packing, acceptance). The testing performed at the basic level, the smallest unit of code is known as *Unit testing*. It is important for the building blocks to be error free to deliver a quality product to the customers. The code under testing is first analyzed for its structure and test cases are derived. The entities of the code can be expression statements, decision statements, loops, variable or data structures. Testing a program with all the possible inputs is not feasible. To select a representative set of important test cases, we quantify them against some criteria for selection. The test cases covering the most of these are criteria are included into the test suite [4]. A test suite is adequate to a criteria if the coverage is 100%. Common adequacy criterion are branch coverage, statement coverage, and line coverage. Generating a test case that adds to the coverage of the test suite is an important phase of testing [3]. To get the run details of the entities or the path covered by the test case, the program is first instrumented. The code is analyzed as per the adequacy criteria and probes or *traces* are inserted at appropriate places to collect the desired information. The details of run time data is a set of entities visited by test case, and known as *execution profile*, or coverage information. The run time details of all the test cases are compiled as a *diagnostic matrix*. In this paper, we have two sets of diagnostic matrix, first, a collection of vectors, where for each trace the respective test cases are stored as a vector. Second, for each test case, where each trace is marked as visited or not visited. Test suite generation, testing and retesting is a repetitive ask in software development cycle. The initial test suite is small, however, as existing are modified or new components are added to the software, it grows in size and complexity. To test the modifications and capture new entities, new test cases are generated. Removal of duplicate and less relevant test cases can reduce the test suite size. In this paper a test case is redundant if it reaches a *redundant threshold*. The redundant threshold is calculated by mutual information of information theory.



```
if(x>y)
    print "x"
else
    print"y"
```

```
if(x==y)
    print("=")
else
    if (x>y)
        print("x")
    else
        print("y")
```

```
if(x>y)
    print("x")
else
    if (y>x)
        print("y")
    else
        print("=")
```

Fig. 1. Evolution of Software. Two different Control flow Graphs presenting same logic

A software can have a number of versions and the execution profile of each test case of one version can be different from another. This brings uncertainty in test case execution profile. To capture the ambiguity, this paper presents an information theory derived, probabilistic measure, mutual information [8], [9]. Test cases are measured for mutual information gain. If there is none, then the test cases are declared as redundant and any one of them is selected for the reduced TS, marking the other as redundant. Figure 1 illustrates the various versions of a software. Figure 1 (1) is the initial program. The small code is shown as Control flow graph. The program in Figure 1 (1) shows one branch statement with two

Table 1. Test cases and code coverage for three versions of code.

TS	V1			V2(a)						V2(b)					
	1	a	b	1	a	b	2	c	d	1	a	b	2	c	d
t1	1	1		1	1		1	1		1	1				
t2	1		1	1		1	1			1	1		1	1	
t3	1		1				1		1	1		1	1		1
t4	1		1				1		1	1		1	1		1
t5	1	1		1	1		1	1		1	1				
t6	1	1		1	1		1	1		1	1				

Table 2. Traces and respective test cases

Version	Traces	Test Cases
V1	1	t1, t2, t3, t4, t5, t6
	a	t1, t5, t6
	b	t2, t3, t4
V2(a)	1	t1, t2, t5, t6
	a	t1, t5, t6
	b	t2
	2	t1, t2, t3, t4, t5, t6
	c	t1, t5, t6
	d	t4, t5
V2(b)	1	t1, t2, t5, t6
	a	t1, t2, t5, t6
	b	t3, t4
	2	t2, t3, t4
	c	t2
	d	t4, t5

branches. Figure 1(2) and Figure 1 (3)shows the two possible versions of the new logic added to the existing logic of Figure1(1). The test case can also be quantified with its ability to find faults. The fault detection capabilities of TS, can highly affected by test suite reduction [23]. Table 1 illustrates the code coverage of TS . The table is a sample to showing differences in execution profile the change in versions. The values of test cases are t1(2,1), t2(1,2), t3(1,1), t4(0,0), t5(45,5), t6(68,-3). The traces are shown as 1,a,b,2,c,d. The first version V1 had only (1,a,b), while, later version has (1,2,a,b,c,d). The existing test cases are re-run on the new versions. Test case t1 has coverage of (1,a) in version V1, while a coverage of (1,a,2,c) for V2(a) and (1,a) for version V2(b). There are six test cases in the test suite. From the table 1 it is clear that t4, t5, and t6 are redundant. However, for large and complex source doe with test cases in thousands, the task of identifying redundancy is very laborious. A heuristic is designed to check the redundant test cases. To address this question, we have designed an algorithm RedundancyReduction(RR). The algorithm, RR is based on the concepts of information theory, which is covered in details in section 2.1. Table 2 shows the requirement's matrix.

2.1 Entropy, Joint Entropy, Conditional Entropy and Mutual Information Gain

This section define Entropy, Joint Entropy , conditional Entropy, Information Gain and Mutual Information Gain, briefly.

2.1.0.1 Entropy H. The measurement of uncertainty in a variable is *Entropy*. Given a random variable (rmv) X, with a set of possible discrete values $x_1, x_2, ... x_n$, and $p_i > 1$ then, Shannon's Entropy (H) [9] can be defined as by following equation.

$$H = -\sum p_i * log(p_i) \tag{1}$$

In this paper the probability is the frequency of the occurrence of a value in a given data set, divided by the total number of transactions or occurrences. Example 1: Given X, with probability set as (1/2, 1/4, 1/4), then the entropy is $H(X)$ as shown in following equation:
$H(X) = -1/2\ log\ 1/2\ - 1/4\ log\ 1/4\ - 1/4\ log\ 1/4 = 1.5$
Given $X = 1, 0$, with probability of $p(1) = p$ and $p(0) = 1 - p$, then, $H(X)$ is given by equation 2.

$$H(X) = -p\ log\ p - (1 - p)\ log\ (1 - p) \tag{2}$$

Joint Entropy $H(X, Y)$ Entropy measures uncertainty for one variable. When there are two variables, suppose X and Y, then the $H(X, Y)$ is given by equation (3) [9].

$$H(X, Y) = -\sum_{x \in X}\sum_{y \in Y} p(x, y)\ log(x, y) \tag{3}$$

In equation (3), $p(x)$ is the probability of x in the rmv X and $p(y)$ is the probability of y in the rmv Y. **Conditional Entropy** $H(X|Y)$ When there are two rmv, X and Y, and the entropy of one variable is conditional on the value of the other variable, then the $H(X|Y)$ as in equation 4 and 5 [9] with given probabilities $p(x)$, and $p(y)$.

$$H(Y|X) = -\sum_{x \in X} p(x) H(Y|X = x) \tag{4}$$

$$H(X|Y) = -\sum_{x \in X} p(y) H(X|Y = x) \tag{5}$$

Mutual Information I It is the measure of information that one rmv contains about rmv. It is the reduction of uncertainty of one rmv due to the knowledge of the other [9]. Suppose there is a rmv X and Y. The mutual information $X;Y$ is the reduction in X due to Y [9].

$$I(X; Y) = H(Y) - H(Y|X) \tag{6}$$

$$I(X; Y) = H(X) - H(X|Y) \tag{7}$$

$$I(X; Y) = H(X) + H(Y) - H(X, Y) \tag{8}$$

Given marginal probabilities function $p(x)$, $p(y)$ and joint probability mass function $p(x, y)$ MI can be rewritten as

$$I(X; Y) = \sum_{x \in X}\sum_{y \in Y} p(x, y) * log\frac{p(x, y)}{p(x) * p(y)} \tag{9}$$

Example 2. Suppose there are two vectors, VC1 and VC2. The values of these vectors are: VC1 = [00110011] and VC2 = [10110110] The mutual information gain of these is =0.905. Suppose the values of VC3 and VC4 are given as: VC3 = [00110010] and VC3 = [00110010] The mutual information gain is =0.0.

2.2 Measurement

The test suite is measured for the coverage of traces and the ability of detecting faults FDE (Fault Detection Effectiveness). The original source code is altered, creating multiple versions, with each version called *mutant* having an artificial fault induced through *mutant operators* [10, 11, 12, 13, 14, 15, 16]. The each test case in the test suite is executed with P. The result is analyzed with the oracle , if there is difference, the mutant is said to be *killed*, else it is *alive* [17]. Live mutants that cannot be killed are known as *equivalent mutants*. The number of mutants killed by a test case is the *mutant score* for the test case. Test cases that have good mutant score are said to be good at detecting faults and kept in test suite. A test suite is said to be test adequate if it able to kill all the non-equivalent mutants.
Let Mn be a set of mutants $Mn = \{m_1, m_2, ...m_n\}$, EMn be Equivalent mutants, KMn be the number of mutants killed, then the Mutant Score (MS) for a test suit is given by the following equation.

$$MS = \frac{KMn}{Mn - EMn} * 100, whereFDE = MS \tag{10}$$

To check the FDE for the reduced test suite is given by equation 11.

$$RMS = \frac{RKMn}{Mn - EMn} * 100 \tag{11}$$

FDE might not be effected due to the reduction, as the test cases not contributing to the code coverage, may not be eliminating any unique faults[24]. Loss of FDE ($LFDE$) is given by equation 12.

$$LFDE = \frac{MS - RMS}{MS} * 100 \tag{12}$$

The loss of coverage ($LCOV$) is defined by the difference between the coverage of original test suite ($FCOV$) and the reduced test suite ($RCOV$).

$$LCOV = \frac{FCOV - RCOV}{FCOV} * 100 \tag{13}$$

The reduction in size is the main measure for the RR and RR2 algorithm. The aim of the algorithms is to reduce the test suite and bring it minimal. The reduction in size$Rsize$ is given by equation 14. The test suite size of the test suite TS is $TSsize$ while the reduced suite is $RTSsize$.

$$Rsize = \frac{TSsize - RTSsize}{TSsize} * 100 \tag{14}$$

Table 3. Details of variable of Algorithm

Variable	Details
P	Source code of Program under testing
S	Population of Test case inputs for program P
TS	Test suite to be optimized
$Testcase$	Test case of TS
$TClog$	Test case log
$EClog$	Execution profile as an array of 0's and 1's
$RMlog$	Log of requirements
$Selected$	Reduced Test suite
$Redundant$	Set of redundant test cases
RL	List of requirements
$Tnext$	Test case of TS
MI	mutual information gain of two test cases
$H()$	Entropy of an array
$H(\|)$	Condition Entropy of two array
$RedundancyLevel$	Cutoff point to declare a test case redundant
$TCovReq$	Requirements covered by test cases
$TTrace$	Traces covered by test T
FST	Full set of test cases

3. REDUNDANCY REDUCTION

This section covers the Redundancy Reduction algorithm RR and RR2. The algorithm are explained followed by analysis of reduced test suits.

3.1 Algorithm RedundancyReduction RR and RR2

Given : A program P with latest version of test suite TS to be optimized. A set of requirements RS that have been covered by TS.
Problem : Design a minimal test suite Selected, form the existing suite TS, such that $Size(Selected) < Size(TS)$ and $Coverage(Selected) = Coverage(TS)$.
Suppose a test suite TS of size N is given with good coverage of the given criteria. The reduced or minimal set is a subset of size NM, is derived from the TS such $NM < N$, will the same coverage for the same criteria[3]. A requirement which is executed by a single test case only, should be present in the TS [2], hence generating a minimal test suite. The important point in executing a test suite size reduction is the analysis of the criteria coverage of the reduced test suite. However, good test suite reduction should reduce the size, while keeping the coverage intact. This condition is the motivation to include the greedy algorithm of Harnold et al. [2], to the $RR2$. The algorithm RR is verified for the above minimal constraint by comparing its results with $RR2$. The steps of RR are common to both the algorithms. However, $RR2$ has one additional step. Primary inputs to the algorithm are the log details of the test suite and the traces derived from the structure of P. The test log contains the details like test number, and execution profile (the annotations of the traces covered). The requirements' log stores the details of the traces and the test cases that executed them (annotations of the test ids) , showing the cardinality of each trace . The table 3 shows the details of the variable of algorithms proposed.

The first step is the initialization of inputs. Generate a test suite TS. TS has test sets $T_1, T_2, ...T_N$, which may contain redundant test cases. Each test case is executed with the program P. Log $EClog$ indexes the test cases of TS, with the traces covered as the execution profile. The execution path of the test cases is then converted into an array of 0s and 1s to for MI calculation. Initially test set $Selected$ and $Redundant$ are empty. Second step is the creation of reduced test set Selected. Read a trace $R$from the log of requirements. For each R, generate a subset of test cases that include R in their execution profile. Select the first test case T_1 from the list, and successive test case T_{next}. Compute the MI between these test cases. If the MI is equal to the $RedundancyLevel$, then the test cases are similar (for a particular coverage criteria). Any one the them (algorithm selects T_1), can be added to the $Selected$ test set and the other one added to the $Redundant$ set. Before adding to the $Selected$ set, it is verified that T_1 is not present in any of these test sets. T_{next}, is added to $Redundant$ if it is does not exists in the set. The process is iterative in nature and all the test cases are analyzed for MI . Finally the $Selected$ test set is returned as the reduced set.
The algorithm 1 $RR2$ has an additional step at step 2. The $RR2$ creates an additional empty set $TCovReq$. The set stores the traces covered by the set $Selected$. Initially both are empty. As the algorithm is executed, the first test case is selected and added to the $Selected$ and its execution profile is added to the $TCovReq$. For the next iteration, the algorithm $RR2$ has an additional step. It checks if the next trace, to be analyzed does not

Algorithm 1: RedundancyReduction RR2

```
Input: $TS$, $TClog$, $EClog$, $RMlog$, $TL$, $Selected$,$Redundant$ , $RL$, $P$, $Testcase$, $RedundancyLevel$, $TCovReq$ = $\phi$
Output: $Selected$, reduced Test suite
for $Testcase \in TS$ do
    $P(Testcase)$ (// execute test case with program)
    Update $TClog$
    Update $EClog$
    Update $RMlog$
for $R \in RMlog$ do
    $TL$ = SelectTestList($R$) (// select test case array for R)
    if $R \exists TCovReq$ then
        continue (// skip to next R)
    for $T \in TL$ do
        Read $Tnext \in TL$
        $MI$ = Mutualinfogain ($T$,$Tnext$ )
        $RedundancyLevel$ = getRedundancyLevel()
        Update $TTrace$
        if $MI == RedundancyLevel$ then
            if $T \nexists Selected$ then
                if $T \nexists Redundant$ then
                    $Selected = Selected \cup T$
                    for $R \in TTrace$ do
                        if $T \nexists TCovReq$ then
                            $TCovReq = TCovReq \cup R$
            if $Tnext \nexists Redundant$ then
                if $Tnext \nexists Selected$ then
                    $Redundant = Redundant \cup Tnext$
        else
            if $T \nexists Selected$ then
                $Selected = Selected \cup T$
                for $R \in TTrace$ do
                    if $T \nexists TCovReq$ then
                        $TCovReq = TCovReq \cup R$
            if $Tnext \nexists Selected$ then
                $Selected = Selected \cup Tnext$
                for $R \in TTrace$ do
                    if $T \nexists TCovReq$ then
                        $TCovReq = TCovReq \cup R$
    return $Selected$
```

Algorithm 2: Functions for RedundancyReduction RR and RR2

```
RequirementsCovered($TestList$) $RequirementsCoveredSet = \phi$ for $T \in TestList$ do
Read $EC$ from $EClog$
for $R$ in $EC$ do
    if $R \nexists RequirementsCoveredSet$ then
        $RequirementsCoveredSet = R \cup RequirementsCoveredSet$
return $RequirementsCoveredSet$
MutualinfoGain ($TestList1$, $TestList2$) $Entropy$ = H($TestList1$)
$CondEntropy$ = CH($TestList1$, $TestList2$)
return ($Entropy$ - $CondEntropy$ )
```

exists in $TCovReq$. The traces present in the set $TCovReq$ are the ones that are covered by test cases of the $Selected$, and hence skipped from the redundancy analysis. The other difference in the two algorithms RR and $RR2$ is the addition of traces to the set $TCovReq$ once a test selected is selected. The additional overhead of $TCovReq$ can reduce the execution process by skipping the MI analysis.

Algorithm 3: RedundancyReduction RR

```
Input: $TS$, $TClog$, $EClog$, $RMlog$, $TL$, $Selected$, $Redundant$ , $RL$, $P$, $Testcase$, $RedundancyLevel$
Output: $Selected$, reduced Test suite
for $Testcase \epsilon TS$ do
    $P\ (Testcase)$ (// execute test case with program)
    Update $TClog$
    Update $EClog$
    Update $RMlog$
for $R \epsilon RMlog$ do
    $TL$ = SelectTestList($R$) (// select test case array for R)
    for $T \epsilon TL$ do
        Read $Tnext \epsilon TL$
        $MI$ = Mutualinfogain ($T$,$Tnext$ )
        $RedundancyLevel$ = getRedundancyLevel()
        if $MI == RedundancyLevel$ then
            if $T \nexists Selected$ then
                if $T \nexists Redundant$ then
                    $Selected = Selected \cup T$
            if $Tnext \nexists Redundant$ then
                if $Tnext \nexists Selected$ then
                    $Redundant = Redundant \cup Tnext$
        else
            if $T \nexists Selected$ then
                $Selected = Selected \cup T$
            if $Tnext \nexists Selected$ then
                $Selected = Selected \cup Tnext$
    return $Selected$
```

4. EXPERIMENTAL STUDY

The subject programs we considered for our study are benchmark classes from the experimental work of Polo [18] and triangle class by Sthamer [19] . The repository is quite popular with research community. The classes were analyzed for their structure and instrumented with *etoc* [20]. The second level of instrumentation was done manually to obtain additional information. Mutants were created with eclipse [21]. The experiments were conducted with *NetBeans* [22]. The classes differ in complexity and size. The class triangle classifier, takes three inputs and based on the values gives output if a triangle can be formed or not, and if a triangle can be formed, then which type(obtuse, right angle). The program has many level hierarchy and the test suite size is huge compared to others. The classes are comparatively less complex, but differ from each other. We implement the RR and RR2 algorithms on all the classes with test suites of different sizes, generated from a random pool of test data. The execution profile obtained is converted to the diagnostic matrix and then for each trace, the test cases are filtered. The selected test data are added to the test suite "$Selected$", and the redundant ones are added to the "$Redundant$". The results of the experiments are discussed below in section 5. The algorithms are compared by executing first set of test suites with RR and recording the data. The same set of test suites is executed with RR2 and the results are recorded, for coverage, time and test suite size, percent reduction in TS, and loss of mutant coverage.

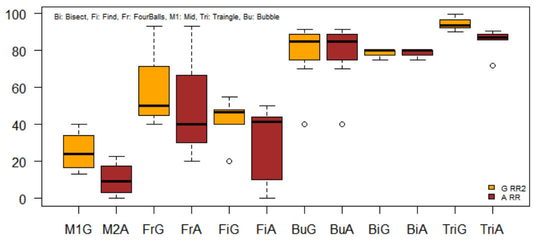


Fig. 2. Reduction of size in test suite after the implementation of the algorithms RR and RR2 as compared to the original with test suite. The plot depicts the two algorithms with RR as

5. RESULTS AND DISCUSSION

In this section we present the results of the experiments along with the discussion.

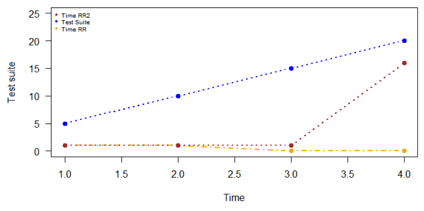
Test suite size reduction. The statistical data for the all the classes is shown together in figure 2. The results of the algorithm RR2 are shown as orange box-plots and RR in brown for the series of test suits of various size. Test suite size reduction for the both the algorithm RR and RR2 calculated by equation 14. The figure shows the *Rsize* for each class for both the algorithms. It can be seen that for all the classes, algorithm RR2 performed consistently better than RR, with same coverage of requirements, other than few outliers. Figure 4 shows the class level comparison of test suite size. For each class the size of original test suite and the "Selected " test suite of RR and RR2 are shown as box plotters. It can be seen that for each class the there is a significant reduction in size of the resultant test suites. The size of the test suite of algorithm RR2 is either same or less than the test suite of RR2, not bigger.

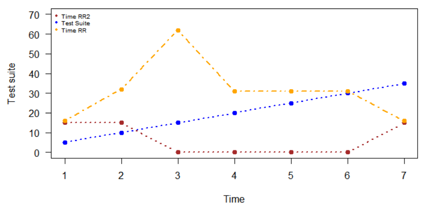
Test suite Coverage. The table shows that their is no major loss of coverage after implementing the algorithms RR and RR2. However, it can be seen that with same coverage, the test suites of RR2 are smaller than the test suites of RR.

Execution Time. To get complete picture of the efficiency and effectiveness, the execution time for both the algorithms were recorded. It can be seen that there is no significant time consumption by RR2 as compared to RR. Figure 3 shows that graphs for all the classes, individually. The data is shown as lines, along with the size of the original test suite. However no pattern can be drawn as many factors can influence the execution time(complexity of program, quality of test data).
Table 4 shows details of the statistical data for all the classes. Here TS is the size of the original test suite which has to be optimized. For each class we conducted experiments on test suites of different sizes ($t1$, $t2$,$t3$...). For each test suite, we implemented the algorithms RR and RR2. The execution time for each execution

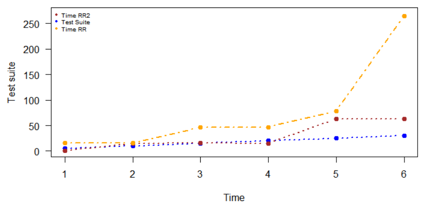
Table 4. Class level Comparative Analysis

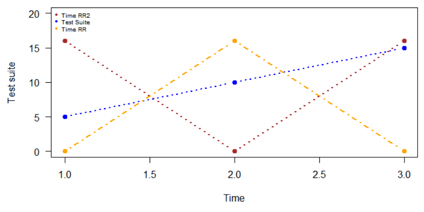
Class	TS	Algorithm	Time	Reduced	Redundant	Rcov	$Rsize$	MutantLoss
MID	t1	RR2	50	3	2	16	40	13.3
		RR	80	5	0	16	0	13.3
	t2	RR2	235	13	2	16	13.3333333333	13.3
		RR	93	14	1	16	6.6666666667	13.3
	t3	RR2	891	20	5	16	20	13.3
		RR	322	22	3	16	12	13.3
	t4	RR2	1922	25	10	16	28.5714285714	13.3
		RR	250	27	8	16	22.8571428571	13.3
FOUR	t1	RR2	1	3	2	6	40	6.6
		RR	15	4	1	6	20	6.6
	t2	RR2	16	5	5	8	50	6.6
		RR	15	6	4	8	40	6.6
	t3	RR2	47	1	14	0	93.3333333333	6.6
		RR	16	1	14	0	93.3333333333	6.6
FIND	t1	RR2	125	3	2	16	40	40
		RR	15	5	0	16	0	40
	t2	RR2	110	8	2	16	20	40
		RR	47	9	1	16	10	40
	t3	RR2	203	8	7	16	46.6666666667	40
		RR	63	9	6	16	40	40
	t4	RR2	422	9	11	16	55	40
		RR	125	10	10	16	50	40
	t5	RR2	609	13	12	16	48	40
		RR	140	14	11	16	44	
	t6	RR2	1032	16	14	16	46.6666666667	40
		RR	172	17	13	16	43.3333333333	40
Bubble Sort	t1	RR2	38	3	2	9	40	60
		RR	49	3	2	9	40	60
	t2	RR2	5	3	7	9	70	60
		RR	34	3	7	9	70	60
	t3	RR2	9	3	12	9	80	60
		RR	96	3	12	9	80	60
	t4	RR2	22	3	17	9	85	60
		RR	34	3	17	9	85	60
	t5	RR2	21	3	22	9	88	60
		RR	48	3	22	9	88	
	t6	RR2	11	3	27	9	90	60
		RR	82	3	27	9	90	60
	t7	RR2	23	3	32	9	91.4285714286	60
		RR	103	3	32	9	0	60
BISECT	t1	RR2	6	1	4	5	80	30
		RR	8	1	4	5	80	30
	t2	RR2	8	2	8	5	80	30
		RR	18	2	8	5	80	30
	t3	RR2	15	3	12	5	80	30
		RR	29	3	12	5	80	30
	t4	RR2	16	5	15	5	75	30
		RR	19	5	15	5	75	30
Triangle	t1	RR	109	14	36	34	72	3 0
		RR2	47	5	45	29	90	38
	t2	RR	93	14	86	34	86	30
		RR2	47	8	92	34	92	30
	t3	RR	167	20	130	36	86.6666666667	28
		RR2	90	9	141	35	94	30
	t4	RR	520	22	178	37	89	28
		RR	94	11	189	36	94.5	30
	t5	RR2	618	31	219	39	87.6	20
		RR	187	17	233	38	93.2	20
	t6	RR2	619	33	267	39	89	18
		RR	156	21	279	39	93	18



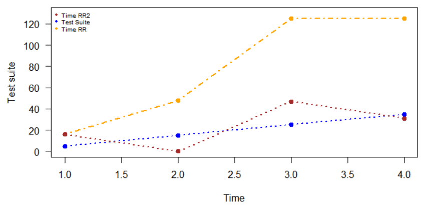


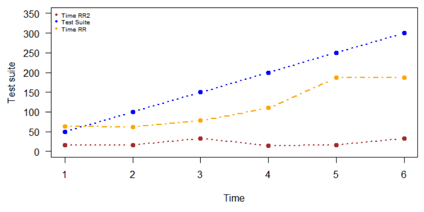
(a) Bisect (b) Bubble sort





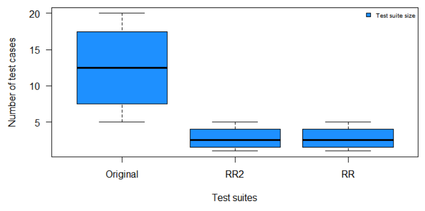
(c) Find (d) Four Balls

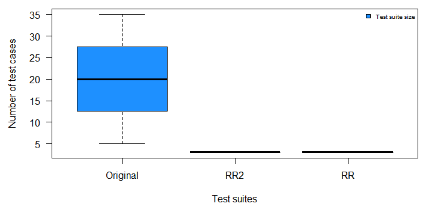




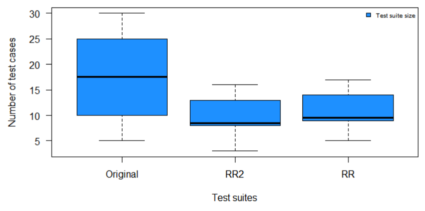
(e) Mid (f) Triangle

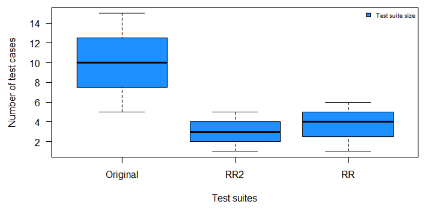
Fig. 3. Time of RR and RR2 algorithms with Test Suites



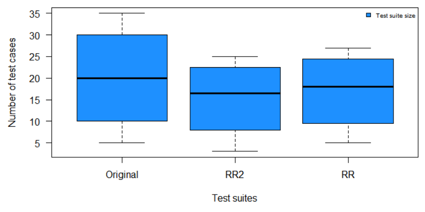


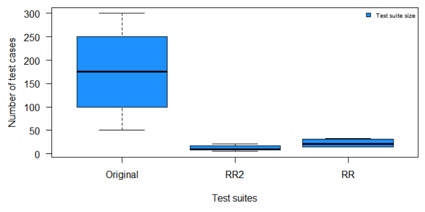
(a) Bisect (b) Bubble sort





(c) Find (d) Four Balls





(e) Mid (f) Triangle

Fig. 4. Test suite size for Original, RR and RR2

is shown as column Time. The reduced, optimized test suite size is shown in the column Reduced, while the redundant test suite size is shown under heading $Redundant$. The Rcov is the coverage for each and every test suite executed and $Rsize$ is the percent of reduction in test suite. The $MutantLoss = numberofmutantsnotcovered/totalnumberofmutants$. Compared to the mutation score of the original test suite, there is no significant loss of mutant coverage. The table shows for a given a test suite, comparing the two algorithms, there is no loss of coverage. However, there is no trend in the execution time. The results show that RR2 generates smaller test suites. Here we elaborate our discussion to other topics. The most important concern is the overhead of the set up for redundancy reduction. The question of "what are the overheads and how are they helpful?". The inputs of the algorithms are very basic and one time investment. The set up need to be generated once, and can be used with every test data generation with every version of the software. The return is in the form of saving of time, resources and human efforts required to analyze the outputs.

Implementation phase. The most important thing is the time or phase of the development cycle, where the algorithms are implemented. The implementation in the first phase of test data generation will save more time, resource and cost in each and every run as compared to the implementation at the later phases (regression testing). The table 4 show that the number of test cases, that are redundant is huge. This shows the amount of time and human efforts saved (execution cycle, storage, analysis of outputs and maintenance). From the results it is clear that the algorithm RR2 is good at eliminating more test cases, which is due to the fact that some times, the test size of reduced is less once the coverage is complete, the rest are treated as redundant. However, for RR, the test suite size is more as it included that test cases which cover the same requirement, but they are far distant. The algorithm RR, checks redundant test for each and requirement. This can be one reason that in many cases, the execution time is more. The algorithm RR implemented at the test data generation phase may add more test cases, but filters the redundant ones efficiently. This may help the tester and developers to have a good diversity with good coverage and minimal size. The implementation of RR2 at the later stages can help the testers and developers to a smaller test suite, as the test suite grows significantly in later stages.

Criticality of testing time and resources. The algorithms can be implemented depending on the criticality of testing time. If the time allows, the algorithm RR can give a good filtering of the redundant test cases. If the testing time or resources are constraint, then RR2 can be implemented to obtain a smaller test suite. In RR2,objective of the filtering process to obtain the desired coverage. Once the objective is fulfilled, the filtering stops. RR2 can be a savior at the time of crisis.

6. RELATED WORK

The work by Jones[2] presents a test suite reduction algorithm, designed for MC/DC coverage criteria. The test cases that are weak, are identified and removed, creating a test suite of strong test cases [2]. Based on re-ordering of test sequence, Pan [3] propose a number of strategies to reduce test suite. Test suites reduction techniques based on coverage are better than random reduction [23]. The on-demand reduction technique allows the test suite to be reduce to achieve a predefined level of coverage and fault detection ability[7]. The test cases that do not cover any unique requirement, may not add to the fault finding capability [24].The fault detection capability can be maintained after test reduction by adding additional test cases, which may increase the size and redundancy of test suite [1]. Yang [25] in his work has defined testing as bringing down the uncertainty in software. Entropy can be a syntactic independent criteria for coverage for any testing artifact[25]. Information theory techniques can be successfully applied in test data generation [26]. Yang [27], suggested the application of entropy for the comparison of test cases covering different criteria. Miranskyy [28] applied entropy to compare traces.

7. CONCLUSION

The paper presents an empirical study of two proposed algorithms for filtering the redundant test suites. The initial results show that the proposed algorithms RR and RR2, are efficient at reducing the test suites and maintaining the required coverage. There is marginal difference in the mutant loss between the RR and RR2. The algorithms suggested are based on information theory based metrics to compare the test cases and can be easily embedded with other heuristics or can be implemented as stand alone. The overhead of the setup is a one time investment, which is low as compared to amount of saving due to elimination of redundant test cases. The limit of the study is that the test suites were executed on the same version. The future work includes the execution of all the test cases on various versions for analysis for efficiency. The algorithms need to be implemented for the black box testing and other testing strategies, along with the implementations on the different phases of development cycle.

8. REFERENCES

[1] Jeffrey, Dennis, and Neelam Gupta. "Improving fault detection capability by selectively retaining test cases during test suite reduction." IEEE Transactions on software Engineering 33.2 (2007).

[2] Jones, James A., and Mary Jean Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage." IEEE Transactions on software Engineering 29.3 (2003): 195-209.

[3] Pan, Jie, and Loudon Tech Center. "Procedures for reducing the size of coverage-based test sets." Proceedings of International Conference on Testing Computer Software. 1995.

[4] Zhu, Hong, Patrick AV Hall, and John HR May., Software unit test coverage and adequacy. Acm computing surveys (csur) 29.4 (1997)

[5] Rothermel, Gregg, et al. "Empirical studies of testsuite reduction." Software Testing, Verification and Reliability 12.4 (2002): 219-249.

[6] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology, 2(3):270285, July 1993.

[7] Hao, Dan, et al. "On-demand test suite reduction." Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 2012.

[8] Shannon, Claude Elwood, A mathematical theory of communication, ACM SIGMOBILE Mobile Computing and Communications Review 5.1 (2001)

[9] Cover, Thomas M., and Joy A. Thomas, Information theory and statistics, Elements of Information Theory (1991)

[10] Wong, Weichen Eric, On mutation and data flow, Diss. Purdue University, (1993)

[11] Patrick, Matthew Timothy, Mutation-Optimised Subdomains for Test Data Generation and Program Analysis, Diss. University of York, (2013)

[12] Mathur, Aditya P. Foundations of Software Testing, 2/e. Pearson Education India (2008)

[13] Papadakis, Mike, and Nicos Malevris. Automatic mutation based test data generation. Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. ACM, pp. 247-248 (2011)

[14] May, Peter Stephen. Test data generation: two evolutionary approaches to mutation testing. University of Kent, (2007)

[15] Mathur, Aditya P., and W. Eric Wong. An empirical comparison of data flow and mutationbased test adequacy criteria. Software Testing, Verification and Reliability 4.1, pp. 9-31 (1994)

[16] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, An Extended Overview of the Mothra Software Testing Environment, in Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA88). Banff Alberta,Canada: IEEE Computer society, pp. 142151 (1988)

[17] K. N. King and A. J. Offutt, A Fortran Language System for Mutation Based Software Testing, Software:Practice and Experience, vol. 21,no. 7, pp. 685718 (1991)

[18] Polo, Macario, Mario Piattini, and Ignacio Garca Rodrguez de Guzmn. Decreasing the cost of mutation testing with second-order mutants. Softw. Test., Verif. Reliab. 19.2 , pp. 111-131 (2009) available at http://www.inf-cr.uclm.es/www/mpolo/stvr/.

[19] Sthamer, Harmen-Hinrich. The automatic generation of software test data using genetic algorithms. Diss. University of Glamorgan, (1995)

[20] Tonella, Paolo. Evolutionary testing of classes. ACM SIGSOFT Software Engineering Notes. Vol. 29. No. 4. ACM,pp. 119-128 (2004)

[21] Eclipse URL: https://eclipse.org/juno/ Retrieved 8 August (2016)

[22] NetBeans URL: https://netbeans.org Retrieved 8 August (2016)

[23] Rothermel, Gregg, et al. ”Empirical studies of testsuite reduction.” Software Testing, Verification and Reliability 12.4 (2002): 219-249.

[24] Wong, W. Eric, et al. ”Effect of test set minimization on fault detection effectiveness.” Proceedings of the 17th international conference on Software engineering. ACM, 1995.

[25] Yang, Linmin, et al., Entropy and software systems: towards an information-theoretic foundation of software testing., Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, (2010)

[26] Campos, Juan, et al, Entropy-based test generation for improved fault localization, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on.

[27] Yang, Linmin, Zhe Dang, and Thomas R. Fischer, Information gain of black-box testing, Formal aspects of computing 23.4 (2011)

[28] Miranskyy, Andriy V., et al, Using entropy measures for comparison of software traces, Information Sciences 203 (2012)