Comparing Test Case Redundancies in Manually Written Versus Automatically Generated Test Suites

Mubaraka N. Parekh
Department of Computer Science
University of Colorado, Colorado Spring
Email: mparekh@uccs.edu

Kristen R. Walcott-Justice
Department of Computer Science
University of Colorado, Colorado Spring
Email: kjustice@uccs.edu

ABSTRACT

Software testing is an important phase in software development where test suite generation is an expensive process in terms of time and analysis. Tests generation should be fast, efficient and effective in terms of quality and time. Writing such test cases is challenging and require more human effort. Thus automated test generation techniques have been developed to ease the load of software developers but its effectiveness and efficiency is very little studied. Automated test generation is fast as compared to manual test creation but its quality should also be compared in terms of time and effectiveness. As test suites are run frequently during regression testing, redundancies in test cases within test suite can negatively impact the overall execution time.

In this research we compare the quality of manually to automatically generated test suites on the basis of redundancy in test cases present using 10 real world applications. Our results show that on average, automated tool as compared to manual produced 13% less 100% redundancy, 75% more partial redundancy and 26% more overall redundancy in test cases.

1. INTRODUCTION

Software testing is an important phase in software development process where the program defects are identified to improve program correctness. There are two approaches used by the developers for generating test suite for a program. As a traditional approach, test cases are manually written by the quality assurance team. Manually written test suites are of high quality but require more human effort, time and maintenance, making the test suite generation for the program more expensive. In general, test suites are requried to be more fast in terms of its time of generation, effectiveness and efficiency. To create such test suites manually is a challenging task. Hence, researchers have spent a lot of time in developing tools that can completely automate the process of test suite generation [16] to ease the load of software testing team. Using automated tools is an alternate approach used for test suite generation. There are several automated tools such as CodePro [1] and EvoSuite [8] which can generate complete test suite within a short period of time. The obtained automatically generated test suites from automated tools reduce time, require less human effort and reduce test suite maintenance cost.

Generation of the test suites using automated tools is fast as compared to manually developed test suites, but automatically generated tests must also be compared to manually written tests in terms of quality. The research by Kracht et al. [15] compares the quality of test suites generated from EvoSuite [8] and CodePro [1] to manual test suite in terms of their branch coverage, mutation score and test suite generation time for 10 different open source applications. They obtained an average branch coverage of 31.5% for manual test suites and 31.86% for automated test suites. The average mutation score was 42.12% for manual and 39.89% for automated test suites. Their results show that the automated tool can generate test suite of similar quality as manually written test suite.

Quality of test suite should be checked both in terms of time and effectiveness metrics. Test suites generated by automated tools have been found to be generating significantly more test cases than manually written test suites. The automated tools CodePro [1] and EvoSuite [8] generates test suite 16.4% larger in code size than manually written test suites [15]. Other tools such as JUnit factory, Randoop and JCrasher, on average, generates 85.13%, 97.3% and 98.4% respectively more lines of code compared to manual test suites [3]. In regression testing environment, test suites are often run very frequently. The efficiency of the test suites in terms of execution time is of particular importance [2, 6, 7, 19] and redundancy in test cases within the test suite can negatively impact the overhead of test execution time.

Over several versions in the development of the software, additional tests are added to manually written test suites to maintain adequate coverage and to save time and resource for retesting the software every time it is modified. During this process some test cases become redundant with respect to the testing requirements.

Test suites either manually or automatically generated, with higher redundant test cases, has comparatively longer execution time and will also consume more resources. Therefore, it is important to know the number of redundant test cases that exists in a test suite. However, redundant test cases are always known as those test cases that completely cover the same statements as other test case and does not include those test cases that partially cover the same statements as covered by other test case in the test suite. Hence, to get an in-depth knowledge about the existing redundancy in a test suite, it is important to consider not only the complete redundant test cases but also consider those test cases that are partially redundant to other test cases in the test suite.

The quality of automatically generated test suite compared to manually developed test suite in terms of redundant test cases is unknown. This comparison between manually and automatically developed test suites in terms of redundancy will give a better insight to the developers for using either of the two approaches for test suite development. The use of test generation technique that generates less redundancy can be encouraged with respect to managing test suite size along with saving execution time and resource during the software development.

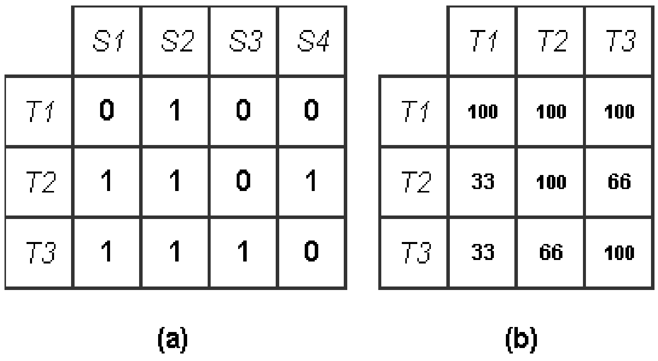
In this research, we compare and analyze manually versus automatically generated test suites from 10 different open source applications on the basis of redundancy in the test cases present in these test suites. We analyze redundancy in the test cases in two ways- 100% and partial redundancy. We refer 100% redundant test to the test cases that test exactly the same statements as the other test or are duplicate of the other test. We refer partially redundant test to the test case that test partially the same statements as the other test case. Partial redundancy is this research is represented as the percentage redundancy of one test case to the other. For example, 80% redundant test is the one that test 80% same statements tested by other test in the test suite.

In summary, this paper's main contributions are:
• An identification of number 100% redundant test cases in test suites using our redundancy finder algorithm (Section III);
• An identification of test execution time savings of test suites after elimination of 100% redundant test cases (Section III);
• A percentage-wise redundancy identification of test suites using correlation matrix algorithm (Section III);
• A comparison and discussion of overall redundancy in test suites, manually to automatically generated. (Section IV)

2. BACKGROUND WORK

In this section, we discuss the two ways of generating test cases i.e. manual and automatic and define the terms that are frequently used in this paper- statement coverage, statement coverage matrix, redundant test case and correlation matrix.

Test Generation Techniques: Manual and automatic are the two ways that are widely used as the test generation techniques. Manually created test suites are manually written by the developers of the program where as the automatically created test suites are generated automatically using automatic test generation tools. For creation of manual test suites, developing team has their own standards set for writing test cases that would maintain different coverages such as statement or branch and fault-finding goals. For generating automatic test suites, there are several techniques used which are most commonly divided into Deterministic and Learning-based. Deterministic test generation technique uses method parameters and basic source code path creating skeletons of test cases. For instance, CodePro test generation tool creates a test class and respective test methods for a given class by analyzing each method and argument and then creating a test that exercises each line of code using combination of static analysis and dynamically execution of the code to be tested. Learning algorithms used by test generation tools improves the overall quality of the test suite. This paper uses EvoSuite [8], ranked first in SBST 2013 and 2016 Tool Competition [10], which uses Genetic Algorithm. The individuals of the population are the test suites of each class. The algorithm works by selecting the test suites based on the branch coverage fitness and then applying crossover and mutation operators to select the test suites for each generation. The fitness is improved generation by generation until a solution is found or a specific set branch coverage has been reached.



	S1	S2	S3	S4
T1	0	1	0	0
T2	1	1	0	1
T3	1	1	1	0

(a)

	T1	T2	T3
T1	100	100	100
T2	33	100	66
T3	33	66	100

(b)

Figure 1: (a) Statement Coverage Matrix, (b) Test Case Correlation Matrix

Statement Coverage: It is a line coverage that checks each and every statement of the code that is executed and also that is not executed and then covers only the true conditions. It basically identifies which statement in a method or class have been executed. The actual benefit of using statement coverage is to identify which parts of the code have not been executed.

Statement Coverage Matrix: This matrix represents the statement coverage by each test case in the test suite. Figure 1 (a) is a statement coverage matrix with each row representing a test case and each column a line-of-code. If a test case covers a particular line of code, it is represented as '1' otherwise a '0' in the respective matrix cell.

Redundancy in Test Case: We define redundancy in test cases in two ways- 100% and percentage-wise. Figure 2 shows example of redundant test cases. Diagram (i) shows test B is a proper subset of test A, covering all statements covered by B and so B is 100% redundant to A. Diagram (ii) shows test A and B are equal, A and B covering exactly same statements and so A and B are 100% redundant to each other. Diagram (iii) shows test A and B intersect with

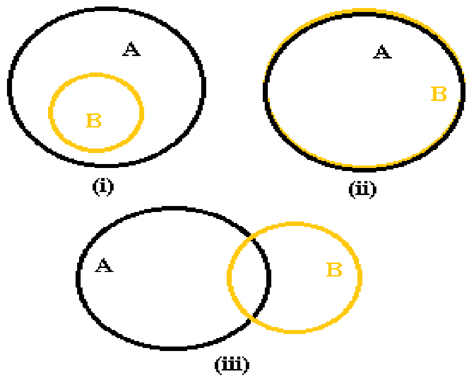


Figure 2: (i) Test B is 100% redundant to Test A, (ii) Test A and B are 100% redundant to each other and (iii) Test A and B are partially redundant to each other.

each other, covering certain number of statements in common. For example if A covers more statements than B, A can be 33% redundant to B and B can be 50% redundant to A. Their redundancy to each other is different depending on the number of same statements covered to the total statements tested by each test case.

Correlation Test Matrix: This matrix is a relation matrix where each row and each column represent a test case making it n∗n matrix where n is total number of test cases in a test suite. It represents relation among test cases by showing how much one test case tests the same statements as the other. The relation is represented as a percentage of test statement similarity in two test cases under comparison as explained in Figure 2 (iii). Figure 1 (b) is a correlation matrix view of the statement coverage matrix (a). Test cases in row are compared with the test cases in column. For example, correlation of test T_3 with other test: test case T_3 covers 33% same statements as T_1, 66% as T_2, and 100% as T_3.

3. EVALUATION

In this section, we evaluate the quality of manually and automatically generated test suites using our evaluation approach shown in Figure 3. We first find statement coverage matrix and then using our algorithm we find redundancies in test suites. For each application, manual and automated test suites, we compare number of 100% redundant test cases, test suite execution time after removing 100% redundant test cases and other number of partial redundant test cases.

3.1 Experimental Subjects

To evaluate our work we use 10 different open source applications from the SF110 code suite [9]. These applications were selected due to their size and due to the existing sets of manually generated and automatically Evosuite [8] generated test suites. The test suites have JUnit test cases, a unit testing framework for the Java Programming Language. Table 1 provides a list of these applications and their information about the lines of code, coverage and number of test cases in their manual and automated test suites respectively. These data is measured using the code coverage tool EclEmma [12].

All the programs have different coverage with different number of total test cases in their respective manual and automatic test suites. Lavalamp has the highest coverage with manual test suite and Jni-inchi has highest coverage with automatic test suite. Jdbacl has the maximum number of test count with 201 test cases in its manual test suite. Jsecurity has the maximum number of test count with 509 test cases in its automatic test suite.

3.2 Implementation

All experiments were performed on Windows operating system, a 2.4GHz Intel(R) Core(TM)i5 processor and with 6GB main memory.

The manually and automatically generated test suites were first operated using Gzoltar library version 1.6.0-SANPSHOT, a command line version of Gzoltar eclipse plugin [5], in order to generate a statement coverage matrix for each test suite. Gzoltar instruments the classes and works with Java bytecode. It executes all JUnit test cases and keeps record of the lines that are executed by each test case and based on that it generates the coverage matrix.

To find the tests that are 100% redundant in a test suite, we developed a Java algorithm shown in Algorithm 1 which checks each test with other tests and gives the number of tests that are duplicate or subset of other test case in a test suite. The algorithm finds 100% redundant tests such that the initial and final coverage would remain exactly same if the tests are removed from the test suite. Algorithm 1 also represents the function that finds coverage of the test suite.

Initially, the algorithm converts the statement coverage matrix in to a covMatrix 2D array. The function findFullRedundantTest() finds 100% redundant test cases using this covMatrix. It compares two test cases by iterating through each row (test case) and each column (statement) in the covMatrix. It saves all the tests that are subset of a test or equivalent to a test in a redTest array. The algorithm validates that the tests are redundant using findCoverage() function. It calculates the coverage of the given coverage matrix by summing up all the 1s for each column one time and then converting the sum of 1s to percentage of the total code length. For example, let initial test suite with four test cases be $T_i = \langle t_1, t_2, t_3, t_4\rangle$. The algorithm makes a covMatrix[i][j], where i is the number of test cases and j is the number of total statements in the code. Algorithm finds t_1 and t_4 as redundant test cases by iterating through each cell of the covMatrix of particular test case and compare to other test case in the matrix such that $t_1 \subseteq t_2$ and $t_4 = t_3$. The tests t_1 and t_4 are saved in an array to give final set of redundant test cases- $T_f = \langle t_1, t_4\rangle$. Say coverage of a test suite T_x is CT_x. The redundancy in this example is found by keeping the initial coverage equal to final coverage such that $CT_i = CT_i$ - CT_f which is coverage of $\langle t_1, t_2, t_3, t_4\rangle = \langle t_2, t_3\rangle$ (set of tests without redundant test cases $\langle t_1, t_4\rangle$).

All the test suites were then minimized by manually removing 100% redundant test that were found in the previous step. JUnit run time was noted in seconds for each test suite before and after the tests were removed. The 100%

Table 1: Benchmarks with Manually (MTS) and Automatically (ATS) Generated Test Suites

Benchmarks	Source LOC	MTS Coverage %	ATS Coverage %	MTS Test Count	ATS Test Count
`Jnfe`	7,893	4.8	67.7	12	87
`Jsecurity`	25,836	33.3	38.4	105	509
`Jni-inchi`	4,483	71.5	69.0	136	171
`Xisemele`	3,036	84.5	30.5	169	152
`Lagoon`	20,097	15.7	20.4	21	281
`Lavalamp`	2,891	97.0	46.9	151	127
`Jdbacl`	48,013	33.1	0.6	201	19
`diebierse`	4,830	1.2	8.7	3	22
`Netweaver`	66,840	15.3	2.0	141	123
`Schemaspy`	30,463	1.5	15.3	7	174

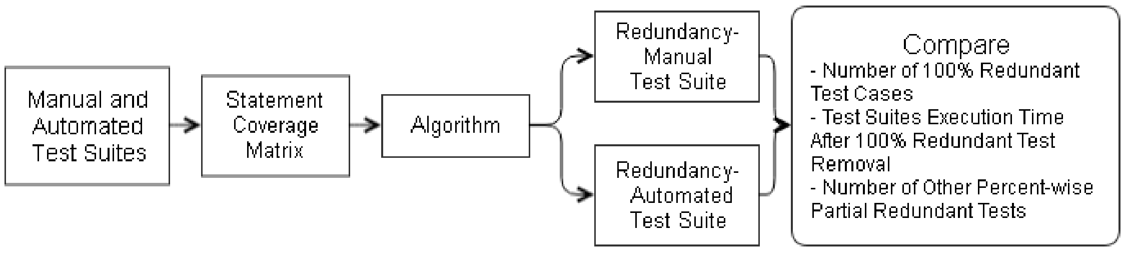

Manual and Automated Test Suites
Statement Coverage Matrix
Algorithm
Redundancy-Manual Test Suite
Redundancy-Automated Test Suite
Compare
- Number of 100% Redundant Test Cases
- Test Suites Execution Time After 100% Redundant Test Removal
- Number of Other Percent-wise Partial Redundant Tests

Figure 3: Experimental Framework

redundant tests were removed keeping the initial and final coverage of the test suites same. This task was performed on Java Eclipse IDE. The coverage was checked using EclEmma coverage tool [12], an Eclipse plugin. EclEmma calculates coverage at run time by instrumenting classes at byte code level. It does the byte code manipulation using a lightweight library ASM which is an all-purpose Java byte code manipulation and analysis framework.

The correlation matrix that shows percentage-wise redundancy was generated for each test suite using the Algorithm 1. As seen in Algorithm 1, the function findCorrltnMatrix() generates correlation matrix for a test suite. This function takes coverage matrix of a test suite as an input. It compares two test by iterating through the matrix and finding the sum of number of 1s that occur in both the tests that are compared. It stores the percentage of similarity of statement tested between two tests in corMatrix, a 2D array which is of size $n * n$ where n is the number of test cases in a test suite. In Figure 1(a), where the test suite is $T = \langle t_1, t_2, t_3 \rangle$. Partial redundancy between each two test cases is compared and stored in a correlation matrix of size 3∗3 as shown in Figure 1(b). Test t_1, t_2 and t_3 are 100% redundant to itself and so the diagonal cells of the correlation matrix show 100 percent. Test case t_1 is 100% redundant to other two test cases in the test suite and so cells t_1t_2 and t_1t_3 are 100 percent each. Next, test case t_2 has only 1 statement out of 3 statements similar to test t_1 and has 2 statement out of 3 statements similar to t_3 and so 33 and 66 percent in the cells t_2t_1 and t_2t_3 respectively. Similarly, for other test cases in the test suite, partial redundancy is represented in a cell of correlation matrix by finding similar statements between two test cases of a test suite. From the generated correlation matrix, for each manual and automated test suite of 10 applications we found the number of tests in between redundancy ranges- 1%-25%, 26%-50%, 51%-80% and 81%-99% respectively. Figure 1(b) has two test case (t_2, t_3) between redundancy range 26%-50% and two test case (t_2, t_3) between redundancy range 51%-80%.

3.3 Results

Our results are based on experiments conducted on 10 applications using two sets of test suites-manual and automated for each application.

1) 100% Redundant Test Cases: For each application, we found the number of 100% redundant test cases for both manual and automatic test suites using our algorithm 1. Figure 4 displays the number of redundant tests in manual and automatic test suite with respect to their coverage. 6 out of 10 applications have higher 100% redundant tests in their manually written test suites. On average for 10 applications, manual has 32 and automated has 28 redundant test cases. This indicate that the automated tools as compared to manual produce 13% less complete redundant test cases than manual generation of test suites. Test suites with coverage higher than 15% has considerably high redundant tests in manually generated test suites as compared to automatic. Number of redundant tests in automatic does not form any specific pattern but are random throughout the coverage. The coefficient of determination R^2 value for manual test suite is 0.73, closer to 1, indicating that the redundancy is much likely to increase with increase in coverage. R^2 value for automatic test suite is 0.208 which shows that it is less likely to follow- increase in coverage will increase redundancy.

2) Execution Time of Minimized test Suite: Table 2 shows

Algorithm 1: Pseudo Code for Redundancy Finder Algorithm

```
covMatrix[i][j] ⟵ Statement coverage matrix file
i ← totalTests
j ← codeLength
function findFullRedundantTests(covMatrix, i, j)
initialize reTest[], count;
for m←0 to i do
    for n←0 to i do
        while $x<j$ do
            if covMatrix[n][x] is 0 and covMatrix[m][j] is 1 then
                break;
            end
            if x is end of line then
                // save redundant tests
                reTest[count]←m;
            end
        end
    end
end
function findCorrltnMatrix(covMatrix, i, j)
initialize corMatrix[][], count;
for m←0 to i do
    for n←0 to i do
        while $x<j$ do
            if covMatrix[n][x] is 1 and covMatrix[m][x] is 1 then
                count++;
            end
            if j is end of line then
                //save correlation percentage
                corMatrix[m][n]← (count in m) %;
            end
        end
    end
end
function findCoverage(covMatrix, i, j)
initialize covArray[], sum, coverage;
for m←0 to i do
    for n←0 to j do
        if covMatrix[m][n] is 1 and covArray[n] is 0 then
            covArray[n] ← 1;
            //add all ones
            sum++;
        end
    end
    coverage ← (sum in i) %;
end
```

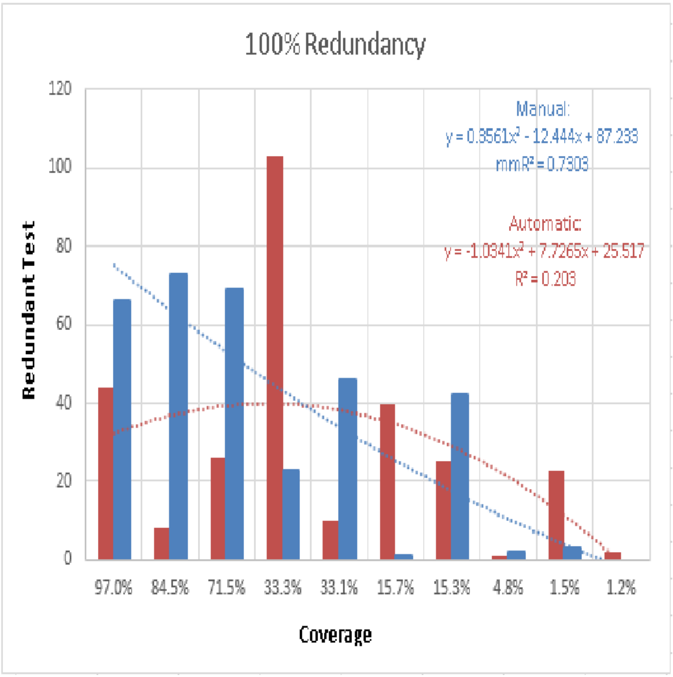


Figure 4: Redundant Test Cases in Manually and Automatically Obtained Test Suites

the time saved from minimized test suites in both manually and automatically generated. As seen from the table, the time saved for manually generated test suite is higher in 8 out of 10 applications. The number of redundant test exceeds in manual test suites and so the time saved from minimization is higher in manual test suites. The results also show that the time saved from minimized manually written test suites is considerably high than automatically generated test suites though the difference in number of tests that are removed from both the test suites is not more. For application Jni-inchi, time saved saved by automated test is 0.014 and by manual is 0.336 which is 24 times higher time saved than automated though only 25 test cases are executed more by automated than manual test suite.

Table 2: Execution Time Saved After Removal of 100% Redundanct Test Cases in Manual (MTS) and Automatically (ATS) Generated Test Suites

Benchmarks	Redundant Tests		Execution Time Saved	
	MTS	ATS	MTS (sec)	ATS (sec)
`Jnfe`	2	8	0.49	0.015
`Jsecurity`	23	103	0.244	0.15
`Jni-inchi`	69	44	0.336	0.014
`Xisemele`	73	10	0.227	0.008
`Lagoon`	1	40	0.03	0.122
`Lavalamp`	66	26	0.334	0.054
`Jdbacl`	46	2	0.154	0.131
`diebierse`	0	1	0	0.003
`Netweaver`	42	23	0.051	0.024
`Schemaspy`	3	25	0.016	0.262

Table 3: Overall Redundancy in Manually and Automatically Developed Test Suites

Benchmarks	Overall Redundancy %	
	MTS	ATS
Jnfe	38.4	81.1
Jsecurity	36.5	22.7
Jni-inchi	29.7	36.8
Xisemele	64.1	27.1
Lagoon	50.7	56.8
Lavalamp	46.0	15.0
Jdbacl	67.1	50.6
diebierse	59.4	68.8
Netweaver	48.8	27.3
Schemaspy	46.7	1.7

3) Other Percentage-wise Redundancy in Test Cases: Figure 5 show four graphs each display results of comparison between both types of test suites as number of tests found in specific percentage redundancy group at different coverage. For all the redundancy groups, the number of redundant test cases are higher in automatically generated test suites than manually created test suites. On average, for groups 1%-25%, 26%-50%, 51%-80% and 81%-99%, automated tool generated 76%, 87%, 50% and 88% respectively more redundant test cases than manually generated test suites. on average, automated tests generated 75% more partial redundancy than manual tests. The R^2 values for manually generated test suites for all groups of redundancy is stronger indicating that the redundancy will increase as the coverage increases where as the R^2 values for automatically generated test suites is much weaker showing less relation between coverage and redundant tests.

4) Overall Redundancy in Test Suites: Table 3 shows the overall redundancy for all the applications in both manually and automatically generated test suites. The overall redundancy in ATS is maximum in Jnfe and minimum in Schemaspy and in MTS is maximum in Jdbacl and minimum in Jni-inchi. The Pearson correlation between MTS and ATS is 0.12 which is very low and does not show a good correlation between the two types of test suites in terms of redundancy. It shows that the manually written test cases are variant as compared to automatically generated test suite which uses algorithms for developing whole test suite. The standard deviation found for the ATS redundancies is 12 and for MTS it is 25 which is twice larger. These scores show that the manual test cases varies extremely the way they are created whereas automatic tools are much consistent in creating test cases and so the redundancies are consistently high.

4. COMPARISON AND DISCUSSION

From the experiment and results, the number of 100% redundant test cases are more in manual than in automatically generated test suites. The results also show that as the coverage increase, the number of redundant tests in the manually written test suites also increase. It can be inferred that during development to achieve high coverage, test cases that are added to the manual test suite of an application are redundant.

The result of execution time that is saved from minimization of test suites show that minimized manually written test suites save considerably more time than minimized automatically generated test suites. The test cases generated by human are variant as compared to the redundant test cases of automatically generated test suites. Automated tools such as EvoSuite uses genetic algorithm. The algorithm uses population of test suite where each individual is a test suite of the class with set of test cases present in that class. The fitness of each individual is decided by its branch coverage. New generation of the population is obtained by performing two operations on the population- crossover and mutation of test cases. After these operations the fitness of the individuals is checked and the individuals with the lowest fitness are removed. Generation after generation the fitness of the individuals in the population improves. Test suite which achieves targeted branch coverage or the maximum branch coverage are kept. Hence, the way the automatic test generation works becomes another reason for the resulting redundancies to be much more consistent in generated test suites.

From results of finding test cases at various redundancy levels, our results indicate that the automatic test suites produce more redundancy than manual. we also observed that in automatically generated test suites number of redundant test cases do not depend on coverage whereas manually written test suites at low coverage have very less redundancy as compared to redundancy at high coverage. This is because automated test suites are generated all at one time where as the manual test suites are gradually build during the software development. At higher percentage redundancy- 81%-99%, automatically generated test suites have considerably more redundancy in test cases than manually generated test suites. Hence, percentage-wise redundancy must also be considered important as 100% redundancy in terms of the execution time of a test suite.

The quality of automated test suites in past research were compared based on branch coverage and mutation score. For 10 applications, the average branch coverage of automated test suite was found 1.14% higher than manual and average mutation score was found 5.29 % higher in manual test suite than automated test suites. This results show that there was not a considerable difference in quality found between the manual and automated test suites. Our experiments find quality difference between these test suites in terms of redundancy. Our results show a considerable difference in existing redundancy in 10 real world application's manual and automatic test suites with average partial redundancy 75% and overall redundancy 26% higher in automatic test suite than manual. Hence, our results show that manually written test suites exceeds in quality in terms of redundancy than test suites generated using automated tool.

5. RELATED WORK

Study by Kracht et al. [15] and Bacchelli et al. [3] compares quality of automated test suites to manual test suites using the branch coverage and mutation score of the test suites. Our study compares manually to automatically generated test suites by performing several comparisons by examining the redundancies in the test cases at different redundancy percentage levels.

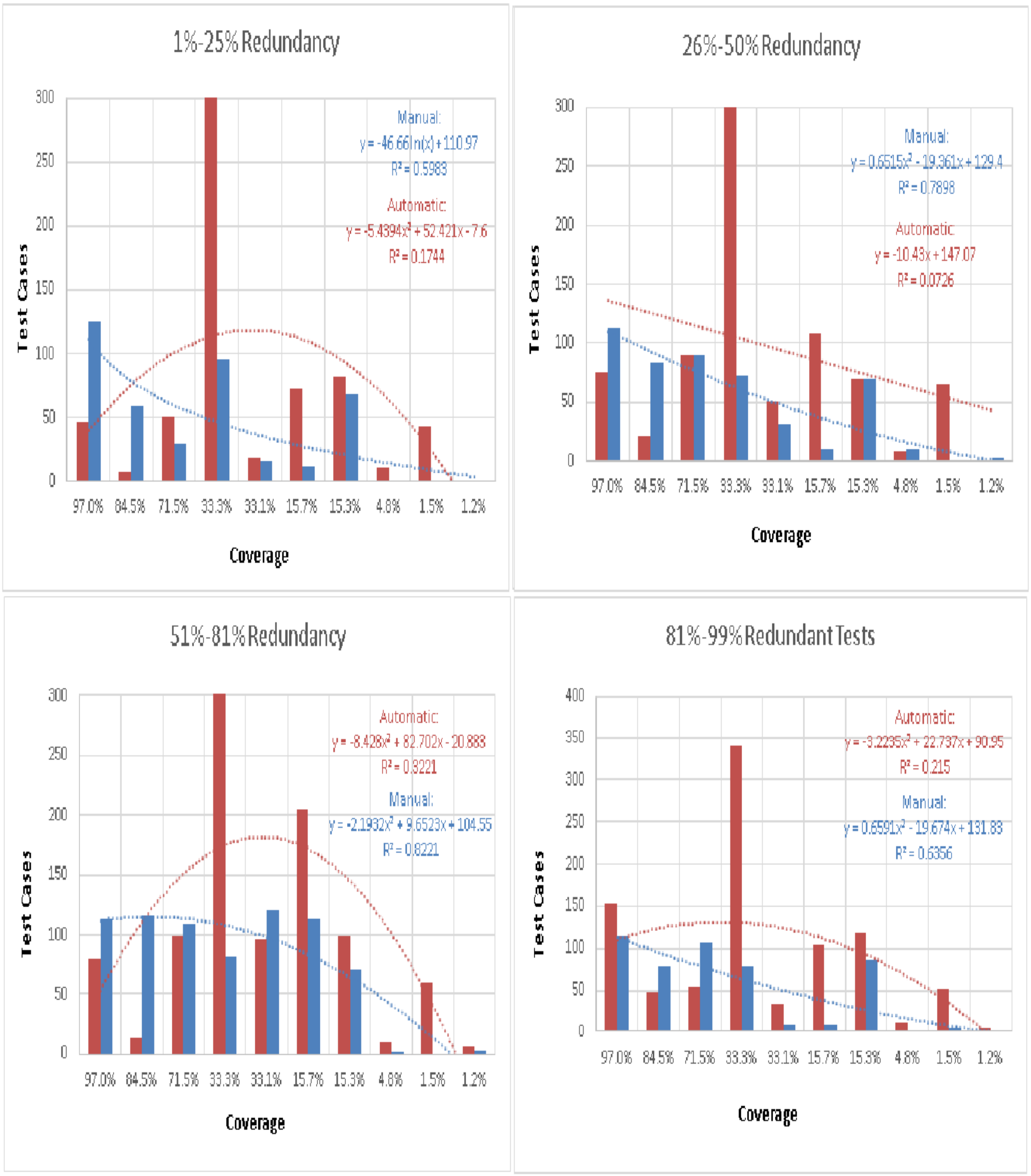


Figure 5: Percentage-wise Redundancy in Test Cases in Manually and Automatically Obtained Test Suites

There are several techniques proposed by researchers for test suite minimization [4, 11, 13, 20]. Test suite minimization considers one or more criteria as specified by the testers and eliminates the redundant test cases from the test suites to reduce the size of the test suite. Our research minimize the test suite by eliminating redundant tests that have 100% redundancy to other test in the test suite and also find redundant test cases at different redundancy percentage level rather than performing minimization so that we can compare redundancies in manual and automated test suites.

CodeCover [17] is an Eclipse plugin coverage tool which facilitates correlation view among test cases in a test suite. This view can only generate a matrix for test to test redundancy comparison for a test suite size of maximum 70 test cases and takes considerably long time to generate this matrix. Our research presents an algorithm that can generate correlation matrix of test suite of size 1000 test cases in less than 3-4 seconds. It also provides detail information on which test case is covered by which other test cases in the matrix and the number of test cases in specified redundancy level.

Tao Xie et al. [21] proposed a framework for detecting redundant unit tests in tools that automate generation of unit tests for Java programs. Their research focuses on finding redundancy only in automatic tools available for test generation where as we compare manual and automatically generated test suites by finding redundancies in both the types of test suites.

Pretschner et al. [18] compares manual and automated test suites generated with or without the development model of the software in terms of error detection, model coverage, and implementation coverage. Though this research compares the efficiency of both types of test suites our goals of comparison are different. We specifically focus on redundancy that lies in manual and automated test suites which can help developers manage the size of the test suites.

Negar Koochakzadeh [14] finds the redundancy in test suites and compares the found redundancy based on automated and manual redundancy decisions using 4 real Java programs. Our research focus on comparing manual to automatically generated test suites by finding redundancy in test suites using our own algorithm which finds redundancy in test cases at different redundancy levels. This helps better understand the redundancy not only at 100% redundancy percentage but at various other percentages such as 50%, 80% and upto 99%.

6. CONCLUSION

There is very little research done on comparing manually written to automatically generated test suites. Automated tools which are sophisticated and uses learning based algorithm such as EvoSuite have been found to produce test suite of similar quality on average as compared to manual. Though the test generation time using such tools is considerably less than manual, the generated test suites are significantly large. As test suites are frequently run during testing, efficiency of test suite is required in terms of its execution time. Redundancy in test cases within a test suite can increase the overhead of execution time. We compare the manual and automatic tests in terms of redundancy in the test cases. We evaluate the quality of automatic and manual test suites by finding 100% redundant test cases, execution time saved from performing minimization of test suites and percentage-wise redundancy. Our results indicate that automated test generation tools produced more redundancy than manual. On average, automated produced 13% less 100% redundant tests than manual test suites. For partial redundancy percentage groups used in this research, automated tools generated on average 75% more partial redundancy than manual tests. Overall redundancy produced by automated tools was found to be 26% more than manually generated test suites.

In future work, we will increase the number of case study applications to get significant statistical results. We will also consider other competitive automated test generation tools such as CodePro and Randoop for comparison with manually written test suites.

7. REFERENCES

[1] Codepro analytix: Java code quality and security analyisis. https://marketplace.eclipse.org/content/codepro-analytix. Accessed: 2016-05-01.

[2] S. Alspaugh, K. R. Walcott, M. Belanich, G. M. Kapfhammer, and M. L. Soffa. Efficient time-aware prioritization with knapsack solvers. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 13–18. ACM, 2007.

[3] A. Bacchelli, P. Ciancarini, and D. Rossi. On the effectiveness of manual and automatic unit test generation. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pages 252–257. IEEE, 2008.

[4] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, pages 106–115. IEEE Computer Society, 2004.

[5] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM.

[6] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 71–82. ACM, 2008.

[7] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages

235–245. ACM, 2014.

[8] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[9] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 178–188. IEEE Press, 2012.

[10] G. Fraser and A. Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 33–36. ACM, 2016.

[11] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical report, Citeseer, 1997.

[12] M. R. Hoffmann. Eclemma-java code coverage for eclipse, 2009.

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

[14] N. Koochakzadeh, V. Garousi, and F. Maurer. Test redundancy measurement based on coverage information: evaluations and lessons learned. In *2009 International Conference on Software Testing Verification and Validation*, pages 220–229. IEEE, 2009.

[15] J. S. Kracht, J. Z. Petrovic, and K. R. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 256–265. IEEE, 2014.

[16] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: The autotest experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261a–261a. IEEE, 2007.

[17] A. H. Patil and D. N. S. Sidnal. Codecover: A codecoverage tool for java projects in ercica.

[18] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.

[19] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[20] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2006.

[21] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 196–205. IEEE Computer Society, 2004.