Test Suite Reduction Using K-means Clustering

By

Jabran Saleem

Fall-2019-MS-CS CMSID 00000318846

Supervisor

Dr. Seema Jehan

Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science (MS CS)

In

School of Electrical Engineering & Computer Science (SEECS) ,

National University of Sciences and Technology (NUST),

Islamabad, Pakistan.

(July 2022)

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis entitled "Test Suite Reduction using K-means Clustering" written by JABRAN SALEEM, (Registration No 00000318846), of SEECS has been vetted by the undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/M Phil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature: [illegible]

Name of Advisor: Dr. Seema Jehan

Date: 21-Feb-2022

HoD/Associate Dean: ________________________

Date: ____________________________________

Signature (Dean/Principal): ____________________

Date: ____________________________________

Approval

It is certified that the contents and form of the thesis entitled "Test Suite Reduction using K-means Clustering" submitted by JABRAN SALEEM have been found satisfactory for the requirement of the degree

Advisor : Dr. Seema Jehan

Signature: [illegible]

Date: 21-Feb-2022

Committee Member 1:Dr. Omar Arif

Signature: [illegible]

Date: 22-Feb-2022

Committee Member 2:Dr. Muhammad Ali Tahir

Signature: Ali

Date: 22-Feb-2022

Committee Member 3:Dr. Qaiser Riaz

Signature: Qaiser Riaz

Date: 21-Feb-2022

Dedication

I would like to dedicate this thesis to my family without whom it was impossible for me to complete my thesis work.

They always support me throughout this course and motivate me whenever I was down and lost hope in me. Thanks to their care, love, support and prayers.

Certificate of Originality

I hereby declare that this submission titled "Test Suite Reduction using K-means Clustering" is my own work. To the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics, which has been acknowledged. I also verified the originality of contents through plagiarism software.

Student Name:JABRAN SALEEM

Student Signature: [illegible]

Acknowledgments

First of all, I would like to pay my gratitude to Allah Almighty who give me strength to conduct this work. I would then like to thank my supervisor Dr. Seema Jehan for her continuous guidance and encouragement. Last but by no means least, I am truly thankful to my parents for their endless moral support throughout the period.

Jabran Saleem

Contents

List of Figures

List of Tables

Abstract

Regression testing involves re-execution of test suite whenever a software undergoes any update. This cost is increased manifolds due to recent continuous integration practices in software industry. There are many approaches proposed by researchers to reduce the execution time of test suite. This thesis examines two variants of K-means clustering algorithm for test suite reduction. The K-means algorithm has been used for grouping of various objects based on similar features and data organization. The suggested algorithms for test suite reduction will cluster those test cases which are testing the same requirements. For clustering, the Euclidean distance has been used to calculate the distance between test cases. Once clustered, algorithm will select the one representative vector or test case from each cluster and add it to reduced test suite. We have tested this algorithm on 43 versions of seven different NPM packages. Out of these, 20 package versions are giving test suite reduction for all alpha values 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50, whereas other package versions are giving test suite reduction for some alpha values. The experimental results have shown that both K-means and K-means++ clustering algorithms produce approximately similar results for test suite reduction with a slight difference in fault detection loss. For some package versions, the fault detection loss is slightly higher in K-means whereas for other package versions, the fault detection loss is little bit higher in K-means++. The async package has the highest test suite reduction which lies between 70% to 80% and fault detection loss lies between 4% to 30% for alpha values of 5-50. Hence, we can state that these algorithms can effectively reduce the size of test suite while having minimal fault detection capability loss. However, we cannot generalize the results because the algorithm still needs to be executed on JavaScript programs with larger test suites. In the future, we plan to implement this algorithm with other distance metrics such as Hamming distance and Levenshtein Edit distance.

Chapter 1

Introduction

In this chapter, we will give an overview of test suite reduction and why should we research in this area? The chapter highlights the problem statement and research questions.

1.1 Background

Regression testing of large-scale software systems is a time-consuming task since it involves re-execution of all test cases from the previous release after every modification. This process entails high cost due to continuous integration practices adapted by the software industry. According to IBM

"Regression testing involves the reuse of all tests to ensure your software experiences no regression - in other words, to ensure that the repair of one defect doesn't break some other feature that worked in the past."

We need some solution to decrease the execution time of test suite. The reduction in the execution cost of test suite will result into high-speed deployment and on-going maintenance of software. This will reduce the time during continuous integration and continuous testing. There are many approaches proposed by researchers to reduce the execution cost associated with regression testing. These approaches can be classified into three types: Test Case Selection, Test Case Prioritization, and Test Suite Reduction. [16].

Test case selection and test suite reduction both reduced the number of test cases of test suite in different manners. In test suite reduction, we aim to take out the unused or unnecessary test cases whereas in test case selection, we re-run the set of test cases which is testing changed features. The last technique is completely different in which we are prioritizing the test cases' execution. Those test cases which are covering most of the source code statements will be executed first.

Test suite reduction approaches can be further categorized into adequate and inadequate approaches. The adequate approach reduces the number of test cases of the test suite while keeping the code coverage equal to original test suite's code coverage. This means that all the test requirements covered by original test suite will also be covered by reduced test suite. On the other hand, an inadequate approach reduces the test suite's size but allows fault detection capability loss. So, if there is good amount of reduction in the execution time and size of original test suite, we are allowed to have less code coverage as compared to code coverage of original test suite but this reduction in code coverage is allowed up-to some defined inadequacy level.

Out of all the algorithms (2-Optimal, GE, and GRE, Greedy, Delayed Greedy) proposed for test suite reduction, the clustering algorithm outperformed all of these. Clustering algorithms greatly reduced the size of the test suite while maintaining the fault detection capability [14]. More research has been conducted in this area to explore better similarity techniques which can cluster test cases more efficiently. Normally code coverage is measured by statement coverage, branch coverage, modified condition or decision coverage; there has been another metric proposed by researchers called mutation score [13]. A tool named Mutode is developed to test the JavaScript programs using mutation testing. In mutation testing, the faults are introduced into source program and new files are created with these faults which we refer to as mutants. Once the mutants are created, the test suite is executed on each of these mutants to check whether it is able to detect faults or not. In the end, we got the mutation score of test suite which is calculated by taking percentage of killed mutants divided by the total number of mutants.

Most of the clustering algorithms for test suite reduction have been implemented with statement coverage, recently there are authors who implemented K-means clustering algorithm with mutation testing [15]. They have used the mutation score as their fault coverage metric. They have implemented the algorithm for Java Programs and showed that the original test suite is reduced 82.5% by maintaining the original mutation score.

1.2 Research Purpose

We aim to generalise whether K-means algorithm reduces the test suite's size not only for Java programs but also for other language programs. With the use of K-means for test suite reduction, we aim to optimize the testing, deployment and maintenance phases of software development life cycle. We have selected 43 versions of seven NPM packages for testing and validation. We have chosen these packages because their test suites are very complex and dynamic in nature. It's more difficult to apply test suite reduction algorithm on these as compared to other frameworks like Java. So, if K-means clustering for test suite reduction performs well on these test suites, then we can more confidently generalize its results for other frameworks. The algorithm will cluster similar test cases based on the Euclidean distance metric. It will reduce the execution cost of regression testing of NPM packages that are widely used by the freelancer community for software development.

1.3 Problem Statement

The aim is to reduce the size of NPM test suite using K-means while keeping the fault detection capability of the reduced test suite within certain threshold.

1.4 Research Questions

In this thesis, we investigate the following research questions:

a) Can we reduce the size of test suite without compromising on fault detection capability?

b) Can we generalize the results of K-means test suite reduction for JavaScript programs?

The remaining part of the thesis is organized as follows: first, the terminology required to understand the discussed thesis, then we put some light on existing work that has been done in the domain of test suite reduction. After that, we describe the proposed methodology and implementation of K-means for size reduction of JavaScript programs' test suite. Afterwards, we discuss the experimental results and challenges faced during implementation. In the end, we describe briefly the outcomes of the thesis and future research directions.

Chapter 2

Preliminaries

In this chapter, we describe some commonly used terms in thesis. It will help in better understanding of the rest of the thesis.

1. **Software Testing:** "Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do. The benefits of testing include preventing bugs, reducing development costs and improving performance." – IBM. According to Ammann and Offutt software testing is known as "Evaluating software by observing its execution" [8, p.13]

2. **Regression Testing:** "Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the System Under Test (SUT) do not interfere with the existing features." [5]

3. **Test Case:** According to Ammann and Offutt, "A test case is composed of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test." [8, p.15]

4. **Test Suite:** A test suite consists of a number of test cases.

5. **Fault:** "A static defect in the software." [8, p.15]

6. **Statement Coverage:** It is a measurement of number of source code lines covered when the test suite is executed.

7. **Branch Coverage:** This coverage criterion ensures that all possible outcomes of decision point are covered when the test suite is executed.

8. **Mutation Operator:** Mutation operator is "A rule that specifies syntactic variations of strings generated from a grammar." [8, p.173]

9. **Mutant:** Ammann and Offutt describes the mutant as "The result of one application of a mutation operator." [8, p.173]

10. **Mutation Testing:** Mutation testing create faulty versions of source code and execute the test suite to identify whether it can detect faults or not. [4] Traditional mutation testing is based solely on two hypotheses:

    a) **Competent Programmer Hypothesis:** "It states that programmers are competent, which implies that they tend to develop programs close to the correct version." [4]

    b) **Coupling Effect:** "It states that Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors." [4]

11. **Mutation Score:** It is calculated by taking percentage of killed mutants divided by the total number of mutants. The formula is given below:

$$\frac{KilledMutants}{TotalMutants} \times 100$$

12. **Killed Mutants:** Given program p and mutated program p', if any test case of test suite produces different result for p' and p, then we can state that the mutant p' is killed. [4]

13. **Survived Mutants:** Given program p and mutated program p', if any test case of test suite produces same result for p' and p, then we can state that the mutant p' is survived. [4]

14. **Clustering:** It is stated as "Given a representation of n objects, find K groups based on a measure of similarity such that the similarities between objects in the same group are high while the similarities between objects in different groups are low." [3]

2.1 Mutation Testing

“Mutation testing is a technique in which faults (mutants) are inserted into an application to assess its test suite effectiveness. It works by inserting mutants and running the application’s test suite’s to identify if the mutants are detected (killed) or not (survived)” [13]. In 1971, Richard Lipton introduced the mutation testing for the first time in his paper [4]. Authors have discussed the theory, process, applications, tools of mutation testing in detail and describe the development timeline of it.

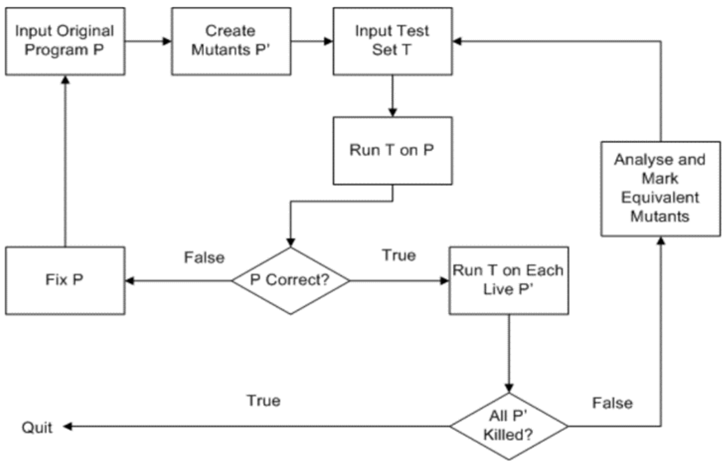


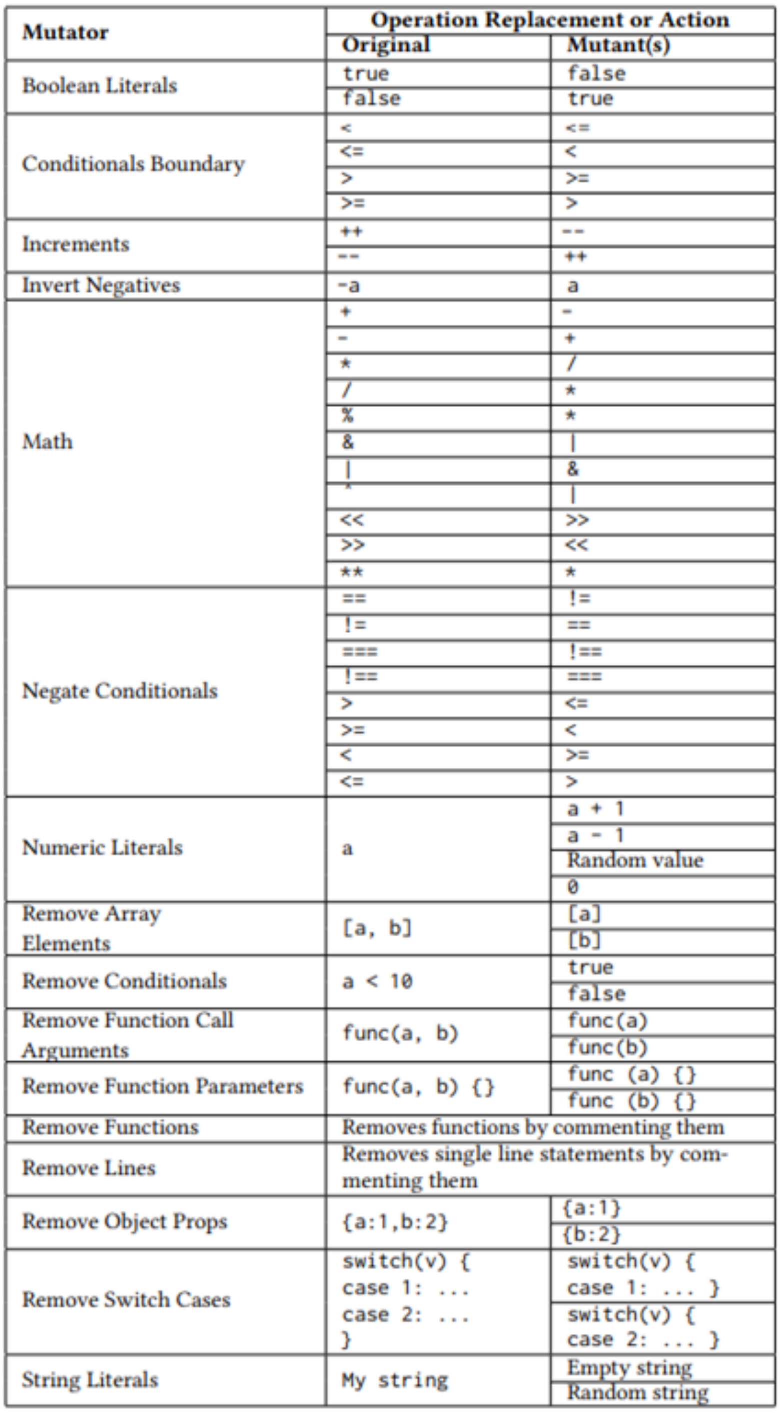
Figure 2.1: Illustration of Mutation Testing [4]

The Figure 2.1 describes the overall process and theory of mutation testing. First, the original program (P) and test suite (T) is given as an input to mutation testing application, the application creates mutants of original program source code, the application runs the test suite over original program and checks whether it is working as expected, after that, the application runs test suite over mutated programs and checks whether mutant are killed or survived. If mutant is killed, then it means that test suite has identified the fault and if mutant is survived, then it means that test suite has not identified the fault. The paper [4] showed that mutation testing become a famous testing technique and there are tools available to apply it on real projects.

One of the tools developed by Diego and Mario for mutation testing over Node.js programs is known as Mutode [13]. They have used 43 mutation operations grouped into 16 mutators in Mutode to target a variety of JavaScript and Node.js applications. We can see the mutators in Figure 2.2. In the following code snippet, we can see an example of mutant where the original program's if condition is changed from greater operator to less than operator.

```
Program (P):                              Mutant (P'):
    if(x > 0)                                 if(x < 0)
    {                                         {
        y = 1                                     y = 1
        return false;                                return false;
    }                                         }
```

Mutode uses npm test command to avoid extra configurations or plugins installation to run tests. Moreover, the mutants run parallel to reduce the time and improve CPU power. The authors have evaluated the performance of Mutode on 12 out of top 20 npm modules. The test suites of these npm packages have received an average mutation coverage of 70.59%.



Mutator	Operation Replacement or Action	
	Original	Mutant(s)
Boolean Literals	true	false
	false	true
Conditionals Boundary	<	<=
	<=	<
	>	>=
	>=	>
Increments	++	--
	--	++
Invert Negatives	-a	a
Math	+	-
	-	+
	*	/
	/	*
	%	*
	&	|
	|	&
	^	|
	<<	>>
	>>	<<
	**	*
Negate Conditionals	==	!=
	!=	==
	===	!==
	!==	===
	>	<=
	>=	<
	<	>=
	<=	>
Numeric Literals	a	a + 1
		a - 1
		Random value
		0
Remove Array Elements	[a, b]	[a]
		[b]
Remove Conditionals	a < 10	true
		false
Remove Function Call Arguments	func(a, b)	func(a)
		func(b)
Remove Function Parameters	func(a, b) {}	func (a) {}
		func (b) {}
Remove Functions	Removes functions by commenting them	
Remove Lines	Removes single line statements by commenting them	
Remove Object Props	{a:1,b:2}	{a:1}
		{b:2}
Remove Switch Cases	switch(v) { case 1: ... case 2: ... }	switch(v) { case 1: ... }
		switch(v) { case 2: ... }
String Literals	My string	Empty string
		Random string

Figure 2.2: Mutators and Operations in Mutode [13]

Chapter 3

Literature Review

This chapter discusses different types of regression testing: test suite reduction, test case selection and test suite prioritization techniques.

3.1 Regression Testing

Regression testing is known as execution of complete test suite after each feature deployment or bugfix. It ensures that software is bug free and work as expected. The following notation is used for regression testing:

"P be the current version of the program under test, and P' be the next version of P. Let S be the current set of specifications for P, and S' be the set of specifications for P'. T is the existing test suite." [5]

As the time passes, the new features are introduced in the software which result into addition of new test cases in test suite. This increases the re-execution time of test suite on each new feature deployment or bugfix. This can slow down the software deployment and maintenance processes. To reduce the regression testing cost, research has been done in three main categories: Test Case Selection, Test Case Prioritization, and Test Suite Reduction.

3.2 Test Suite Reduction

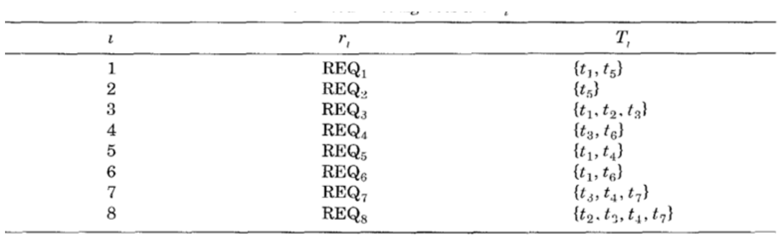
"Given: A test suite, T, a set of test requirements $\{r_1,...,r_n\}$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, $T_1,...,T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i. Problem: Find a representative set, T', of test cases from T that satisfies all r_i's." [5] The problem of test suite reduction is NP complete. Therefore, we need heuristics to solve the problem. It can be further categorized as Adequate Test Suite Reduction and Inadequate Test Suite Reduction.

3.2.1 Adequate Test Suite Reduction

In this test suite reduction approach, we take out the unused or unnecessary test cases while not compromising on the fault detection capability.

HGS Algorithm

M. J. Harrold et al, presented HGS algorithm which finds the reduced test suite based on testing sets (test cases associated with testing requirements) [1]. This algorithm falls under adequate approach which means that test requirements covered by reduced test suite and original test suite remains same. "Given a test suite T having testing requirements $\{r_1, r_2, ...r_n\}$ and associated testing sets for each requirement, the algorithm finds the subset of testing sets that cover all the testing requirements by meeting desired coverage criterion." As this is an NP-hard problem, the authors use heuristics to approximate the reduced test suite. We can run the HGS on example data in Figure 3.1



i	r_i	T_i
1	REQ_1	$\{t_1, t_5\}$
2	REQ_2	$\{t_5\}$
3	REQ_3	$\{t_1, t_2, t_3\}$
4	REQ_4	$\{t_3, t_6\}$
5	REQ_5	$\{t_1, t_4\}$
6	REQ_6	$\{t_1, t_6\}$
7	REQ_7	$\{t_3, t_4, t_7\}$
8	REQ_8	$\{t_2, t_3, t_4, t_7\}$

Figure 3.1: Example Data of Test Cases t_i, Requirements REQ_i, and Related Testing Sets are T_i [1]

First the heuristic selects test cases with minimum cardinality and then it keeps on repeating the process until it reaches the maximum cardinality. If two test cases have same cardinality, then algorithm continue processing of unmarked testing sets of higher cardinality. In the above example, first the algorithm selects t_5 because there is only one single element set T_2. Test case t_5 is added to representative test set. REQ_1 and REQ_2 will be marked as satisfied. Next, the algorithm considers the unmarked testing sets Ti of cardinality two (T_4, T_5, T_6). As t_1 appears in T_5 and T_6 whereas t_6 appears in T_4 and T_6, so there is tie between these testing sets, hence algorithm will continue processing unmarked testing sets of cardinality three (T_3 and T_7). As t_1 appears in T_3, and t_6 not appears in neither of T_3 and T_7, so algorithm choses T_3 and add t_1 in the representative set. Next, the algorithm looks again for T_4 as it is the only unmarked testing set with cardinality two. Again there is a tie between t_3 and t_6, so algorithm moves to T_7 which has cardinality three and has t_3. It is added to representative test suite. After this, all T_i's will be marked, so representative test suite becomes $\{t_1, t_3, t_5\}$

Delayed Greedy Algorithm

Tallam et al. presents the new algorithm for test suite minimization under adequate class [2]. They named this algorithm Delayed Greedy Algorithm. This has overcome the limitations of classical greedy heuristic algorithm for test suite reduction. "Given a test suite T having test cases $\{t_1, t_2, \ldots t_n\}$ and testing requirements $\{r_1, r_2, \ldots r_m\}$, the algorithm finds the minimum number of test cases that cover all the testing requirements." The greedy algorithm follows heuristic which makes the best optimal choice locally at each stage aiming to get global optimum solution. The authors have demonstrated that the reduction in the size of test suite can be done using a greedy heuristic algorithm, but the reduced test suite still contains the redundant or unnecessary test cases and does not have an optimal reduced test suite.

	r_1	r_2	r_3	r_4	r_5	r_6
t_1	X	X	X			
t_2	X			X		
t_3		X			X	
t_4			X			X
t_5					X	

Figure 3.2: Testing requirements covered by each test case of a test suite [2]

In the Figure 3.2, we can see that when we apply greedy heuristic algorithm, it will select the t_1, t_2, t_3, t_4. By observing we can note that t_1 is redundant test case because it is covering the requirement r_1 which is also covered by t_2. This happens because greedy heuristic algorithm made early selection. To resolve this problem, the authors presented Delayed-Greedy algorithm relied on Concept Analysis Framework. In this framework, the objects represent the test cases whereas attributes represents the testing requirements. The coverage information of testing requirements exercised by test cases is considered as a relation between objects and attributes. The goal is to identify maximal grouping of test cases and testing requirements. This grouping is known as concepts. A Concept can be defined as an ordered pair (X, Y) where X equals to subset of test cases and Y equals to subset of testing requirements. This pair should satisfies the rule that Y is the maximal set of attributes that are related to all the objects in X and X is the maximal set of objects that are related to all the attributes in Y. They have tested their algorithm on the Siemen suite and space program and identified that algorithm always produced the smaller size or same size test suite as compare to prior heuristics. Moreover, time performance of delayed greedy algorithm is comparable to classical greedy heuristic algorithm.

Model-Based Test Suite Reduction

Model-based testing generates test suite for a given system automatically (Emilia et al.) [6]. This reduces the effort required to create test cases but on the other hand it also comes with a problem of resource constraints such as executing high number of automatically generated test cases. To fix this problem, there are a couple of test suite reduction methods proposed and tested in the past, however most of those are applied on code based test cases and not specification based test cases. The code based test cases are those which are built by developer and written in the form of code in some framework whereas specification based test cases are those which are generated automatically from specification of system behavior under model based testing. In this paper, the authors have suggested a new technique to reduce test suite's size generated from model-based testing. The method identifies the similar test cases based on similarity matrix (generated using distance measuring between two test cases) and creates a reduced test suite that contains dissimilar test cases which can test most of the testing requirements. The results showed that the test suite's size has been reduced significantly while keeping the fault detection capability equal to original test suite's fault detection capability.

3.2.2 Inadequate Test Suite Reduction

In this approach, the size of test suite is reduced by removing redundant or unnecessary test cases while allowing loss in fault detection capability up to some inadequacy level.

FAST-R

Cruciani et al. have proposed similarity-based techniques borrowed from big data domain for test suite size reduction [14]. They proposed the FAST-R algorithm family which includes FAST++ and FAST-CS based on K-means++ and coresets respectively to perform test suite size reduction. The difference between K-means++ and standard K-means is the initialization of centroids. The standard kmeans can select the centroids which are far-off points and as a result we could have centroid with no cluster or more than one cluster (poor clustering). To overcome this problem, K-means++ is proposed; first pick the centroid point randomly from data points, then calculate the distance between each point and previously selected centroid point, the point having maximum distance from centroid will be selected next. Repeat the step until k centroids have been selected. Once centroids are selected, then apply the standard kmeans algorithm. After K-means++, the next interesting approach is coresets in which the aim is to find small subset of original points which approximates the geometric features of original points. The FAST approaches used locality-sensitive hashing and minhashing algorithms. They have increase the scalability of these algorithms by using the random projection technique to reduce the dimensions of space while keeping the pairwise distances of the points. For test cases prioritization, they used the FAST-all and FAST-pw to have the optimal test suite. The authors have tested the FAST-R algorithms on SIR and Defect4J benchmark programs. For evaluation purposes, they used fault detection loss metric to measure the quality of the reduced test suite. The formula to calculate fault detection loss is:

$$FDL = \frac{|F| - |F'|}{|F|} \times 100$$

The F represents the fault detection capability of original test suite whereas F' represents the fault detection capability of reduced test suite. The authors have applied the algorithms on more than five hundred thousand test cases and got promising results.

In the Figure 3.3, we can see that fault detection loss of state-of-the-art techniques is comparable to Fast-R family algorithms. The only difference between these techniques is reduction time, the Fast-R has algorithm (Fast-CS) which is reducing the test suite in very less time (10 seconds) as compared to traditional approaches which takes many hours.

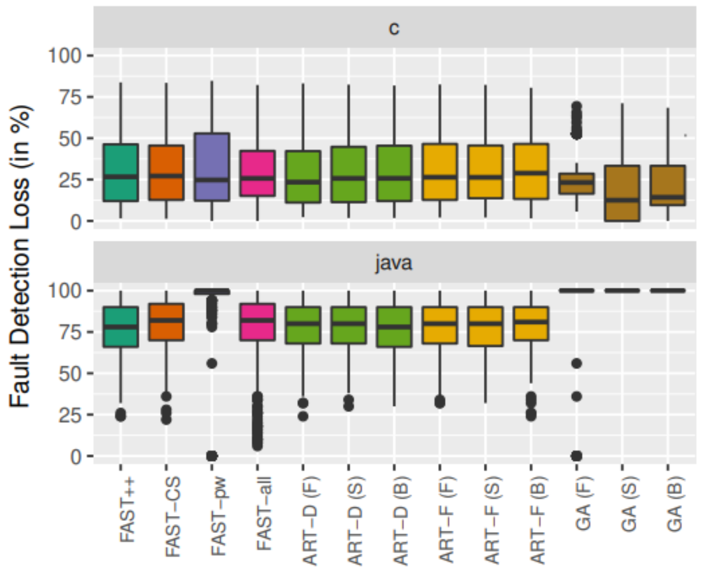
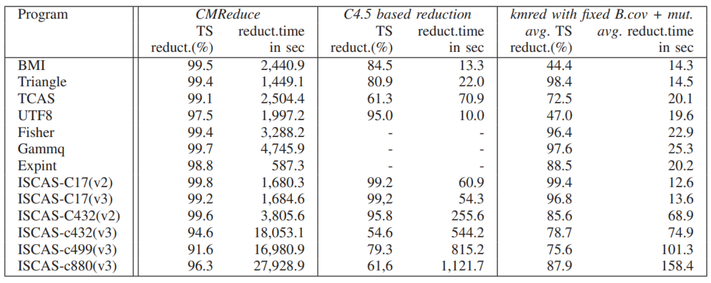


Figure 3.3: Fault Detection Loss for the test suite reduction approaches [14]

K-means Clustering

Chetouane et al. have proposed the machine learning algorithm known as K-means clustering to reduce the test suite [15]. The algorithm creates the clusters for related test cases and then adds representative test cases in the reduced test suite. The binary search is used for selecting the appropriate number of clusters that will not have much effect on the coverage or mutation score. The authors have tested the algorithm for 13 Java programs. The results showed that the test suite has been reduced on average by 82.5% by maintaining the original code coverage and mutation score. We can see the results in Figure 3.4. K-means reduction (kmred) is compared with Coverage and Mutation score based reduction (CM Reduce) and C4.5 based reduction. The coverage metrics used in the algorithms are fixed branch coverage and mutation score.

A. Pandey, A. K. Malviya et al. reduced the test suite's size with the help of K-means



Program	*CMReduce*		*C4.5 based reduction*		*kmred with fixed B.cov + mut.*	
	TS reduct.(%)	reduct.time in sec	TS reduct.(%)	reduct.time in sec	*avg.* TS reduct.(%)	*avg.* reduct.time in sec
BMI	99.5	2,440.9	84.5	13.3	44.4	14.3
Triangle	99.4	1,449.1	80.9	22.0	98.4	14.5
TCAS	99.1	2,504.4	61.3	70.9	72.5	20.1
UTF8	97.5	1,997.2	95.0	10.0	47.0	19.6
Fisher	99.4	3,288.2	-	-	96.4	22.9
Gammq	99.7	4,745.9	-	-	97.6	25.3
Expint	98.8	587.3	-	-	88.5	20.2
ISCAS-C17(v2)	99.8	1,680.3	99.2	60.9	99.4	12.6
ISCAS-C17(v3)	99.2	1,684.6	99,2	54.3	96.8	13.6
ISCAS-C432(v2)	99.6	3,805.6	95.8	255.6	85.6	68.9
ISCAS-c432(v3)	94.6	18,053.1	54.6	544.2	78.7	74.9
ISCAS-c499(v3)	91.6	16,980.9	79.3	815.2	75.6	101.3
ISCAS-c880(v3)	96.3	27,928.9	61,6	1,121.7	87.9	158.4

Figure 3.4: Comparison of different test suite reduction approaches using branch coverage and mutation score [15]

algorithm and elbow method [12]. The elbow method helps in determining the value of k or the number of clusters for a given data set. In elbow method, we run the K-means algorithm for different values of k on the data set, after that sum of squared errors is calculated for each value of k and plotted. If the plot forms the arm like line, then the value on elbow of line arm is the best value for k.

The Figure 3.5 shows the proposed methodology: first the source program and its test cases are selected. Second, the dataset or vectors are generated from the test cases. Third, the K-means algorithm is applied on the dataset and the elbow method is used to get an accurate value of k. At the last, the redundant test cases are removed by applying specific filters. They performed the experiment on a Java Program called Triangle and used the WEKA library for K-means clustering. There are three inputs and one output of the program.

In Figure 3.6, we can observe the notable reduction in the number of test cases. The algorithm intelligently removed the redundant or similar test cases from the triangle test suite.

3.2.3 Comparison of Adequate and Inadequate Approaches

C. Coviello, S. Romano and G. Scanniello et al. have compared the adequate and inadequate methods for test suite reduction. The algorithms under adequate and inadequate categories include "Harrold-Gupta-Soffa, Greedy, Delayed Greedy, 2-Optimal, Greedy essential (GE), and Greedy Redundant Essential (GRE)". For comparison, they an-

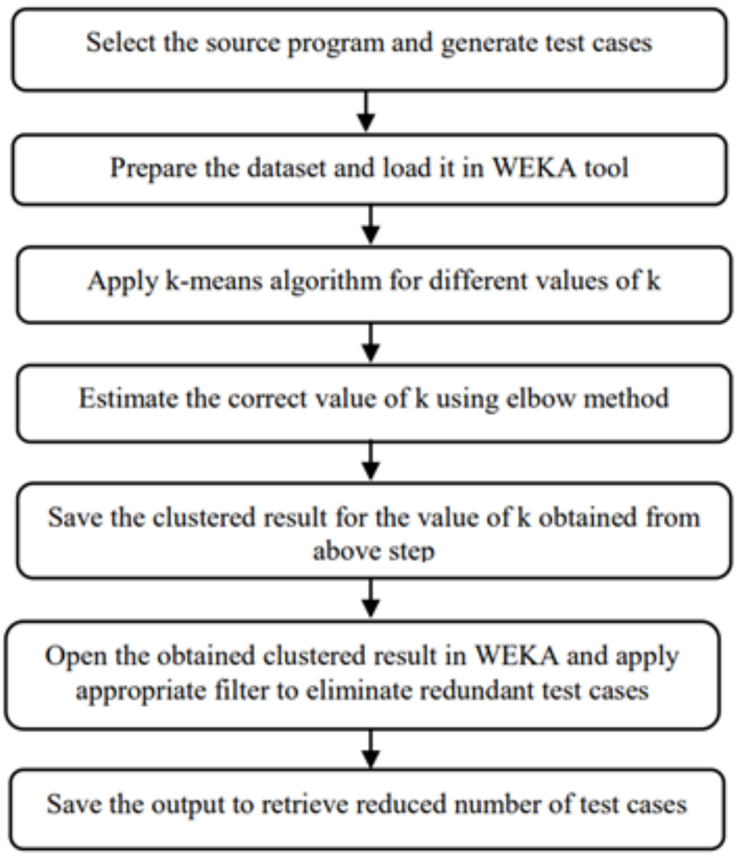
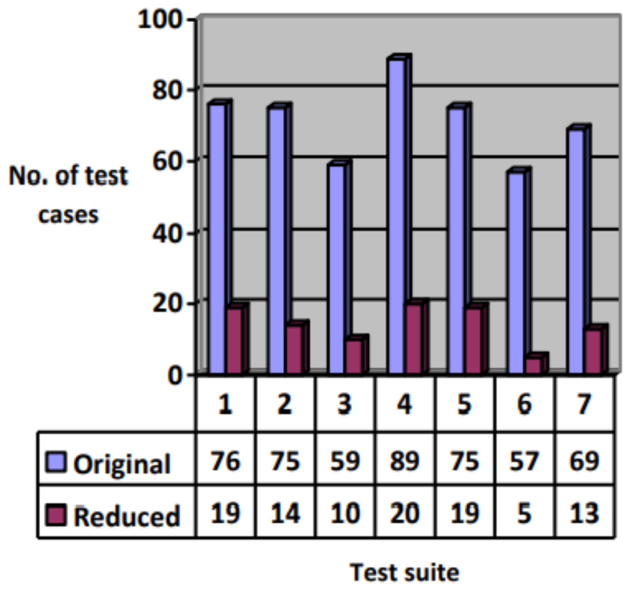

Select the source program and generate test cases
Prepare the dataset and load it in WEKA tool
Apply k-means algorithm for different values of k
Estimate the correct value of k using elbow method
Save the clustered result for the value of k obtained from above step
Open the obtained clustered result in WEKA and apply appropriate filter to eliminate redundant test cases
Save the output to retrieve reduced number of test cases

Figure 3.5: Proposed Methodology of K-means clustering test suite reduction using elbow method [12]



	1	2	3	4	5	6	7
🟦 Original	76	75	59	89	75	57	69
🟥 Reduced	19	14	10	20	19	5	13

Figure 3.6: Graph showing original count of test cases vs reduced count of test cases [12]

alyzed two metrics: reduced test suite's size and fault detection capability loss. For clustering-based algorithm, the authors used various number of distance and similarity measures such as Euclidean Distance, Cosine (Dis)similarity, Jaccard-Based Dissimilarity, Hamming Distance, Levenshtein Edit Distance, K-Based Dissimilarity, and String Kernels-Based Dissimilarity.

Euclidean Distance: It is the distance between two vectors, let a and b be two vectors, then Euclidean distance of these two vectors is defined as:

$$d(a, b) = \sqrt{(b - a)^2}$$

Cosine (Dis)similarity: It can be calculated between two vectors a and b as follows:

$$d(a, b) = 1 - \frac{a.b}{|a||b|}$$

Jaccard-Based Dissimilarity: "Let X and Y be two sets, then Jaccard-based dissimilarity between them is equal to one minus the Jaccard co-efficient":

$$d(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|}$$

Hamming Distance: "Let x and y be two vectors of the same length, the hamming distance between them is the number of elements in which they differ. For example, given x = (1,1,0,1) and y = (0,1,0,0) their hamming distance is two."

Levenshtein Edit Distance: "Let a and b be two sequence of characters (two strings), the Levenshtein edit distance between them is the minimum number of operations required to convert a into b. The operations are as follows: add new character, delete a character, replace one character with another."

K-Based Dissimilarity: "The K-based dissimilarity, namely Cohen's Kappa index, measures the degree of agreement between two raters, which classifies items over two or more categories. Let p_o be the observed proportion of agreement and p_e the agreement expected just by chance, the K-based dissimilarity is defined as follows:"

$$k = 1 - \frac{p_0 - p_e}{1 - p_e}$$

String Kernels-Based Dissimilarity: "The kernel is a function that computes the inner product of two vectors. A string kernel measures the similarity between two strings. Let a and b be two strings, the kernel function is given as:"

$$k(a, b) = \sum_{s \in A^+} num_s(a) num_s(b) \lambda_s$$

Among all of the aforementioned distance methods, the results showed that test suite reduction with clustering method performs well with Euclidean Distance, Hamming Distance, and Levenshtein Edit Distance. Moreover, the comparison results showed that inadequate approaches, especially the clustering-based approaches outperform others with respect to test suite size reduction and keeping the fault detection capability loss to a minimal level. We can see the test suite size reduction for both adequate and inadequate approaches in Figures 3.7 and 3.8.

The authors have discussed the difference between adequate and inadequate test suite reduction approaches and developed CUTER (Eclipse Plug-in) [10]. This is based on an inadequate approach and uses clustering to group the similar test case. They have identified that test cases are similar if they are covering the same statements of code. They used the Hierarchical Agglomerative Clustering (HAC) to cluster the same test cases. Once clusters are formed, CUTER creates a reduced test suite by selecting the representative from each cluster. The representative is selected based on number of code coverage. Test case that covers higher code will have high fault detection capability. The

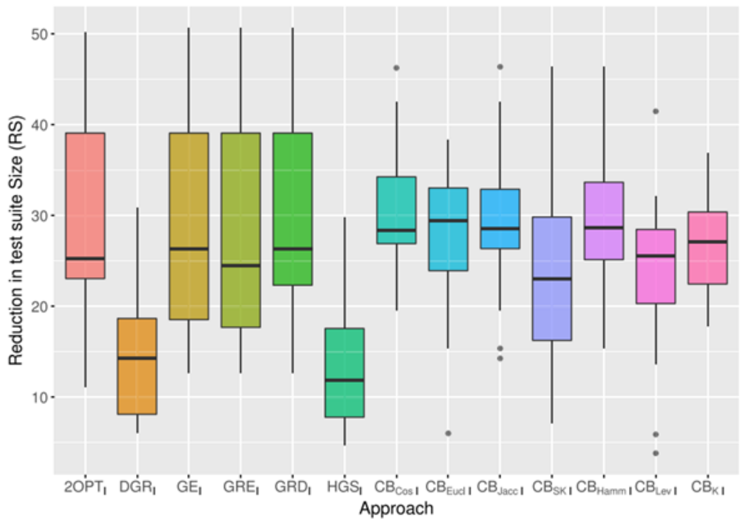


Figure 3.7: Boxplots of test suite size reduction for each inadequate TSR approach (inadequacy level = 95%) [16]

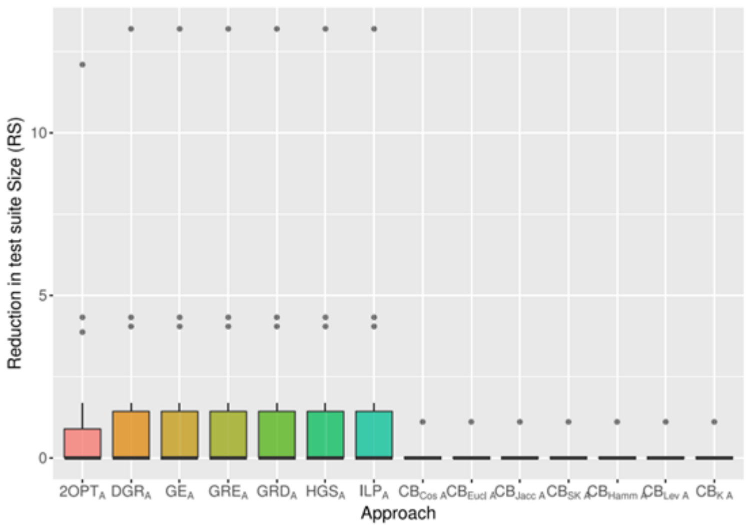


Figure 3.8: Boxplots of test suite size reduction for each adequate TSR approach [10]

authors have tested their tool on 19 versions of four Java programs. The results showed that the effectiveness of detecting faults in less time has been improved. This is to note here that all these approaches make use of C and Java programs as test data. So far, we have seen the test suite reduction approaches. As we mentioned in the start of this chapter that there is another category where we can reduce the cost of regression testing and it is known as Test Case Selection.

3.3 Test Case Selection

"Given: The program, P, the modified version of P, P' and a test suite, T. Problem: Find a subset of T, T', with which to test P." [5]

Test case selection is also used to save time consumed on regression testing. However, test case selection is limited to modified program, we only perform test case selection on modified version of program. In this approach, we try to find those test cases which are more related to modified part of the program and execute only selected test cases on modified version of program, thus the overall execution time of regression testing is reduced.

3.4 Test Case Prioritization

"Given: A test suite, T, the set of permutations of T, PT, and a function from PT to real numbers, f: PT →R. Problem: To find $T' \in PT such that (\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$." [5]

In the end, we found rearranged test suite T' that detects the maximum faults in the earlier stage. The most common test case prioritization is code coverage based where those test cases are executed first which covers the maximum number of lines of code. However, this rearrangement cannot be beneficial every time in detecting faults. To overcome this problem, Chunrong Fang et al. proposed similarity-based test case prioritization using relative execution frequency of program entities which executes the test case in specific order to identify faults early [7]. They performed the experiment on five open-source Java programs and observed that their technique has increased the fault detection rate significantly.

Chen et al. proposed Adaptive Random Sequence approach on clustering algorithms such as K-means and K-mediods to improve the effectiveness and earlier detection of faults in regression testing [9]. The clustering algorithm is applied on test cases to group them into clusters. Test cases having the same properties will be grouped together and they can detect the same type of faults. Once clusters are created, the adaptive random sequences are generated from developed clusters. These sequences are used to prioritize test cases. The authors have tested their proposed algorithm on seven open source programs. The results showed that the fault detection capability has been improved in less time.

When manual test steps are translated into executable code, it is known as Test Case Automation [11]. Many researchers have identified similarities in these kinds of automatically generated test steps scripts. In some test cases, there are chances that the same test steps repeat, so Flemstrom et al. proposed that instead of regenerating the automated script again for that test step, we should reuse the existing script which does the same job. Moreover, their solution finds an optimal order of execution of test steps in such a way that similar scripts do not execute again. They have tested their method on four projects having 3919 integration test cases with 35180 test steps. The results showed that their prioritization method reduced the time of integration testing effectively.

Chapter 4

Methodology

This chapter describes the methodologies and tools used in the research.

4.1 Proposed Methodology

A variant of K-means suggested by Chetuone et. al is implemented in JavaScript to reduce the execution cost of node package's test suite [15]. It works as illustrated in the Figure 4.1:

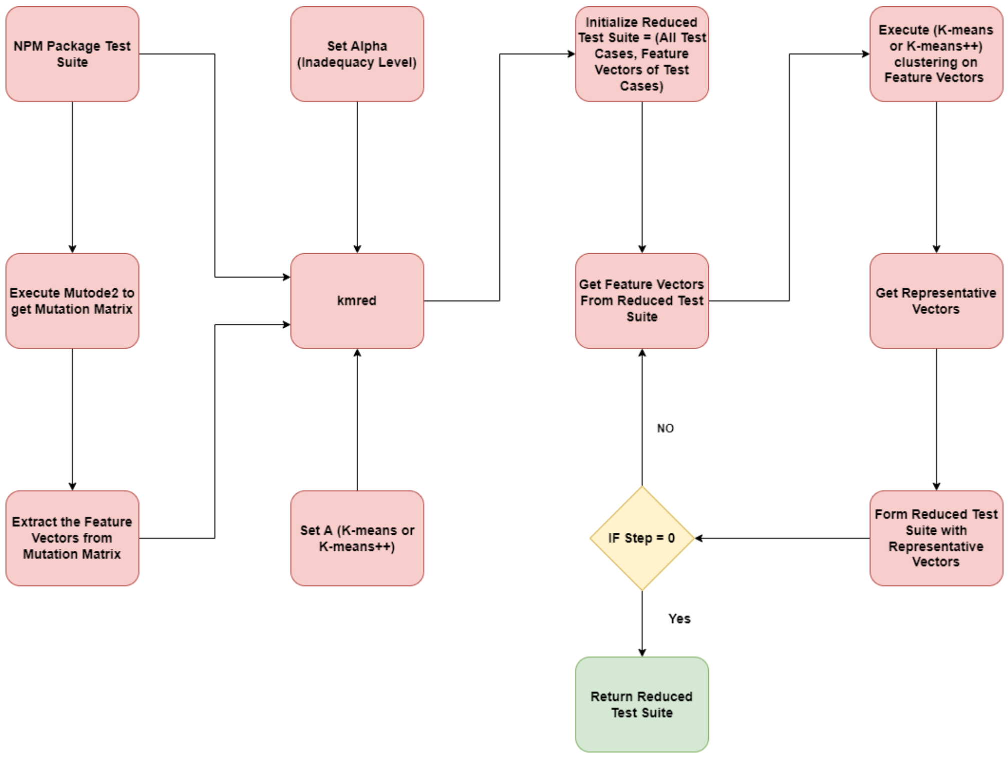

NPM Package Test Suite
Set Alpha (Inadequacy Level)
Initialize Reduced Test Suite = (All Test Cases, Feature Vectors of Test Cases)
Execute (K-means or K-means++) clustering on Feature Vectors
Execute Mutode2 to get Mutation Matrix
kmred
Get Feature Vectors From Reduced Test Suite
Get Representative Vectors
NO
Extract the Feature Vectors from Mutation Matrix
Set A (K-means or K-means++)
IF Step = 0
Form Reduced Test Suite with Representative Vectors
Yes
Return Reduced Test Suite

Figure 4.1: Flow Chart of the Proposed Algorithm

1. Run the Mutode2 (extended version of Mutode) on NPM package. It will return a mutation matrix. The mutation matrix tells that how many numbers of mutants are killed by each test case.

2. Run the text parser to get feature vectors from mutation matrix. The parser will return the 2d vector [X, Y] for each test case where X represents the total count of killed mutants and Y represents the total count of non-killed mutants for test cases. The vectors will be returned in JSON format. Each vector contributes to overall mutation score, the X having higher value and Y having lower value means that the vector contributes more to mutation score as compared to the one which has lower value of X and higher value of Y.

3. The feature vectors of each test case are passed to K-means or K-means++ clustering algorithm along with alpha (Inadequacy Level).

   a. The K-means algorithm performs clustering by selecting the initial centroids randomly. The distance between centroids and data points are computed. Once distance is measured, data points are allocated to nearest centroids (clusters). In next iteration, the new centroids are calculated by taking the average of all data points belonging to a centroid. The algorithm repeats until there is no change in centroids.

b. The K-means++ algorithm performs clustering in the same way as done by K-means, the only difference between K-means and K-means++ is initialization of centroids. In K-means++, initially the centroid points are picked randomly from data points. Afterwards, calculate the distance between each point and previously selected centroid point, the point having maximum distance from centroid will be selected next. Repeat the step until k centroids have been selected. After centroid selection, the standard K-means algorithm is applied.

4. The representative vector against test case is selected from each cluster. The vector point that is nearest to the centroid of the cluster has been taken as representative.

5. The new test suite is formed by combining all the representative test cases. We call this new test suite a reduced test suite.

6. Coverage of the reduced test suite is calculated and compared with the original test suite's coverage.

7. The steps from 2-5 will be executed until the step size goes to zero. In the end, algorithm will return the reduced test suite having fault detection loss within inadequacy range. If the algorithm is unable to found the reduced test suite, then it will return zero reduction.

4.2 Pseudocode of K-means Test Suite Reduction

Preprocessing: Run Mutode2 on package to extract the mutation matrix

```
1: procedure kmred(A package Π, Mutation Matrix M, $\alpha$ is maximum allowed deviation between mutation score of original TS and mutation score of obtained TS, ‘A’ is an algorithm used for clustering)
2:     Execute text-parser on M to get the feature vectors ($f_v$) for each test case.
3:     Let $cov_B$ be max((coverage(M) − $\alpha$), 0).
4:     Let step be $\lceil \frac{\text{number of test cases}}{2} \rceil$.
5:     Let k be step.
6:     Initialize reduced test suite ($TS_{red}$ = (Test Case, $f_v$)). It is a map containing key as test case and value as feature vector against that test case. Initialize ($TS_{red}$) with all test cases along with their feature vectors.
7:     repeat
8:         Get feature vectors from ($TS_{red}$) and store it in $f_v$
9:         if A = ‘kmpp’ then
10:            Call K-means++ ($f_v$, k) to obtain set of clusters (C).
11:        else
12:            Call K-means ($f_v$, k) to obtain set of clusters (C).
13:        end if
14:        Let TS = set of test cases retrieved by choosing a representative vector from the cluster C.
15:        Let cov = coverage (TS).
16:        if step = 1 then
17:            Let step = 0.
18:        else
19:            Let step = $\lceil \frac{step}{2} \rceil$
20:        end if
21:        if cov < $cov_B$ then
22:            Let k = k + step.
23:        else
24:            Let $TS_{red}$ = TS.
25:            Let k = k − step.
26:        end if
27:        if k > number of test cases then
28:            Let step = 0.
29:        end if
30:    until step = 0
31:    return $TS_{red}$
32: end procedure
```

Figure 4.2: kmred – K-means Clustering For Test Suite Reduction [15]

In the above algorithm, we need mutation matrix of package. To do this, we must do some preprocessing step; execute the Mutode2 on package to get the mutation matrix. The mutation matrix file contains information of killed/non-killed mutants against each test case. The example of mutation matrix can be seen in table 4.1 where 0 represents the killed mutant and 1,-2 represents the non-killed mutants. 1 means that mutant is survived and -2 means that test cases are taking time to run over mutant, therefore, resulting into timeout.

Mutant Id	Time to Execute	0-nodeRNG	1-mathRNG	2-cryptoRNG	3-sha1 node	4-sha1 browser	5-md5 node
MUTANT 3	1196ms	0	0	0	0	0	0
MUTANT 1	1221ms	0	0	0	1	0	1
MUTANT 4	1323ms	0	1	0	0	0	0
MUTANT 6	1281ms	0	0	1	0	0	0
MUTANT 5	1308ms	0	0	0	0	0	0
MUTANT 7	1381ms	0	0	0	0	1	0
MUTANT 9	1103ms	1	1	1	1	1	1
MUTANT 8	1174ms	1	1	1	1	1	1
MUTANT 10	1145ms	1	1	1	1	1	1
MUTANT 12	1151ms	1	1	0	1	0	1
MUTANT 11	1261ms	1	1	1	1	1	1
MUTANT 13	1257ms	1	0	0	1	1	1
MUTANT 14	1175ms	1	1	1	1	1	1
MUTANT 18	1207ms	-2	-2	-2	-2	-2	-2
MUTANT 15	1176ms	1	1	1	1	1	1
MUTANT 16	1229ms	1	1	1	1	1	1
MUTANT 19	1172ms	-2	-2	-2	-2	-2	-2
MUTANT 20	1371ms	1	1	1	1	0	1

Table 4.1: Mutation Matrix for some test cases of uuid-3.4.0 package

Once we have mutation matrix, we run our text parser which reads the mutation matrix and form feature vector [X, Y] where X represent the total count of killed mutants and Y represents the total count of non-killed mutants. There will be one vector for each test case. The example can be viewed in table 4.2.

Test Case	Feature Vector (X,Y)
0-nodeRNG	(6,10)
1-mathRNG	(6,10)
2-cryptoRNG	(7,9)
3-sha1 node	(5,11)
4-sha1 browser	(7,9)
5-md5 node	(5,9)

Table 4.2: Feature vectors extracted for some test cases of uuid-3.4.0 package

Once vectors are formed, we passed these to K-means or K-means++ algorithm. The algorithm clusters similar vectors and provide it to kmred algorithm. The kmred algorithm selects a representative vector from each cluster. The vector which is nearest to the centroid of the cluster has been selected as a representative vector. We have used Euclidean distance to compute the nearest point to centroid. After that, we can get the actual test cases from these representative vectors and form a reduced test suite. We will keep reducing the test suite until step size goes to zero. Further, we will apply coverage criteria like mutation score to ensure that the reduced test suite has minimal loss in failure detection capability as compared to original test suite. This way, we aim to reduce the regression testing costs.

Chapter 5

RESULTS & DISCUSSION

This chapter discusses the results of the K-means and K-means++ clustering algorithms.

5.1 Test Data Details

The suggested algorithms are tested against 43 package versions of seven NPM packages. The name of packages are as follows: async, body parser, cheerio, express, passport, shortid, and uuid. There is short description provided for each package in the Table 5.1. The package versions for each package can be viewed in Table 5.2. Out of these, 20 package versions gave test suite reduction for maximum number of alpha values. These 20 package versions can be viewed in the Table 5.3. However, we only discuss the results of those package versions which gives test suite reduction for all alpha values. This section also highlights the limitations of implemented algorithms.

To perform experiments on NPM packages, we have implemented the kmred in JavaScript language. We have used tf-kmeans and skmeans node packages. The first package uses K-means to cluster the vectors based on the Euclidean distance whereas the second package uses K-means++ for clustering. These packages also have methods to do synchronous or asynchronous clustering. Apart from that, to check the quality of the test suite, we used a mutation score as a quality measuring metric. For generating mutants, we used Mutode2: a JavaScript mutation tool. This tool generates the mutation score and a csv file containing all the information of mutants (Survived or Killed). We have used this file to generate the mutation score for reduced test cases instead of calling again the Mutode2 because it's execution time is quite long for larger test suites. We

Name	**Description**
aysnc	• It is utility module which provide functions to do asynchronous JavaScript programming.
cheerio	• Parser tool used to parse any HTML and XML structure • Use to build web scraping bots
express	• It is used to build HTTP public APIs, single page apps, and hybrid apps • It reduces the time of web application development through built-in modules and features
passport	• Passport is used to implement authentication systems in Express-based web applications with only fews commands
shortid	• shortid package is used whenever we want to generate some unique ids which are url friendly
uuid	• It is also used to generate unique ids
body-parser	• It is mostly used with the express package to parse the body of web requests before it is actually processed by the handler method

Table 5.1: Node Package Versions and their Usage

performed the experiment using a Dell Inspiron 15 3000 laptop having a core i3 with processing speed of 2.6 GHz. We have executed each algorithm (K-means and K-means++) ten times on each package version.

5.2 Results

The experiment showed notable reduction in the test suite's size while having minimal effect on the fault detection capability. The results can be observed from Tables 5.4, 5.5, 5.6, 5.7 and Figures 5.1, 5.2, 5.3, 5.4.

In the table 5.4, $|T_{avg}|$ represents average number of test cases in original test suite of all the node package versions. For example, we have tested against 9 versions of uuid,

NPM Package	**Versions**
async	async-2.4.0,async-2.5.0,async-2.6.0,async-2.6.1
body parser	body-parser-1.16.0,body-parser-1.16.1,body-parser-1.17.0,body-parser-1.18.0,body-parser-1.18.2
cheerio	cheerio-1.0.0
express	express-4.15.0,express-4.16.0,express-4.16.1,express-4.16.2,express-4.16.3
passport	passport-0.2.0,passport-0.2.1,passport-0.2.2,passport-0.3.0,passport-0.3.1,passport-0.3.2,passport-0.4.0,passport-0.4.1,passport-0.5.0
shortid	shortid-2.2.7,shortid-2.2.8,shortid-2.2.9,shortid-2.2.10,shortid-2.2.11,shortid-2.2.12,shortid-2.2.13,shortid-2.2.14,shortid-2.2.15,shortid-2.2.16
uuid	uuid-3.0.1,uuid-3.1.0,uuid-3.2.0,uuid-3.2.1,uuid-3.3.0,uuid-3.3.1,uuid-3.3.2,uuid-3.3.3,uuid-3.4.0

Table 5.2: NPM Package Versions

NPM Package	**Versions**
async	async-2.4.0
body parser	body-parser-1.16.1,body-parser-1.17.0,body-parser-1.18.0
cheerio	cheerio-1.0.0
express	express-4.16.0,express-4.16.1,express-4.16.2
passport	passport-0.2.0,passport-0.2.1,passport-0.2.2, passport-0.3.0, passport-0.5.0
shortid	shortid-2.2.12,shortid-2.2.14,shortid-2.2.15
uuid	uuid-3.2.0,uuid-3.2.1,uuid-3.3.0,uuid-3.3.3

Table 5.3: NPM Package Versions Gave Test Suite Reduction For Maximum Number of Alpha Values

so $|T_{avg}|$ will be average number of test cases in original test suite of all uuid package versions. In addition to that, α shows inadequacy level, $|T'_{avg}|$ is equivalent to average number of test cases in reduced test suite of all package versions of some particular node package. $|F_{avg}|$ represents average mutation score of original test suite of all package versions belonging to some particular node package, $|F'_{avg}|$ represents average mutation score of reduced test suite of all package versions related to some specific node package.

In table 5.4, the row having test suite reduction for body parser against alpha value of 5 is calculated as follows: First the K-means is executed ten times for each body parser package version "body-parser-1.16.0, body-parser-1.16.1, body-parser-1.17.0, body-parser-1.18.0, body-parser-1.18.2". After that, the average of test suite reduction and average of fault detection loss is calculated against ten executions of each example version. This is reported in sheet named 'K-means Results Averages' which has been mentioned in appendix A. Note that, we have removed "body-parser-1.16.0, body-parser-1.18.2" because those were not giving test suite reduction for all alpha values (5, 10, 15, 20, 25, 30, 35, 40, 45, 50). In the last, we compute average of $|T'_{avg}|$, in this case, average of "body-parser-1.16.1, body-parser-1.17.0, body-parser-1.18.0" and report it in the table 5.4 as our final result. So in the last, we can see that body parser has test suite reduction of 77.1% against alpha value of 5 for versions "body-parser-1.16.1, body-parser-1.17.0, body-parser-1.18.0". We have computed the results for K-means++ in the same way. For K-means++, we have sheets named 'K-means++ Results' and 'K-means++ Results Averages' in appendix B. This can be observed that by increasing alpha value, the test suite reduction also increased.

			Test Suite Reduction			Fault Detection Loss	
Name	$\|T_{avg}\|$	α	$\|T'_{avg}\|$	$\frac{\|T_{avg}\|-\|T'_{avg}\|}{\|T_{avg}\|} \times 100$	$\|F_{avg}\|$	$\|F'_{avg}\|$	$\frac{\|F_{avg}\|-\|F'_{avg}\|}{\|F_{avg}\|} \times 100$
async	477.0	5	125.8	73.63	39.18	37.47	4.36
async	477.0	10	113.7	76.16	39.18	36.87	5.90
async	477.0	15	111.4	76.65	39.18	36.60	6.59
async	477.0	20	109.8	76.98	39.18	36.87	5.90
async	477.0	25	110.6	76.81	39.18	36.77	6.15
async	477.0	30	109.4	77.06	39.18	36.54	6.74
async	477.0	35	112.6	76.39	39.18	35.84	8.52
async	477.0	40	104.7	78.05	39.18	30.52	22.10
async	477.0	45	105.2	77.95	39.18	33.02	15.73
async	477.0	50	102.2	78.57	39.18	27.65	29.42
body parser	209.7	5	77.1	63.21	82.99	81.64	1.63
body parser	209.7	10	74.1	64.65	82.99	78.07	5.92
body parser	209.7	15	71.1	60.23	82.99	76.31	8.70
body parser	209.7	20	68.8	67.17	82.99	74.36	10.39
body parser	209.7	25	69.6	66.78	82.99	73.50	11.43
body parser	209.7	30	67.4	67.83	82.99	71.40	13.96
body parser	209.7	35	64.8	69.10	82.99	67.11	19.14
body parser	209.7	40	56.2	73.17	82.99	63.38	23.63
body parser	209.7	45	49.8	76.26	82.99	55.01	33.72
body parser	209.7	50	50.0	76.14	82.99	54.76	34.01
cheerio	650.0	5	0.0	0.00	83.54	0	0.00
cheerio	650.0	10	0.0	0.00	83.54	0	0.00
cheerio	650.0	15	0.0	0.00	83.54	0	0.00
cheerio	650.0	20	0.0	0.00	83.54	0	0.00
cheerio	650.0	25	186.0	71.38	83.54	64.28	23.05
cheerio	650.0	30	161.0	75.23	83.54	60.7	27.34
cheerio	650.0	35	153.6	76.37	83.54	59.19	29.15
cheerio	650.0	40	155.2	76.12	83.54	60.04	28.13
cheerio	650.0	45	153.4	76.40	83.54	59.02	29.35
cheerio	650.0	50	151.6	76.68	83.54	58.87	29.53
express	859.3	5	277.4	67.72	68.75	66.94	2.63
express	859.3	10	270.9	68.48	68.75	64.59	6.05
express	859.3	15	262.0	69.51	68.75	62.29	9.41
express	859.3	20	251.9	70.68	68.75	59.59	13.33
express	859.3	25	247.8	71.17	68.75	58.20	15.35
express	859.3	30	242.7	71.76	68.75	56.40	17.97
express	859.3	35	233.1	72.88	68.75	54.43	20.83
express	859.3	40	226.0	73.70	68.75	51.98	24.40
express	859.3	45	203.5	76.32	68.75	45.56	33.74
express	859.3	50	197.5	77.02	68.75	42.64	37.98
passport	489.6	5	0.0	0.00	82.61	0.00	0.00
passport	489.6	10	0.0	0.00	82.61	0.00	0.00
passport	489.6	15	0.0	0.00	82.61	0.00	0.00
passport	489.6	20	0.0	0.00	82.61	0.00	0.00
passport	489.6	25	0.0	0.00	82.61	0.00	0.00
passport	489.6	30	0.0	0.00	82.61	0.00	0.00
passport	489.6	35	0.0	0.00	82.61	0.00	0.00
passport	489.6	40	64.7	86.79	82.61	50.02	39.45
passport	489.6	45	57.4	88.28	82.61	48.87	40.84
passport	489.6	50	57.1	88.34	82.61	48.77	40.97

Table 5.4: Test Suite Reduction Using K-means

			Test Suite Reduction			Fault Detection Loss	
Name	$\|T_{avg}\|$	α	$\|T'_{avg}\|$	$\frac{\|T_{avg}\|-\|T'_{avg}\|}{\|T_{avg}\|} \times 100$	$\|F_{avg}\|$	$\|F'_{avg}\|$	$\frac{\|F_{avg}\|-\|F'_{avg}\|}{\|F_{avg}\|} \times 100$
shortid	17.0	5	9.8	42.65	48.05	47.57	1.01
shortid	17.0	10	9.4	44.61	48.05	46.62	2.97
shortid	17.0	15	6.4	62.09	48.05	44.05	8.32
shortid	17.0	20	4.8	71.57	48.05	42.43	11.70
shortid	17.0	25	6.7	60.78	48.05	43.87	8.71
shortid	17.0	30	4.7	72.16	48.05	38.04	20.83
shortid	17.0	35	4.7	72.55	48.05	35.36	26.41
shortid	17.0	40	4.0	76.47	48.05	33.76	29.73
shortid	17.0	45	4.0	76.47	48.05	33.76	29.73
shortid	17.0	50	2.0	88.24	48.05	25.19	47.57
uuid	16.3	5	8.8	45.83	91.52	90.71	0.89
uuid	16.3	10	8.6	46.79	91.52	90.66	0.94
uuid	16.3	15	7.7	52.89	91.52	89.58	2.13
uuid	16.3	20	7.5	53.65	91.52	89.01	2.74
uuid	16.3	25	8.2	49.32	91.52	88.99	2.77
uuid	16.3	30	7.8	52.05	91.52	88.43	3.38
uuid	16.3	35	8.1	50.32	91.52	89.15	2.60
uuid	16.3	40	7.7	52.38	91.52	89.09	2.66
uuid	16.3	45	8.8	46.08	91.52	88.95	2.82
uuid	16.3	50	7.9	51.62	91.52	89.06	2.69

Table 5.5: Test Suite Reduction Using K-means - Continued

			Test Suite Reduction			Fault Detection Loss	
Name	$\lvert T_{avg}\rvert$	α	$\lvert T'_{avg}\rvert$	$\frac{\lvert T_{avg}\rvert-\lvert T'_{avg}\rvert}{\lvert T_{avg}\rvert}\times 100$	$\lvert F_{avg}\rvert$	$\lvert F'_{avg}\rvert$	$\frac{\lvert F_{avg}\rvert-\lvert F'_{avg}\rvert}{\lvert F_{avg}\rvert}\times 100$
async	477.0	5	127.0	73.38	39.18	37.57	4.11
async	477.0	10	111.6	76.60	39.18	36.74	6.23
async	477.0	15	112.4	76.44	39.18	36.53	6.76
async	477.0	20	103.8	78.24	39.18	36.73	6.25
async	477.0	25	112.0	76.52	39.18	36.67	6.42
async	477.0	30	100.5	78.93	39.18	36.10	7.86
async	477.0	35	110.2	76.90	39.18	34.99	10.69
async	477.0	40	105.8	77.83	39.18	33.46	14.61
async	477.0	45	104.5	78.09	39.18	30.48	22.21
async	477.0	50	102.8	78.45	39.18	26.51	32.35
body parser	209.7	5	76.8	63.35	82.99	81.72	1.53
body parser	209.7	10	73.3	65.05	82.99	77.24	6.93
body parser	209.7	15	70.5	66.37	82.99	75.68	8.81
body parser	209.7	20	67.4	67.87	82.99	72.85	12.21
body parser	209.7	25	70.7	66.27	82.99	74.56	10.16
body parser	209.7	30	68.4	67.38	82.99	70.31	15.28
body parser	209.7	35	66.7	68.20	82.99	67.31	18.89
body parser	209.7	40	57.1	72.77	82.99	63.83	23.09
body parser	209.7	45	32.4	84.53	82.99	45.87	44.73
body parser	209.7	50	42.3	79.81	82.99	54.40	34.45
cheerio	650.0	5	0.0	0.00	83.54	0	0.00
cheerio	650.0	10	0.0	0.00	83.54	0	0.00
cheerio	650.0	15	0.0	0.00	83.54	0	0.00
cheerio	650.0	20	0.0	0.00	83.54	0	0.00
cheerio	650.0	25	186.3	71.34	83.54	64.15	23.21
cheerio	650.0	30	161.0	75.23	83.54	61.12	26.84
cheerio	650.0	35	153.8	76.33	83.54	59.43	28.86
cheerio	650.0	40	155.3	76.10	83.54	59.91	28.29
cheerio	650.0	45	154.7	76.20	83.54	59.29	29.03
cheerio	650.0	50	152.4	76.55	83.54	59.11	29.24
express	859.3	5	277.2	67.75	68.75	66.82	2.80
express	859.3	10	272.0	68.35	68.75	64.75	5.82
express	859.3	15	261.8	69.54	68.75	62.51	9.08
express	859.3	20	251.9	70.69	68.75	59.63	13.26
express	859.3	25	246.7	71.29	68.75	58.04	15.58
express	859.3	30	238.6	72.23	68.75	55.12	19.82
express	859.3	35	234.1	72.75	68.75	54.32	20.99
express	859.3	40	224.9	73.82	68.75	52.09	24.23
express	859.3	45	202.5	76.43	68.75	45.87	33.28
express	859.3	50	195.3	77.27	68.75	41.94	39.00
passport	489.6	5	0.0	0.00	82.61	0.00	0.00
passport	489.6	10	0.0	0.00	82.61	0.00	0.00
passport	489.6	15	0.0	0.00	82.61	0.00	0.00
passport	489.6	20	0.0	0.00	82.61	0.00	0.00
passport	489.6	25	0.0	0.00	82.61	0.00	0.00
passport	489.6	30	0.0	0.00	82.61	0.00	0.00
passport	489.6	35	0.0	0.00	82.61	0.00	0.00
passport	489.6	40	68.6	85.99	82.61	50.05	39.41
passport	489.6	45	68.6	85.99	82.61	49.80	39.72
passport	489.6	50	68.6	85.99	82.61	49.84	39.66

Table 5.6: Test Suite Reduction Using K-means++

			Test Suite Reduction			Fault Detection Loss	
Name	$\|T_{avg}\|$	α	$\|T'_{avg}\|$	$\frac{\|T_{avg}\|-\|T'_{avg}\|}{\|T_{avg}\|} \times 100$	$\|F_{avg}\|$	$\|F'_{avg}\|$	$\frac{\|F_{avg}\|-\|F'_{avg}\|}{\|F_{avg}\|} \times 100$
shortid	17.0	5	0.0	0.00	48.05	0.00	0.00
shortid	17.0	10	0.0	0.00	48.05	0.00	0.00
shortid	17.0	15	5.5	67.70	48.05	42.09	12.41
shortid	17.0	20	5.0	70.68	48.05	42.00	12.58
shortid	17.0	25	4.9	70.94	48.05	42.09	12.41
shortid	17.0	30	4.1	76.08	48.05	39.08	18.67
shortid	17.0	35	0.0	0.00	48.05	0.00	0.00
shortid	17.0	40	0.0	0.00	48.05	0.00	0.00
shortid	17.0	45	0.0	0.00	48.05	0.00	0.00
shortid	17.0	50	2.0	88.24	48.05	25.10	47.76
uuid	17.0	5	7.3	57.35	92.64	91.98	0.72
uuid	17.0	10	9.9	41.91	92.64	92.03	0.66
uuid	17.0	15	9.3	45.10	92.64	91.98	0.72
uuid	17.0	20	10.8	36.76	92.64	91.98	0.71
uuid	17.0	25	11.0	35.29	92.64	91.98	0.72
uuid	17.0	30	8.6	49.26	92.64	91.47	1.27
uuid	17.0	35	8.7	48.77	92.64	91.98	0.71
uuid	17.0	40	7.3	57.35	92.64	91.98	0.72
uuid	17.0	45	9.1	46.32	92.64	91.76	0.95
uuid	17.0	50	8.1	52.29	92.64	91.64	1.08

Table 5.7: Test Suite Reduction Using K-means++ - Continued

We can observe from Tables 5.4, 5.5, 5.6, 5.7 that both K-means and K-means++ clustering algorithm produces approximately similar results for test suite reduction whereas results vary slightly in fault detection loss. The K-means algorithm shows better results for async where the test suite reductions lies between 70% to 80% and fault detection loss lies between 4% to 30% for alpha values 5-50. On the other hand, K-means++ algorithm also have good results for async with test suite reduction between 70% to 80% and fault detection loss between 4% to 30%. Note that, fault detection loss is reduced slightly in K-means++ for async package. However, in other packages like express, body-parser, and uuid, we can observe that fault detection loss is slightly higher for K-means++ as compare to K-means whereas test suite reduction is approximately equal. It is due to different structure of mutation matrix of test suite. The good clustering happens in mutation matrix where the data points are more near to centroid points. The other reason behind these results that algorithm terminates when the step goes to 1. It was observed during experimental evaluation that the algorithm reduces the test suite size until step goes to zero. Therefore, the algorithm could found the reduced test suite in intermediate iterations but did not mark it as final reduced test suite. The algorithm only returns the reduced test suite within defined inadequacy level upon termination of algorithm.

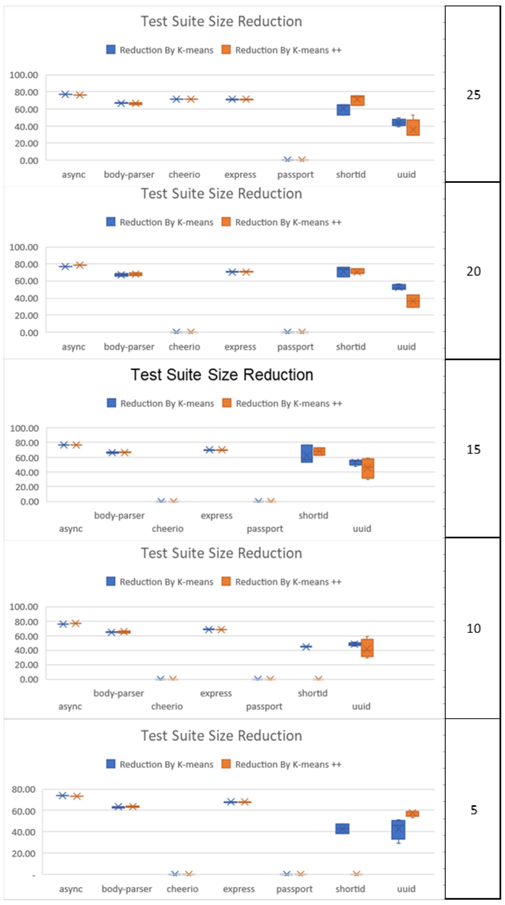


Figure 5.1: Comparison of Test Suite Reduction (α 5 to 25)

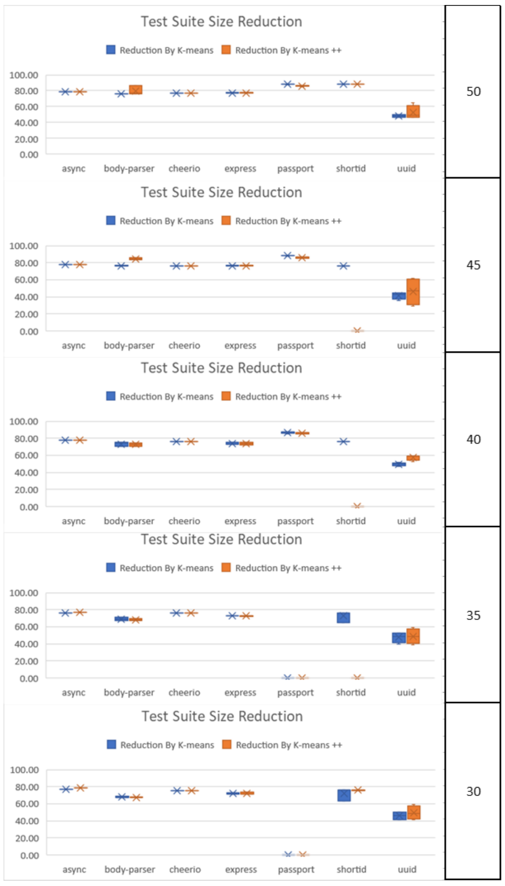


Figure 5.2: Comparison of Test Suite Reduction (α 30 to 50)

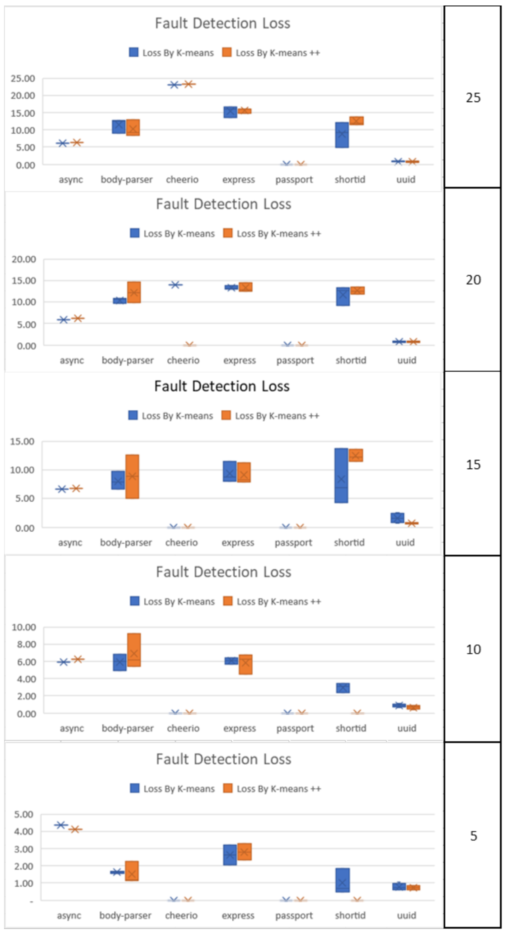


Figure 5.3: Comparison of Fault Detection Loss (α 5 to 25)

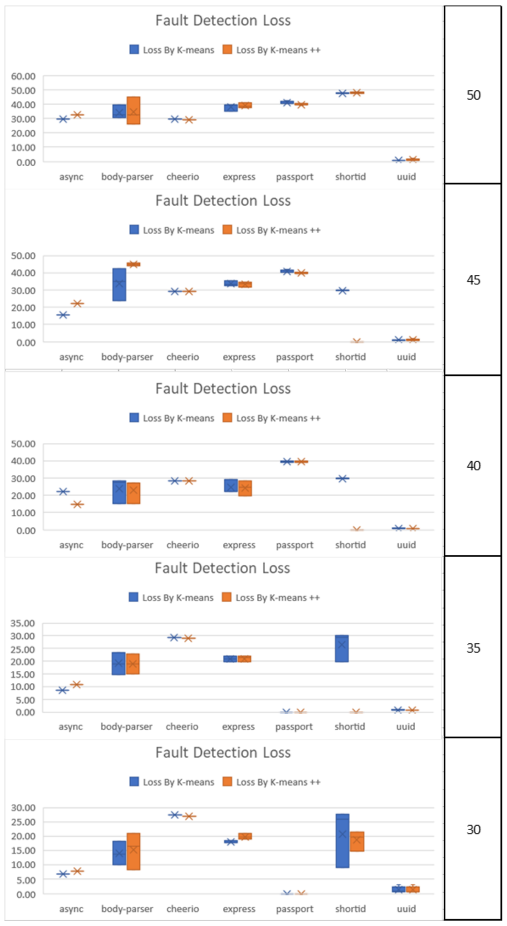


Figure 5.4: Comparison of Fault Detection Loss (α 30 to 50)

In addition to this, we noted that test suite reduction increases with increase in alpha value for both K-means and K-means++ algorithms. As shown in Figures: 5.1, 5.2. Moreover, We can observe from Figures 5.3, 5.4 that fault detection loss also increases with the increase in inadequacy level.

We have faced various challenges while implementing the K-means algorithm to reduce test suite size. First, the node packages test suites have many validation functions (it, expect, should, assert etc.). In each test suite the validation function varies to validate expected and actual values. In one test suite, developer has used *assert* to validate expected and actual values while in another test suite, the *expect* or *should* function is used. This makes it difficult to build a generic parser to read the expected and actual values from test cases. To develop a parser to handle all of the above mentioned validation functions is one of the challenging parts of this project.

Second most challenging issue was clustering the test cases having different numbers of parameters. So far, what we have studied in literature is that the function remains the same for which we want to reduce the test cases. There is a single function having so many test cases, so the existing algorithm tries to reduce the test cases for that particular function meaning, the method body and parameters remain the same for all of the test cases, so it's easy to apply K-means clustering because the size of vectors remains same. However, in Node, we have test cases calling different kinds of functions. So when we converted the test cases inputs and outputs to vectors, the vector size was not the same.

We overcame the above challenges by taking another approach in which we formed the test case feature vectors from the mutation matrix. We created 2d vector for each test case which includes Killed and Non-Killed count of mutants.

Chapter 6

CONCLUSION & FUTURE WORK

This chapter concludes experimental results presented in this work and suggests future research directions.

6.1 Conclusion

In this thesis, we have implemented K-means and K-means++ clustering algorithm for test suite reduction of Node packages. We have implemented a parser that reads the node package's mutation matrix and retrieves the number of killed and non-killed mutants of the test case. Further, the parser creates 2d vectors where one component of vector is killed mutants count and other component is non-killed mutants count. The K-means or K-means++ clustering algorithm clusters the similar feature vectors (test cases) based on the distance. The distance is measured using the Euclidean distance formula. Once clustered, the representative test cases are selected and added to the reduced test suite. To measure the quality of the reduced test suite, we have used mutation score. The implemented algorithms have been tested on 43 versions of seven NPM packages. Out of these, 20 package versions gave test suite reduction for maximum number of alpha values. We can conclude that both K-means and K-means++ clustering algorithm produces approximately similar results for test suite reduction. The results are quite good for async where the test suite reductions lies between 70% to 80% and fault detection loss lies between 4% to 30% for alpha values of 5-50. For inadequacy level 5, the average reduction for body parser, cheerio, express, passport, shortid, and uuid is 63.21%, 0%, 67.72%, 0%, 42.65%, and 45.83% respectively. Interestingly, the fault detection loss in all packages is less than 5% with minimum loss for uuid 0.89% and maximum loss for async 4.36%. Hence, we can reduce the test suite's size by having fault detection loss within range of defined inadequacy level. However, we cannot generalize the results because the algorithm still needs to be executed on more NPM packages with larger test suites.

6.2 Future Work

As next, we plan to investigate how we can make improvements in the performance of algorithm. We aim to use different distance metrics (Hamming distance and Levenshtein Edit distance) in K-means clustering algorithm for test suite reduction and compare the results with traditional K-means clustering algorithm using Euclidean distance.

Bibliography

[1] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. “A methodology for controlling the size of a test suite”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.3 (1993), pp. 270–285.

[2] Sriraman Tallam and Neelam Gupta. “A concept analysis inspired greedy algorithm for test suite minimization”. In: *ACM SIGSOFT Software Engineering Notes* 31.1 (2005), pp. 35–42.

[3] Anil K Jain. “Data clustering: 50 years beyond K-means”. In: *Pattern recognition letters* 31.8 (2010), pp. 651–666.

[4] Yue Jia and Mark Harman. “An analysis and survey of the development of mutation testing”. In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678.

[5] Shin Yoo and Mark Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software testing, verification and reliability* 22.2 (2012), pp. 67–120.

[6] AEVB Coutinho et al. “Test suite reduction based on similarity of test cases”. In: *7st Brazilian workshop on systematic and automated software testing—CBSoft*. Vol. 2013. 2013.

[7] Chunrong Fang et al. “Similarity-based test case prioritization using ordered sequences of program entities”. In: *Software Quality Journal* 22.2 (2014), pp. 335–361.

[8] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[9] Jinfu Chen et al. “Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering”. In: *Journal of Systems and Software* 135 (2018), pp. 107–125.

[10] Carmen Coviello, Simone Romano, and Giuseppe Scanniello. “Poster: CUTER: ClUstering-based TEst suite reduction”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE. 2018, pp. 306–307.

[11] Daniel Flemström et al. “Similarity-based prioritization of test case automation”. In: *Software quality journal* 26.4 (2018), pp. 1421–1449.

[12] A Pandey and K Malviya. “Enhancing test case reduction by k-means algorithm and elbow method”. In: *International Journal of Computer Sciences and Engineering* 6.6 (2018), pp. 299–303.

[13] Diego Rodrıguez-Baquero and Mario Linares-Vásquez. “Mutode: generic javascript and node. js mutation testing tool”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 372–375.

[14] Emilio Cruciani et al. “Scalable approaches for test suite reduction”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 419–429.

[15] Nour Chetouane et al. “On using k-means clustering for test suite reduction”. In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2020, pp. 380–385.

[16] Carmen Coviello et al. “Adequate vs. inadequate test suite reduction approaches”. In: *Information and Software Technology* 119 (2020), p. 106224.

Appendix A

K-means Result Sheets

There are two result sheets of K-means as follow:

1. K-means Results
2. K-means Results Averages

Appendix B

K-means++ Result Sheets

There are two result sheets of K-means++ as follow:

1. K-means Results

2. K-means Results Averages