Mutation Clustering

Shamaila Hussain

shamaila.2.hussain@kcl.ac.uk
Student Number: 0425528

Supervised By:
Professor Mark Harman

King's College London
MSc Advanced Software Engineering 2007/2008
Department of Computer Science

September 5, 2008

Abstract

Mutation testing, a type of white box testing is one of the most important and powerful testing techniques, which guarantees to improve the software quality. This form of testing deals with mutating parts of the program intentionally and then detecting them. The purpose is not to find the faults but to generate an effective test suite, which can detect all the faults in the program.

Mutation testing, in spite of being effective and most powerful, is not widely used in software engineering. Basically because of high computational cost associated with it. The computational cost is due to too many mutants for even small programs. We have experimented with different programs of variable sizes, and was able to come up with the solution to the problem.

We have tried to reduce the computational cost of the mutation testing, reducing the number of the mutants by clustering. K-means clustering algorithm and agglomerative hierarchical clustering algorithm are implemented to cluster the mutants. Thereby, reducing the size of mutants by selecting one mutant from each cluster. The outcome of the results shows that the test set generated from the clusters are more efficient, having a mutation score of 100%, than the test set generated from the random mutants. This report elaborates the experimental steps and results.

Key words: *Mutation testing, clustering*

Acknowledment

I would avail this opportunity to thank my supervisor, Professor Mark Harman, for his support and words of encouragement. He would put me on the right track if I went in the wrong direction, and that happened numerous times.

I would also like to thank Yue Jia from the CREST, for his guidance and help in providing me the programs for clustering. Also the frequent discussions with him gave me a clear understanding of the project and increased my confidence.

Last but not the least; I would like to thank my parents as well for their support and trust in me.

Contents

List of Figures

List of Tables

Abbreviations

LOC....	**L**ines **O**f **C**ode
MS......	**M**utation **S**core

Chapter 1

Introduction

Software testing is one of the vital processes of the software development life cycle. It ensures that high quality systems reach the clients which are bug free and reliable. It is a process of finding faults in a software program, which consists of designing the test cases and executing the software with those test cases. Software testing, previously underestimated, is now given equal importance as software development. According to a research [5], 40-50% of the cost, effort and time is spent on testing and 50% on software development, it is said that this cost is even higher for critical software's like in avionics software's.

There are many techniques of testing software, classified as black box testing, white box testing and grey box testing, each one of these categories target different levels of complexity and coverage.

Mutation testing, a type of white box testing, is one of the most powerful testing techniques. The basic aim of this testing strategy is to generate effective test set, which can detect all the faults, by repeatedly improving the test cases. Mutation testing has two main problems: first, the large number of mutants; and second, the equivalent mutants. Our study focuses on the former problem of the mutation testing.

1.1 Project Aim

The main goal of this project is to overcome one of the problems of the mutation testing: computational cost incurred by large number of the mutants. For that, we cluster the similar mutants together, so that by selecting mutants from each cluster we can reduce the number of mutants required without reducing the power of the set of mutants. Mutant power will me measured in terms of the set of test cases required to kill the entire mutant set.

1.2 Objectives

Having clear, specific and realistic objectives improves the project a great deal. Very clear objectives were outlined for this project, which made it easy to present the deliverables in time throughout the project development.

The main objectives of this project are:

- To have a clear understanding of the mutation testing and clustering technique, and identifying the main issues and problems of mutation testing.
- To reduce the number of mutants by implementing different clustering techniques, in our case we have implemented the k-means clustering algorithm and the agglomerative clustering technique.
- To strive to get an efficient and mutation adequate test set generated from the mutants selected from the clusters.
- To write a detailed and comprehensive document.

1.3 Scope

The scope of this project is made limited in a number of ways. Mutation testing is a challenging strategy. It faces the problem of large number of mutants for even small programs. In addition to this, another problem is that most of these mutants are equivalent to original program. Firstly, we have chosen to deal with the former problem of mutation testing, i.e., large number of mutants. Secondly, it focuses on one of the many ways to deal with the computational cost incurred by large number of the mutants, i.e., by reducing the number of the mutants. Also, there are several different clustering algorithms which can be implemented, but for this project we have only chosen two such algorithms. These restrictions have allowed us to complete the work in the given time constraint.

1.4 Roadmap

The project is divided into phases of development. The first phase will deal with the implementation of the clustering algorithms, and clustering the mutants based on some similarity, to reduce the large number of the mutant's problem. In each cluster, the mutants will be killed by largely the same set of test cases and so we can choose to select only one member of each cluster. The project will start off with the implementation of the k-means clustering algorithm, different values of k will be experimented with, detail in chapter 4, and then the agglomerative clustering technique will be implemented. The experiment will be conducted on five C programs of different sizes and level of complexity. A mutation testing tool need not be implemented, as it is already implemented, all the data will be available from the CREST with the help of Yue Jia, which will help a great deal.

P	x=y+z;	Test Cases
m1	x=y-z;	T1{y=2, z=2}
m2	x=y*z;	T2{y=2, z=3}
m3	x=y/z;	T1{y=2, z=2}
m4	x=y+z+1;	T1{y=2, z=2}
m5	x=y+z+0;	T2{y=2, z=3}
m6	x=y+z+y;	T1{y=2, z=2}

Table 1.1: Original program *P*, associated mutants and test cases

The second stage of the project development, after the clustering of mutants, will be the evaluation as to whether by selecting mutants from clusters we can reduce the number of mutants without reducing the power of the set of mutants.

1.5 Simple Example

We can narrate the purpose of this project by a simple example. Consider a program *P* and some mutants of this simple program in table 1.1. Each one of these mutants are the faulty versions of *P*, which differ from *P* by a single error. These faulty versions or mutants represent the most likely faults made by the programmers. It is safe to say that these mutants can be innumerable for even simple statements like *P*. This forms the basis of the first phase of the project.

The idea behind mutation testing is to kill the mutants by the test cases, so if all the mutants are killed we are left with a good test set that is mutation adequate. The table 1.1 shows the test cases, T1 and T2, which can kill these mutants. But with so many mutants, executing each of the mutants against each test case leads to enormous computation. This computational cost makes mutation testing less feasible to be practised, therefore makes a good and interesting topic for research.

Considerable work is done in this area for reducing computational cost by reducing the number of mutants, discussed in chapter 3. The first phase of this study is based on the problem of tackling the large number of mutants formed. The second phase, however, is based on evaluating and analysing the results of clustering.

1.6 Organization Of Study

The remaining report is divided into following chapters. The chapter 2 gives an elaborated description of the problem statement. It starts off by outlining different problems of the domain, and then narrows off to selecting only one problem to tackle. The next chapter, chapter 3, describes the review from the literature, describing the current status of the research, explaining the key words of the project title and the work already done in this problem domain. The chapter 4 is about the methodology adapted for the project. The chapter 5 deals with the evaluation part of the experiment. This is an important section of the report, as it explains in detail the results achieved. The chapter 6 provides a detailed conclusion of the thesis. Finally, chapter 7 ends the report by suggesting some future enhancements in the project.

Chapter 2

Problem Statement

Software testing has proven to maximize the reliability and quality of the software. Therefore, 50% of the cost and time is spent on testing the software thoroughly. For that purpose different testing techniques have been developed. Out of those, mutation testing has turned out to be most powerful in fault detection. Mutation testing, although being the most powerful testing technique, is not practised and widely used for testing. The major reason for this negligence is the computational cost that is associated with this technique.

2.1 Computational Cost

Let us analyse the cause of the huge computational cost associated with mutation testing, as discussed in [6, 7], which prevent the use of this technique. The main reason of this cost comes from the large number of generating and executing the mutants. A small program can have hundreds of mutants, for example a program less than 100K LOC will generate enormous number of mutants [10]. The C Triangle program used in the program is 50 LOC with 584 mutants. No doubt creating test cases and executing each one of these mutants can be computationally heavy. Each one of these generated mutants must be executed at least once by the test cases which leads to too much of computation. In [10], Wong says that the cost can be measured by two metrics, the size of the test set used to fulfil the criterion and the number of the mutants under observation.

2.1.1 Possible ways to reduce Computational cost of Mutation Testing

There are different ways to reduce this computational cost [6]. The possible approaches are categorized into three different strategies: 'do fewer' approach, 'do faster' approach and 'do smarter' approach. The detail of these can be found in relevant literature.

1. **Do Fewer Approach [6, 9, 18]:** Reducing the number of the mutants without having critical information loss
2. **Do Smarter Approach [6]:** Distributing the computation of mutants over many

machines

3. **Do Faster Approach [6]:** Quick generation and running of the mutants

The limitation of this project is to focus on the 'do fewer' approach, and find a way to reduce the number of mutants leading to reduction in computational cost. Even the title of the project 'Mutation Clustering', further emphasizes the limited scope of the research; clustering the mutants to reduce the number of mutants.

2.2 Hypothesis

We hypothesize that by clustering the large number of the mutants we can considerably lower the computational cost. So that by selecting mutants from each cluster we reduce the size of the mutants. Furthermore, generating a test set that is mutation adequate for the mutants selected from the clusters. Thus, that test set will be efficient in killing all the actual number of mutants. We aim to investigate this hypothesis in this project.

Chapter 3

Literature Review

3.1 Overview

This section will elaborate the main keywords of the research topic, mutation testing and clustering. The first section of this chapter gives a detailed working of mutation testing. The second section explains the clustering technique and the techniques used in this project. The third section highlights the previous work done in this problem area, and the last section explains the validity of our research and how it differs from the current state of the art.

3.2 Mutation Testing

Mutation testing a type of white box unit testing is considered to be the most efficient in detecting faults [7]. This testing technique is based on an original program and numerous faulty versions of that program. These faulty versions are called as mutants. Each of these mutants is formed by mutating the program by one instruction only. See table 3.1 for some possible mutants of a program.

Original Program(P)	Mutant1(m1)	Mutant2(m2)	Mutant3(m3)
a= b+c;	a= b*c;	a= b-c;	a= b+c+c;

Table 3.1: Some possible mutants of a program

The basic aim of the mutation testing is to iteratively produce a powerful test set [1]. This is achieved by intentionally adding faults into the original program and then generating the test cases that detect those faults. This process of generating test cases is continued until all the faults are detected (all the mutants are killed). Hence this is why we say mutation testing produces powerful test set, which detects all the faults.

Furthermore, the mutation operators are responsible in generating mutants of a program; they make syntactic changes to the original program. These syntactic changes reflect the usual mistakes made by the programmers while coding. The mutation operators have two main goals to adhere; introduce typical mistakes the programmers make and enforce testing by executing each branch. The table 3.3, shows the 22 mutation operators of Mothra. The mutation operators are represented by their acronyms. For example, mutation operator 'AAR' is an acronym for 'array reference for array reference replacement', which replaces each array reference in a program by other distinct array reference in the program [9].

The process of mutation testing can be described with an example and a flow chart in figure 3.1, to fully understand the process behind this technique [1, 6]. We will carry on with the example described in the table 3.1.

First of all, the original program P is mutated by applying a single mutation operator. We have 3 syntactically correct mutants; m1, m2, m3 of P, table 3.1. The mutants are generated by applying a mutant operator. Next, test cases are generated. Each test case is run on P, if the output of the P is not correct then a bug is found and P is fixed before the test case can be executed on the mutants. If there are no errors in the P or the errors are fixed, the test cases are executed on each mutant.

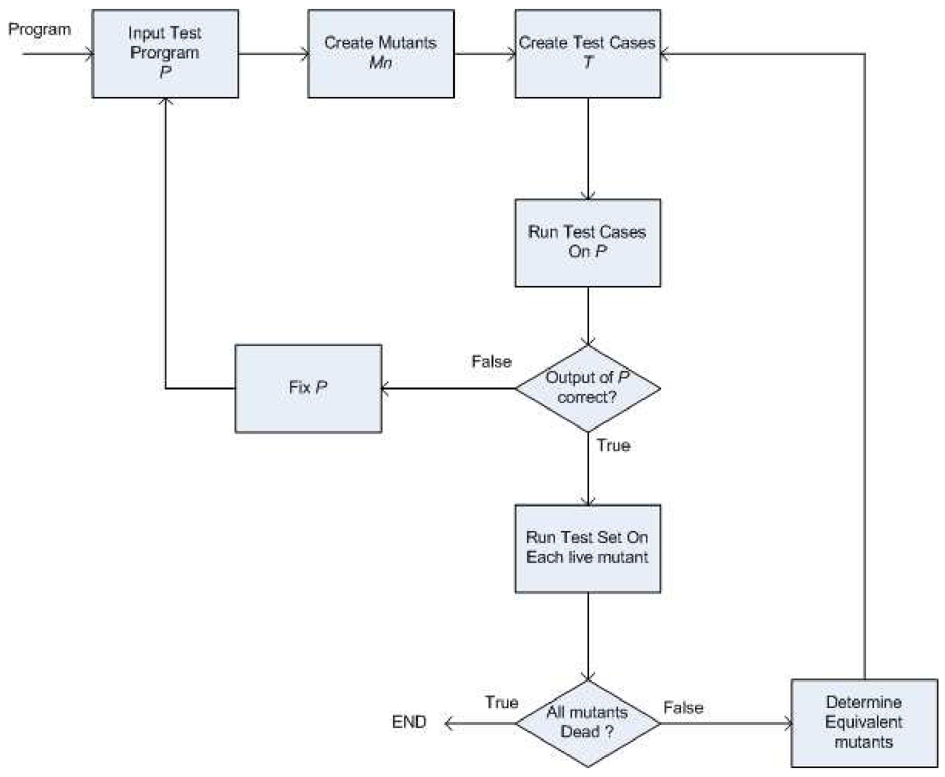


Figure 3.1: Flow chart representing typical process of mutation testing (Adapted from [6])

If the result is different, the fault is detected and the mutant is said to have been killed, as in case of test case 1 in table 3.2. If the result of the test case is same for the original and the mutant program, as in case of test case 2 in table 3.2, then it suggests that either the mutant is an equivalent mutant (which cannot be killed) or the test case is not efficient and needs to be improved. An equivalent mutant is syntactically different but semantically same to the original program. They just increase the computational cost and do not help in determining the effectiveness of a test case; additionally they can never be killed by any test case.

	P a=b+c	*m1* a= b*c
Test case 1 b=4, c=2	a=6	a=2
Test case 2 b=2, c=2	a=4	a=4

Table 3.2: Test cases generated to detect mutants

The test set achieved, after the killing of all the mutants, is checked for its goodness. The goodness of the test set is assessed by a test adequacy criterion. The metric used for the determination of the test adequacy is mutation score. Our aim is to achieve a high mutation score; a score close to 100% makes the test set mutation adequate. Mutation score is calculated by the formula 3.1.

$$MutationScore = \frac{Number\ of\ mutants\ killed}{Total\ number\ of\ non-equivalent\ mutants} \tag{3.1}$$

Where

$$0 <= M.S. <= 1\ or\ 0\% <= M.S. <= 100\%$$

If the test set does not satisfy the adequacy criteria, the test set is improved by adding new test cases until it satisfies the criteria [7]. Hence, the adequacy criterion improves the goodness of the test set and also helps in making new test cases [7].

3.2.1 Computational cost of Mutation Testing

According to Budd [11], number of mutants generated for a program is roughly proportional to the product of the number of data references times the number of data objects. This means that even small programs have large amount of mutants. The computational cost of mutation testing technique is associated with the execution of each and every mutant against the test cases.

3.2.2 Why mutation testing?

Coverage testing does not guarantee fault detection, so no full coverage of branches and statements is possible. However, mutation testing guarantees to detect the fault. This type of unit testing technique assures the improvement of the quality of the software [9],

	Mutation Operators	Description
1	AAR	array reference for array reference replacement
2	ABS	absolute value insertion
3	ACR	array reference for constant replacement
4	AOR	arithmetic operator replacement
5	ASR	array reference for scalar variable replacement
6	CAR	constant for array reference replacement
7	CNR	comparable array name replacement
8	CRP	constant replacement
9	CSR	constant for scalar variable replacement
10	DER	DOstatement end replacement
11	DSA	DATAstatement alterations
12	GLR	GOTO label replacement
13	LCR	logical connector replacement
14	ROR	relational operator replacement
15	RSR	RETURNstatement replacement
16	SAN	statement analysis
17	SAR	scalar variable for array reference replacement
18	SCR	scalar for constant replacement
19	SDL	statement deletion
20	SRC	source constant replacement
21	SVR	scalar variable replacement
22	UOI	unary operator insertion

Table 3.3: 22 Mutation operators of Mothra

and has been empirically proven to be the most efficient testing method to detect faults [11, 13].

3.3 Clustering

Machine learning deals with the extraction of information from the data by transformation, inference and conclusion from the data learning. Clustering falls in one of the sub fields of machine learning known as unsupervised learning [3, 4]. It is unsupervised learning because the algorithm is provided with only the data points and no labels, and the aim is to suitably group those data points [4].

Clustering is a technique used for the grouping of data, for pattern recognition and classification, based on some similar characteristics [2]. It is a form of unsupervised classification as described in [3]. It is defined as the members of a cluster being similar to the other members of that cluster, and dissimilar to members of other clusters [3]. The similarity of the cluster members is measured by some similarity measure. The similarity measure is important for cluster definition for most of the clustering algorithms [3]. The most common similarity or distance measures include Euclidean distance, Hamming distance, etc. The figure 3.2, shows the general idea of clusters. The members of each cluster are grouped beacuse they had some common characteristics, this gives a boundary to each cluster so they can be differentiated form other clusters.

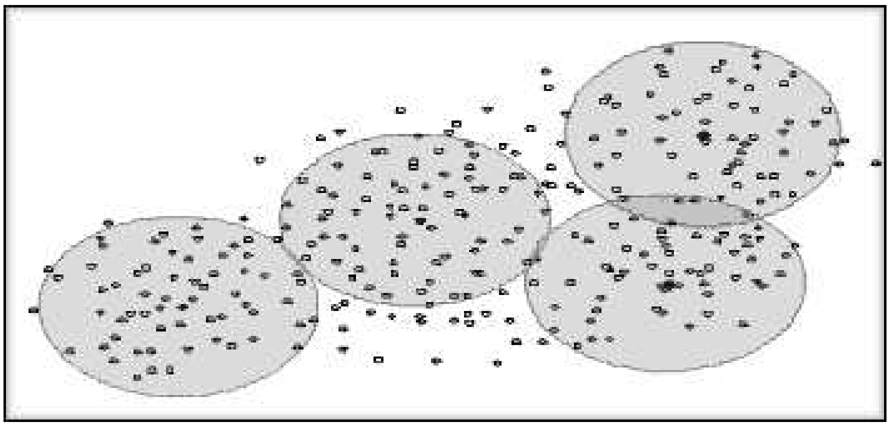


Figure 3.2: General cluster diagram, each one of the marked cluster is different from other clusters

There are different types of clustering algorithms, but the main classifications of clustering algorithms are:

1. Hierarchical clustering methods
   - Agglomerative algorithm
   - Divisive algorithm
2. Partitional clustering methods
   - K-means algorithm
   - Fuzzy c-means algorithm

The k-means clustering algorithm and agglomerative clustering techniques from each classification, which are implemented in this project, are now discussed to give a clear understanding of these techniques and their purpose.

3.3.1 K-means Clustering Algorithm

K-Means clustering algorithm is one of the first, simple and speedy clustering algorithms. The algorithm works as follows:

1. Select k initial clusters (k=1,2,3..).

2. Then assign the data points to those k clusters with smallest distance based on some similarity measure (Eucledean distance, hamming distance).
3. The centroid of the clusters is calculated.
4. Repeat step 2 and 3 until convergence is achieved.

This algorithm depends heavily on the initial choice of clusters, the value of k [4]. The better and thoughtful choice of the initial k value leads to better cluster formation.

3.3.2 Agglomerative Clustering Technique

Agglomerative clustering algorithm is a hierarchical algorithm which goes from bottom to top. It starts off with each data object assigned a distinct cluster (singleton), and continues agglomerating the clusters until a stopping criterion is met [3]. It is represented with a dendrogram, which is a tree showing the merging of the clusters. This dendrogram can be broken at any point to get different clusters at different points. The merging of the clusters depends on the threshold value. If the distance of a cluster to another cluster is less than the threshold value than it is merged with it. Hence if the threshold is 0 then the number of clusters will be same as the data points, on the other hand if the threshold value is too big all the data points will fall into a single cluster.

3.3.2.1 Example [14]

The figure 3.3 shows a dendrogram, abridged from [14]. Each one of the nodes merges two branches into one, which means merging two clusters into one larger cluster. The algorithm is applied to an example of 10 data points. Initially all the data points are single members of the clusters, so the clusters are equal to n. Then for each data point the average is calculated and is merged with minimum average. The dendrogram can be cut at any point to get the desired number of clusters. For example if cut at step 2, we get three clusters.

There are many options of distance measure in hierarchical clustering. The most common [4] are discussed briefly:

1. Average Linkage Clustering

   It is the average distance of the data points in a cluster with all the data points in another cluster. Those two clusters are merged with the lowest average distance.

2. Centroid Linkage Clustering

   The Centroid or mean of each cluster is calculated. Then the distance between the clusters is computed as the distance between their centroids, merging the two clusters with the smallest distance between the centroids.

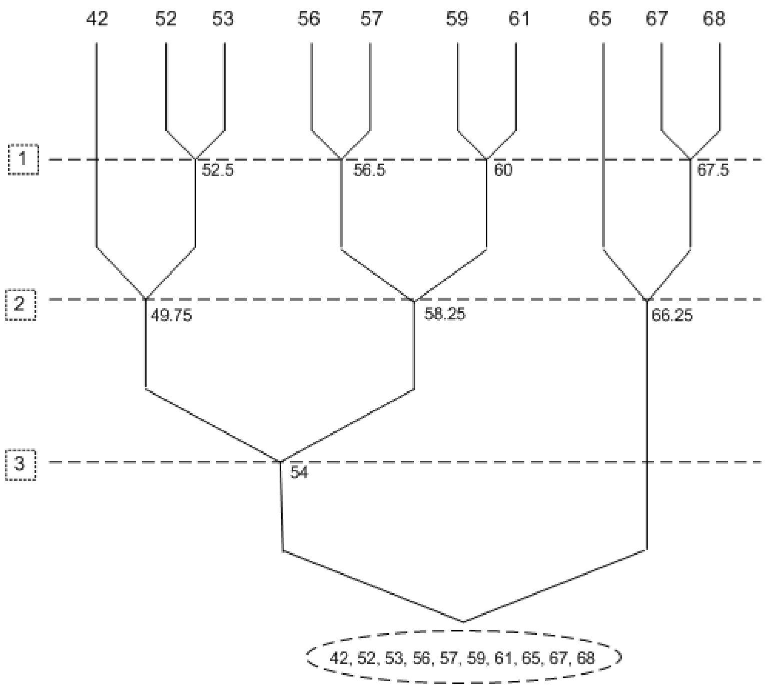


Figure 3.3: Agglomerative dendrogram/tree, showing pair wise merging of clusters at each step (Adapted and abridged from [14])

3. Complete Linkage Clustering

   This is the distance between the farthest data points in two clusters. The two clusters with the minimum farthest distance are merged.

4. Single Linkage Clustering

   It is the opposite of the complete linkage clustering. It is the distance between the nearest data points in two clusters. The clusters with the minimum nearest distance are merged.

3.4 Previous Work

In order to tackle the computational problem of mutation testing, researchers have proposed a number of strategies. Wong and Mathur in [7] proposed a way to reduce the computational cost, by reducing the number of mutants. Mathur suggested using the approximation technique to reduce the number of the mutants and another technique of random mutant selection. These two approaches are referred as ‘do fewer’ approaches in [6] or alternate mutation. There are other approaches as well. For example, ‘do smarter’ approach, this emphasises on dividing the computational cost on several machines. Another approach called as ‘do fewer’ approach is based on generating and running each mutant as quickly as possible.

All these are the approaches to minimize the computational cost, but we will only address the 'do fewer' approach, describing other approaches in detail is out of the scope of this project and report.

3.4.1 Selective/ Constrained Mutation Criterion

The idea of this approach is to select only those mutants which are different from other mutants in the true meaning. It is also called as constrained mutation, as it deals with specific types of mutants. For example, considering only ABS mutants and ignoring the remaining mutants [10]. This idea was first conceived by Wong et al. [8]; they suggested mutating with only critical mutation operators.

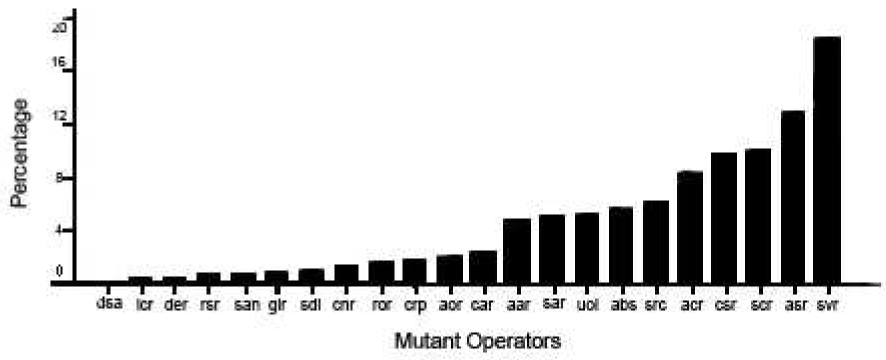


Figure 3.4: Number of mutants produced by the 22 mutation operator (Adapted from [9])

The application of all the mutation operators leads to the generation of large number of mutants. This strategy works by minimizing the number of mutation operators, and getting testing strength with fewer mutants [6]. Figure 3.4 shows the percentage of the mutants formed by the 22 mutation operators for a set of 28 programs. The scalar variable replacement operator (SVR) is evidently the most dominant operator. There are different variations of the selective or constrained mutation; we will discuss them briefly in the following paragraphs.

Mathur suggested not using the operators which produce the most mutants, hence the name selective mutation [12]. Mathur excluded the operators like, scalar variable replacement (SVR) and the array reference for scalar variable replacement (ASR) operators.

This approach was later extended by Offutt. Experimental results shown by Offutt in [9] described that application of only 5 mutant operators provided the same coverage as non-selective mutation, this approach is called as the sufficient mutation. This significantly reduced the cost to 50 times with large programs and 4 times with small programs. It reduced the number of mutants to 60%. According to the work in [9] the key critical operators are ABS, AOR, LCR, ROR and UOI. This subset of mutation operators is sufficient, if they produce fewer mutants and their strength is same as all the mutation operators. Offutt's strategy also ignored the mutation operators that generate large number of mutants like SVR and ASR. This study showed that, a maximum of 99.99% mutation score was achieved with two selective mutation adequate test sets of ten subject programs with a 24% saving in the number of mutants [18].

Wong in [10], emphasised on selecting mutant operators based on their significance in fault detection, operators like ABS and ROR. Wong, by selecting these operators, was able to reduce the number of mutants by 80% and the test cases by 40% to 58%.

Offutt in [9] says that the selective mutation is as powerful as the non-selective mutation and results in less computational cost by reducing the number of mutants. Also, it was aimed that the mutation score be over 95% with selective mutation, since test set with mutation score over this percentage are considered to be effective.

3.4.2 Random Mutant Sampling

This strategy selects random mutants from the large set of mutants. It examines a small subset of randomly selected mutants, ignoring the others. This subset is then tested against the test set and its sufficiency is determined. In case not satisfied, a new sample is selected randomly, and this process is continued until the test set is efficient.

Budd in [11] experimented by selecting only 10% of the mutant samples found that the test cases generated from this sample overlooked only 1% of the non equivalent mutants.

3.5 Justification of the undertaken research

Although researchers have proposed several strategies to control the number of mutants, still they are not very precise. It is still an open research area, and therefore attracts and invites more research to handle the problem.

The two approaches of the ‘do fewer’ strategy are not very optimal. According to [17], the selective mutation ignores some critical mutation operators. The selective mutation by Offutt [9], may lead to possible elimination of some critical mutation operators since it was proposed using only 5 operators out of 22. It is difficult to find a good sufficient subset of mutation operators. This strategy depends heavily on the efficient selection of the mutation operators [10]. Similarly, the random mutant sampling, works better with small randomly selected subset of mutants [17]. This technique is not very precise and efficient, as it is just a random selection of mutants from a large number of mutants, which clearly ignores some critical mutants altogether.

The motivation behind this project was to come up with an optimal way of dealing with the number of mutants.

Chapter 4

Experimental Methodology

4.1 Introduction

This chapter explains the methodology and the strategy used in the under taken project: mutation clustering. We have divided the methodology in specific steps:

Step1: Clustering of the mutants of five different programs.
Step2: Evaluating the effectiveness of the test set generated from the clustered mutants.

4.2 Experiment Setup

In order to evaluate the clustering of mutants, we conducted experiments on five different C programs. Each one of these programs has different number of mutants and varying test cases. The size of the programs varies from 50 to 750 lines of code, table 4.2. To proceed with the experiment we eliminated all the pseudo-equivalent mutants. We call those mutants pseudo-equivalent which are not killed by any test case. These mutants are represented as either all 1s or all 0s.

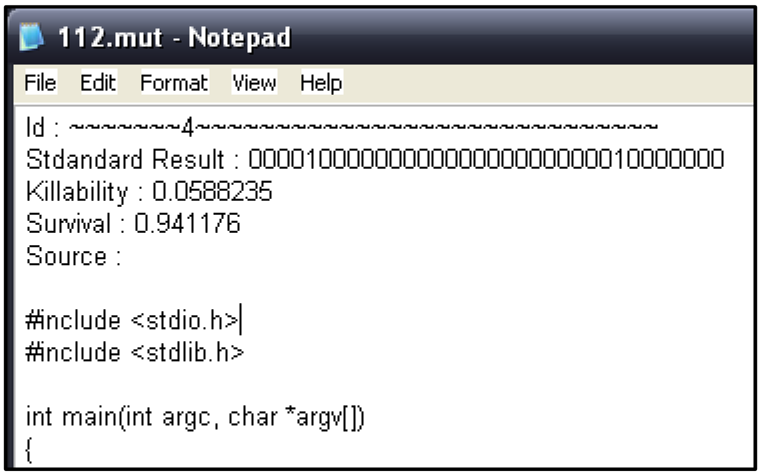
	tc1	tc2	tc3	tc4	tc5	tc6	tc7	tc8
m1	1	0	0	0	1	0	1	0
m2	1	0	1	1	1	0	1	1
m3	0	1	0	0	0	1	1	0
m4	0	1	1	0	1	0	1	0
m5	0	0	0	0	1	0	0	0
......	......	......	......	......	......	......	......	......

Table 4.1: Data strucure representing mutants m1, m2, etc. in terms of test cases tc1, tc2, etc that kill them

The data that we worked on consisted of mutants. These mutants were generated by a C mutation testing tool MILU [21]. Each mutant was represented by the test cases that kill it and the test cases that do not kill it. The table 4.1 shows a data structure that keeps information about the test cases that kill a mutant. For example, the mutant m1 is killed by test cases tc1, tc4 and tc7. Refer to Appendix A, for mutants of triangle program, to get the idea of the kind of data that was experimented with. We give a brief explanation of the programs used for experiment, the table 4.2 summarizes the specifications of the experimetal programs. .

4.2.1 Triangle Program

One of the programs chosen is the Triangle program with 50 lines of code, having 584 mutants. Triangle programs helps in determining the type of the triangle from the length of the three sides [19].The data was available from a *.mut* file generated from a mutation testing tool. See figure 4.1 for the *.mut* file. We were only interested in the *id* and the *standard result* from the file. The size of the test set was 34. After deleting the pseudo-equivalent mutants, we were left with 512 mutants.



```
Id : ~~~~~~~~4~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Stdandard Result : 00001000000000000000000010000000
Killability : 0.0588235
Survival : 0.941176
Source :

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
```

Figure 4.1: Input data from Triangle program (*.mut* file)

4.2.2 TCAS Program

Traffic collision avoidance system, also known as TCAS, is an aircraft collision avoidance system for reducing collisions of aircraft in the air. TCAS was another program used for the experimentation. This program of 150 LOC had 856 mutants and 1608 test cases in all. The deletion of the pseudo-equivalent mutants left us with only 585 mutants.

4.2.3 Schedule2 Program

Schedule2 of 350 LOC was another program under experiment, with 972 mutants and 1399 test cases. It is a program that prioritize the schedulars [19]. The removal of pseudo-equivalent mutants left us with 827 mutants.

4.2.4 Totinfo Program

Totinfo of 500 LOC was the fourth program that we experimented with, it is a program that is used to calculate statistics fom the input data [19]. There were 1879 mutants of this program and a test suite of 1052 test cases, mutants reduced to 1794 after the deletion of pseudo equivalent mutants.

	Program	LOC	Number of mutants	Number of test cases
1	Triangle	50	584	34
2	TCAS	150	856	1608
3	Schedule2	350	972	1399
4	Totinfo	500	1879	1052
5	Printtokens	750	506	2021

Table 4.2: Specifications of five experimental Programs [19]

4.2.5 Printtokens Program

The last program that was clustered and evaluated was Printtokens of 750 LOC, which is a lexical analyser [19]. The program had 506 mutants and 2021 test cases associated with it. The 506 mutants were reduced to 345 after the removal of pseudo-equivalent mutants.

4.3 Clustering Methodology

Clustering is one of the major implementation parts of the project. We aim to reduce the number of mutants by clustering or grouping together the similar mutants. Those mutants are clustered together which are largely killed by the same set of test cases, this will allow us to choose only one mutant from each cluster, subsequently reducing the number of the mutants.

The chosen clustering algorithms are k-means clustering algorithm and the agglomerative clustering algorithm. We will outline these algorithms with respect to their working in the project.

4.3.1 K-means Clustering Algorithm

The general k-means clustering algorithm is described in chapter 3. This algorithm is sensitive to the initial selection of clusters and the value of k. K-means algorithm has many variations [15]. One of these variations is to select good initial values as the k-means clustering depends heavily on the initial selection of the k values. Wrong selection of k values leads to wrong formation of clusters. Hence, in this project we have carefully selected the initial k mutants. Although the initial mutants are selected randomly, but they are truly distinct, this leads to the formation of truly distinct clusters. Also, the initial value of k is determined from the agglomerative algorithm, discussed in subsection 4.3.4..

1. Decide upon an initial k cluster value, the value of k must be less than the number of distinct mutants (value of k is determined from agglomerative algorithm).
2. Randomly select k distinct mutants.
3. Calculate the centroid of the cluster.
4. Calculate the distance, using the hamming distance, of each of the mutants to the centroid.
5. Group the mutants with the clusters with the minimum distance.
6. Recompute the centroid of the gaining cluster (and the losing cluster).
7. Repeat the steps 4-5 until convergence is achieved and the mutants do not move.

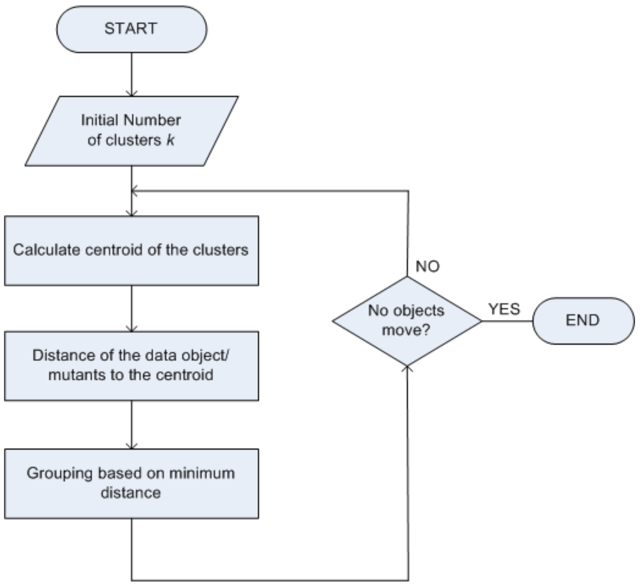

START
Initial Number of clusters k
Calculate centroid of the clusters
Distance of the data object/ mutants to the centroid
Grouping based on minimum distance
No objects move?
NO
YES
END

Figure 4.2: Flow chart of k-means clustering algorithm (Adapted from [22])

4.3.2 Agglomerative Clustering Algorithm

The agglomerative algorithm does not require any prior specification of number of clusters unlike k-means clustering algorithm. However, this algorithm depends on a threshold value T [16]. The data objects are merged pair wise if the distance between them is below the threshold value. The clustering stops when the distance between the clusters exceed the threshold value. This gives us the distinct number of clusters, and this is the number that we use as k in k-means clustering algorithm.

We have done a bit of variation by merging two techniques of agglomerative clustering. Firstly, we start off by merging clusters based on distance lower than T. Afterwards, when the merging of clusters stops due to distance becoming greater than T; the clusters are merged based on minimum distance.

The clustering of the mutants with the agglomerative algorithm followed the following algorithmic steps:

1. Initially numbers of clusters are equal to the number of mutant's n.
2. Then calculate the distance of each cluster to the other cluster, distance measure used is the average linkage.
3. Decide upon a threshold value T, we have set T=2.
4. Merge the two clusters pairs having the distance lower than the threshold value.
5. Repeat steps 2-4, this process continues until the distance becomes greater than the value of T.
6. Now here we start merging the clusters not based on the threshold value but on the minimum distance value.
7. Calculate the average linkage and merge the two clusters with minimum average distance.
8. Repeat step 6-7, this pair wise merging continues until all the mutants are agglomerated into one big cluster of size n.

4.3.3 Similarity Measure

Similarity measures are essential in clustering of the objects as they determine the similarity between two clusters or data objects, or dissimilarity of two clusters. The similarity measure used is the hamming distance. Hamming distance is the number of substitutions needed to change from one string into another. For example consider three mutants m1, m2 and m3, in table 4.3:

m1	1	1	0	0	1	0	1

m2	1	1	1	1	1	0	0

m3	0	1	1	1	1	0	0

Table 4.3: Three mutants m1, m2, m3

	m1	m2	m3
m1	0	3	4
m2	3	0	1
m3	4	1	0

Table 4.4: Distance matrix representing distance between mutants

The Hamming distance between the two mutant strings m1 and m2 is 3, as the two strings differ from one another by 3 digits. It is the XOR between the bits.

We calculate the distance between three mutants m1, m2 and m3. We represent the pair wise distances between mutants with a distance matrix, table 4.4. This n*n matrix is updated after new clusters are formed. Distance between m1 and m2 is 3, whereas, distance between m1 and m3 is 4. On the other hand distance between m2 and m3 is 1. Those mutants are clustered with the smallest distance, because smaller the value of dissimilarity, the more similar are the two clusters. If based on this distance, generally speaking, m2 and m3 will be clustered together and m1 will remain un-clustered.

Why we did not use Euclidean distance, is because Euclidean distance is the distance between two points which is measured using the Pythagoras Theorem. This leads us to say that with mutants being our central data, we can have no such parameters. On the other hand, Hamming distance suits our needs.

4.3.4 Deciding value of *k* for *k*-means Clustering Algorithm

The first time selection of k for the mutants of a particular program is not straight forward as it may seem. The results will not be good if the value of k is unrealistic. Since our implemented algorithm carefully chooses distinct random mutants, so the initial k may not always be the half of the total number of the mutants for the first time. We have used the agglomerative algorithm to determine value of k. Agglomerative algorithm, merges two clusters based on the distance lower than the threshold value, it stops merging the clusters when the distance start becoming more than the threshold. The threshold value was 2, so the point where the algorithm stopped gave us the number of clusters which were very distinct. Subsequently, the first initial k value achieved gives us smaller intra-cluster distance and larger inter-cluster distance.

Furthermore, the later selections of the value of initial k, are half of the first selection. For example, we had 512 mutants from the Triangle program. The first run of k-means clustering algorithm was based on k=64. For the second run, we chose k=31 and the next run was experimented with k=15.

The largest of these values of k, is considered to be optimal. The reason is that the clusters formed based on this value have very less intra-cluster dissimilarity, since mutants in each cluster are killed by same set of test cases. In addition to that, an optimal value will result in a mutation adequate test set.

4.4 Evaluation Methodology

Once the mutants are clustered by the chosen clustering algorithms, the evaluation of the clusters is done in order to support our results. Before we describe the evaluation approach, we mention the notations we will be using in this section and onwards.

- *Sc* is selection of one mutant from each cluster.
- *Tc* is the test set selected from the *Sc*.
- *Sr* is the actual size of all the un-clusterd random mutants.
- *Tr* is the test set selected from *Sr*.

4.4.1 Evaluation approach

In order to do the evaluation the following approach is followed, this approach is also depicted in figure 4.3.

1. Firstly one mutant from each cluster is selected randomly for each of the clustering algorithms (K-means and Agglomerative), this selction is called *Sc*.
2. Now the test set *Tc* is generated for the mutant selection.
3. The test set is generated by using the greedy approach, which only selects those test cases which kills the most mutants in Sc. This way we have a very strong set of test cases, *Tc*. *Tc* is mutation adequate for *Sc*.
4. *Tc* is kept fixed and test set is generated from the random mutants *Sr*, *Tr* is of the same size as the *Tc*.
5. *Tr* is generated 30 times for the fixed number of *Tc*.
6. Both *Tc* and *Tr* are observed to kill the total mutants *Sr* and the efficiency for each set is analyzed based on the mutation score.

4.4.2 Greedy strategy for adequate test set generation Tc

We began by experimenting with the selection of *Tc* from the clusters, in the hope to achieve 100% Mutation score for *Tc*. One of the selections of *Tc* was based on selecting test set *Tc* randomly from the mutants without any greedy selection. However, we noticed that this strategy does not provide a good efficient and mutation adequate test set. Eventually, this led us to follow the greedy approach in the selection of test set *Tc* from the clusters.

Now we explain how the greedy approach works and how it helped us to achieve a highly efficient test set *Tc*. According to this strategy, we achieve local optimum at each

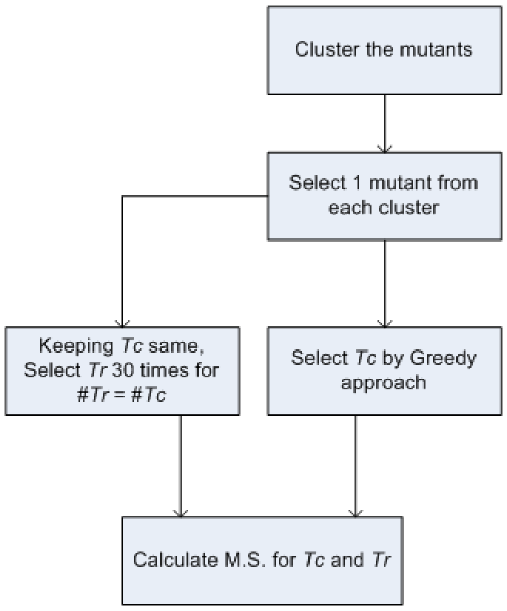

Cluster the mutants
Select 1 mutant from each cluster
Keeping *Tc* same, Select *Tr* 30 times for #*Tr* = #*Tc*
Select *Tc* by Greedy approach
Calculate M.S. for *Tc* and *Tr*

Figure 4.3: Flow Chart of the methodology

step in the hope to achieve global optimum [20]. It makes decisions based on the current information without worrying about the effects of the decisions in the future.

How this strategy works in our evaluation is, that mutants are selected from different clusters then for each step that test case is chosen which kills the most mutants. The mutants killed by that test case are removed and the test cases that kill the remaining mutants are calculated. Then again that test case is selected that kills the most mutants. This process of selecting test cases continues until all the selected mutants are killed. This results in a test set Tc that is mutation adequate for the selected mutants, see the flow chart in figure 4.3.

Chapter 5

Experimental Results and Analysis

5.1 Aim

The aim of this section is to describe the evaluation to investigate our hypothesis. The mutants are clustered and the clusters are evaluated to find that the test cases selected from the clusters are better and efficient in killing the mutants than the test set selected randomly. This is done by calculating how many percentages of the mutants are killed by the test set from clusters and how many by the test set from the random mutants, or in simpler words, by calculating the mutation score for each test set.

5.2 Clustering Results

We cluster the mutants based on the similarity. For each of the clustering algorithms we have repeatedly experimented with different values of the number of clusters to come up with an optimal value for each program under test. The clustering results in figures 5.1, 5.2, 5.3, 5.4, 5.5 are just for the k-means clustering algorithm, to give an idea of the clustering results.

The highest values of k used for the programs under test are, for Triangle program k=64, for TCAS program k=200, for Schedule2 program k=83, for Totinfo program k=173 and for Printtokens k=68. For all these values of k, the intra-cluster distance among the mutants is as less as 0, 1 3, 4. This shows that all those mutants are clustered together which are killed by the same test cases. However, if we start reducing the number of clusters or the value of k, the intra-cluster distance starts increasing, such that for some clusters the intra-cluster distance reaches to 300. This is because the initial selection of the number of clusters are less and this forces the not so similar mutants to be merged in that cluster.

1. **Triangle Program:** The clustering of the 512 mutants from triangle program considerably reduced the number of the mutants. With k-mean algorithm 512 mutants were reduced to 64 clusters for the first run of the algorithm, k =64 being an optimal value, figure 5.1. With Agglomerative clustering technique the 512 mutants were re-

duced to 62 clusters. Subsequently, the clusters were further reduced to k=31 and k=15 with *k*-means, whereas, 30 and 16 with agglomerative algorithm

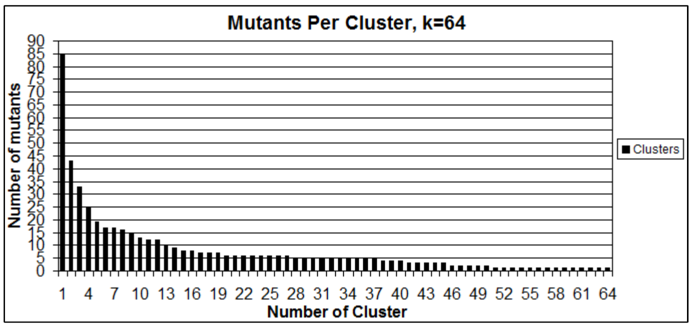


Figure 5.1: *K*-means clustering algorithm clusters of Triangle program mutants

2. **TCAS Program:** *K*-mean clustering algorithm reduced the 585 mutants to 200. *K*=200 is an optimal choice since the mutation score is close to 100%, discussed in section 5.3. With the agglomerative clustering algorithm, the 585 mutants are reduced to 212 mutants, figure 5.2. Additionally, the further reductions with *k*-means, resulted in 100 and 50 clusters and with agglomerative 73 and 48 clusters.

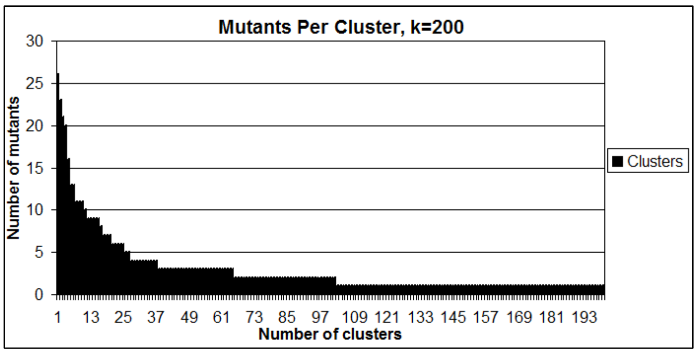


Figure 5.2: *K*-means clustering algorithm result on TCAS program

3. **Schedule2:** The *k*-mean algorithm clustered the 827 mutants of Schedule2 program into 83 clusters, figure 5.3. Similarly the agglomerative algorithm reduces the 827 mutants to 83 clusters. Further reduction in the number of mutants by k-means algorithm, led to 40 and 20 clusters. On the other hand, using the agglomerative algorithm, the mutants were reduced to further 32 and 24 clusters.

4. **Totinfo:** The *k*-means algorithm reduced the 1879 mutants to 173 clusters, figure 5.4, further reduction in the number of mutants led to the formation of 85, 40 and 20 clusters. However, with agglomerative algorithm, the optimal size of the cluster was 173 like *k*-means. Similarly, it also formed clusters of sizes 84, 45 and 19.

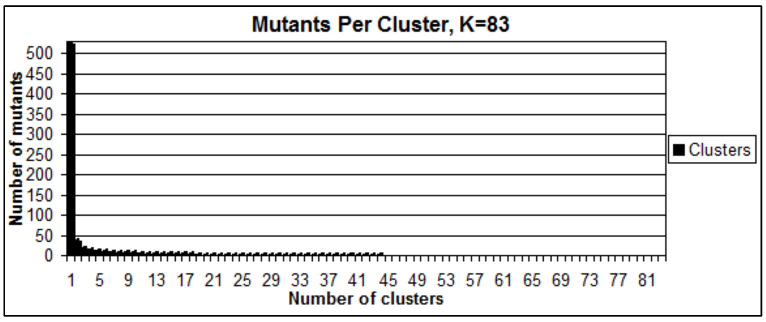


Figure 5.3: K-means clustering algorithm result on Schedule2 program

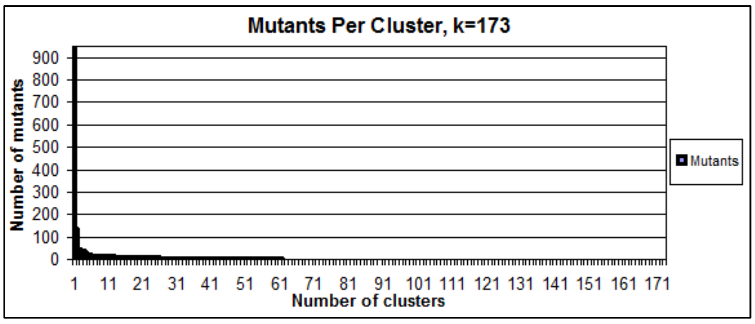


Figure 5.4: K-means clustering algorithm result on Totinfo program

5. **Printtokens:** The 506 mutants of Printtokens program was reduced to 68 clusters with both the k-means and agglomerative algorithm, figure 5.5. Reducing the size of the mutants further, formed 35 and 15 clusters with k-means algorithm and 31 and 16 clusters with agglomeartive algorithm.

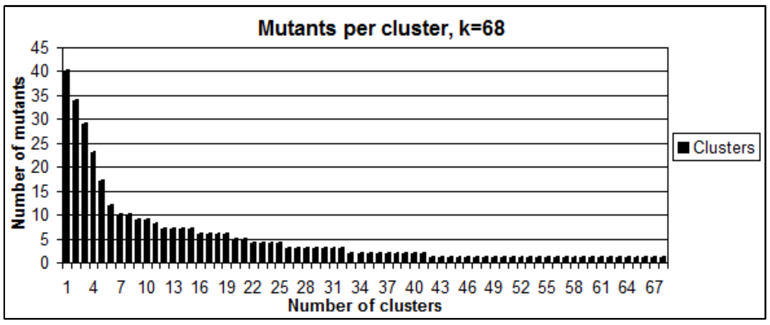


Figure 5.5: K-means clustering algorithm result on Printtoken program

5.3 Experimental Results

This section explains the efficiency of test set for each program under evaluation. Each graph represents the 30 different selections of *Tr* with a fixed number of *Tc* and their corresponding mutation scores. Only two graphs for each algorithm are shown, for complete results refer to Appendix B.

5.3.1 Results for the Triangle Program

Each run of k-means clustering algorithm has a fixed same cluster test set *Tc*, and 30 different random test set *Tr*. In figure 5.6 the graph *a* shows the three runs of the k-means clustering algorithm for different samples of the same number of *Tc*, i.e., 13. Figure 5.6 graph *b* shows the four runs of k-means algorithm on different selection of mutants from each cluster for same value of Tc=9. In the graph *a*, the mutation score of *Tc* is closer to 100 %. The mutation score of *Tr* start off at 85% and gradually increases, although at one point it become closer to *Tc*.

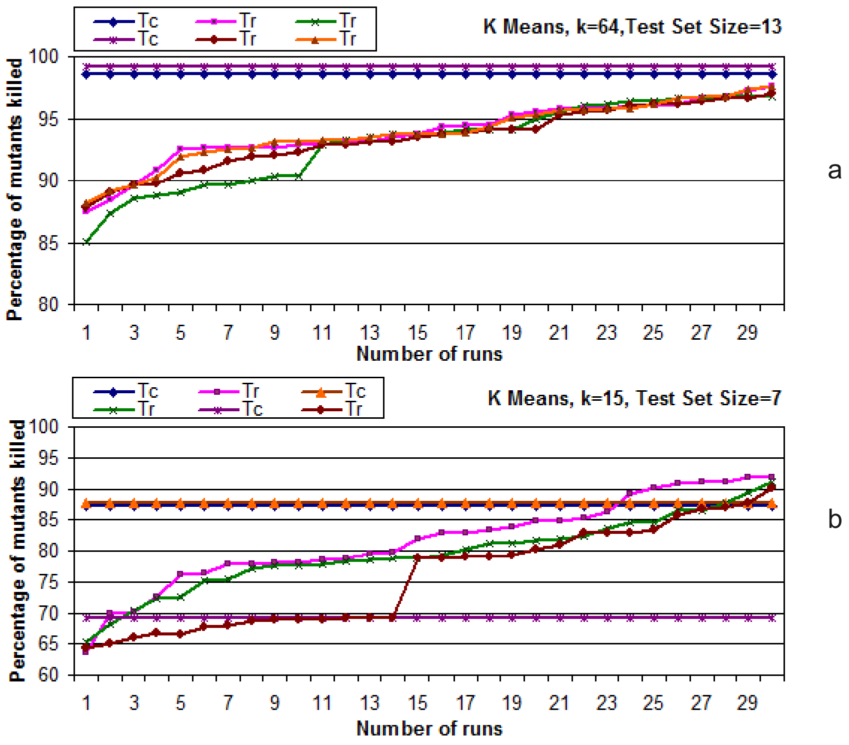
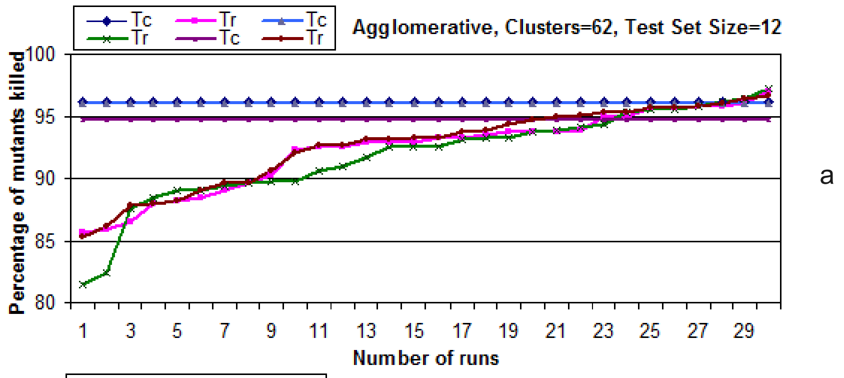


Figure 5.6: K-means clustering algorithm result on triangle program for k=64 and k=15

Applying the agglomerative clustering algorithm on triangle program of 512 mutants, yields 62 clusters. Figure 5.7 shows, the efficiency of *Tc* and *Tr* for three samples of 62 clusters and 16 clusters respectively. The graph with greater number of clusters, i.e. 62, shows that the MS of closer and below 95% is achieved for Tc, which is not very good It is also worth mentioning, that the test set size 34 is reduced to 13 and 12 with k-means and agglomerative algorithms respectively.



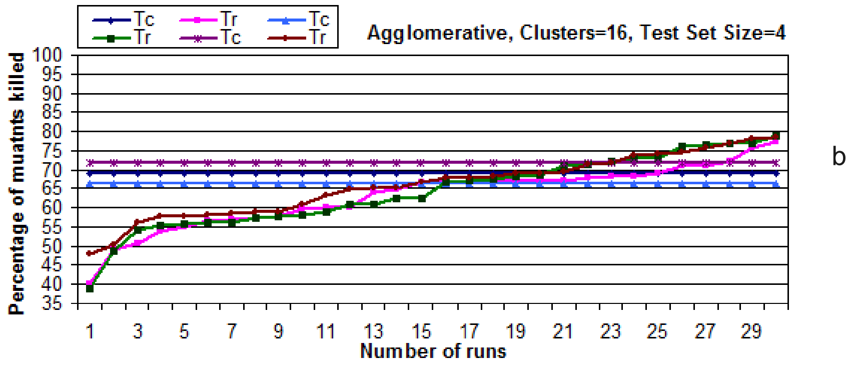


Figure 5.7: Agglomerative clustering result on triangle program for clusters=62 and clusters=16

5.3.2 Results for the TCAS Program

Figure 5.8 graph a shows the three runs of the k-means algorithm having the test set size 19 and k=200. The mutation score of Tc for the three samples is 99.3%. Whereas, the mutation score for the three samples of Tr is less than 95%. Similarly, the graph b in figure 5.8 shows the three runs of k-means algorithm of the same test set size of Tc=8 and k=50. Evidently the mutation score of Tc has decreased in k=50 than in k=200.

On the other hand, the agglomerative clustering algorithm, resulting in 212 clusters, achieved a mutation score of 100% for Tc with the test set size being 21. The graph a in figure 5.9 shows the efficiency of Tc and Tr of five different samples of Sc, and same test size. The mutation score of Tr for the five samples is less than 95%. Also the graph b of figure 5.9 shows the mutation score of Tc and Tr for 48 clusters. The MS for Tc has decreased considerably well below 95%.

The optimal number of clusters is 200 and 212 resulting from k-means and agglomerative algorithm. Furthermore, these algorithms result in the reduction of test cases to 19 and 21 from a total of 1608 test cases, this reduced test set is strong enough to achieve a mutation score of 100% or above 99% in case of k-means algorithm.

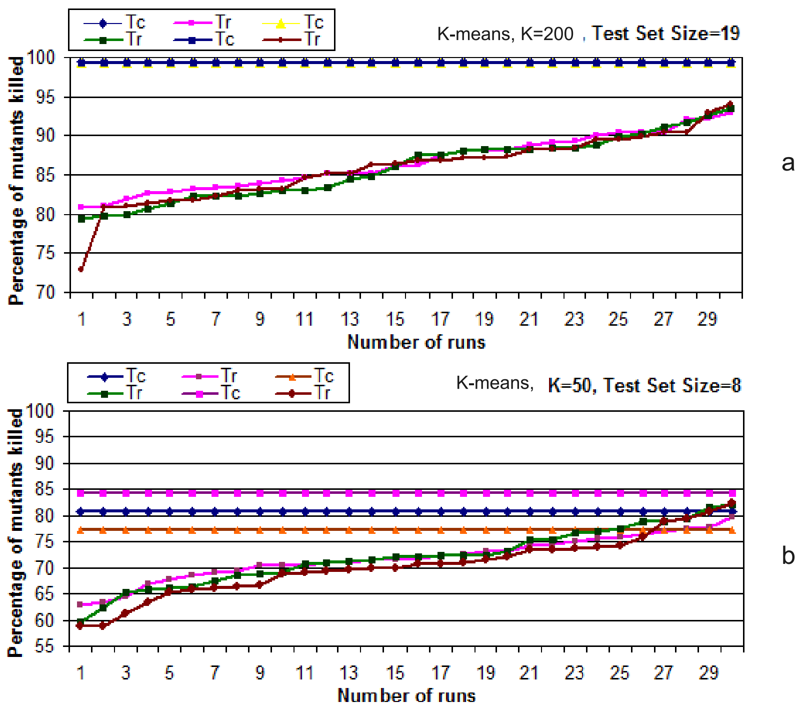


Figure 5.8: *K*-means clustering algorithm result on TCAS program for k=200 and k=50

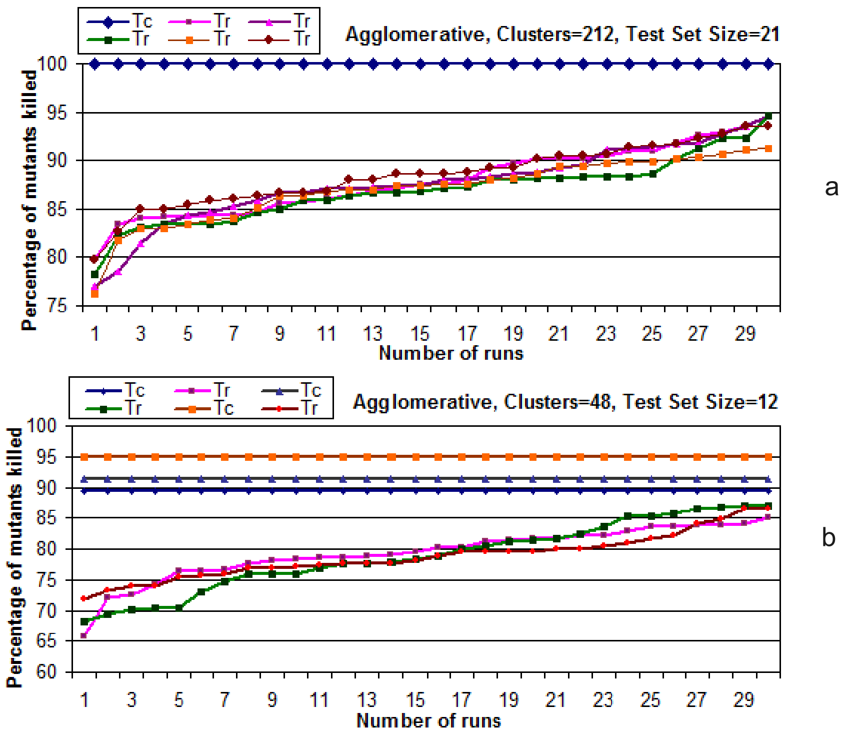


Figure 5.9: Agglomerative clustering algorithm result on TCAS program for clusters=212 and clusters=48

5.3.3 Results for the Schedule2 Program

The efficiency for *k*-means algorithm on the schedule2 program is shown in figure 5.10. The graph *a* for *k*=83, shows the MS of 100% for *Tc*. However, the MS for *Tr* reaches 99.5% at one point. It is also noteworthy that the MS of *Tr* starts off at 79.4% in this case. Similarly the graph *b* of figure 5.10 reveals the MS for *Tc* and Tr for k=20. The MS for *Tc* has decreased. Also the MS for *Tr*, starting off at 76.5% crosses the MS of *Tc* at one point.

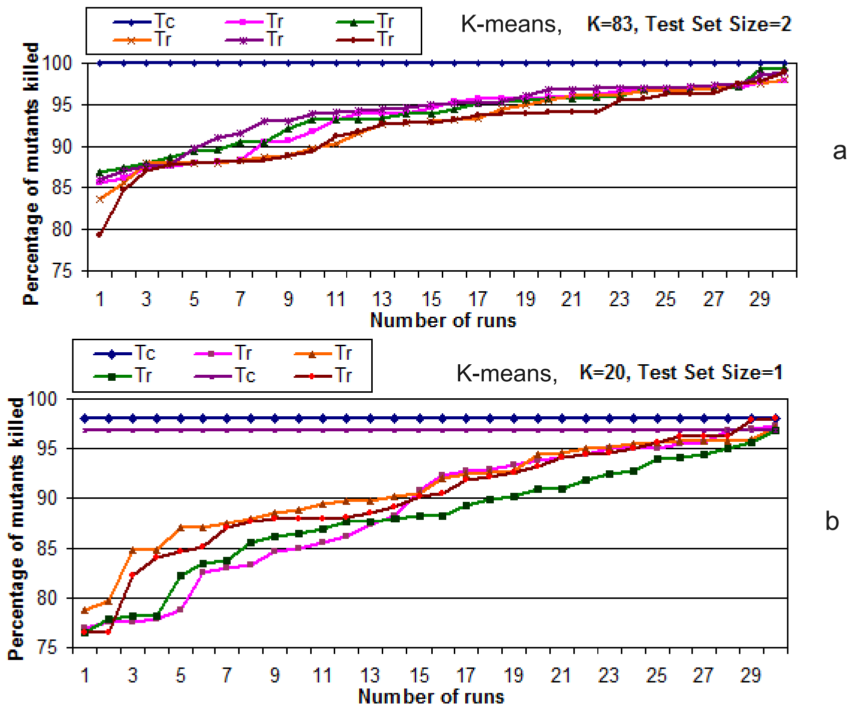
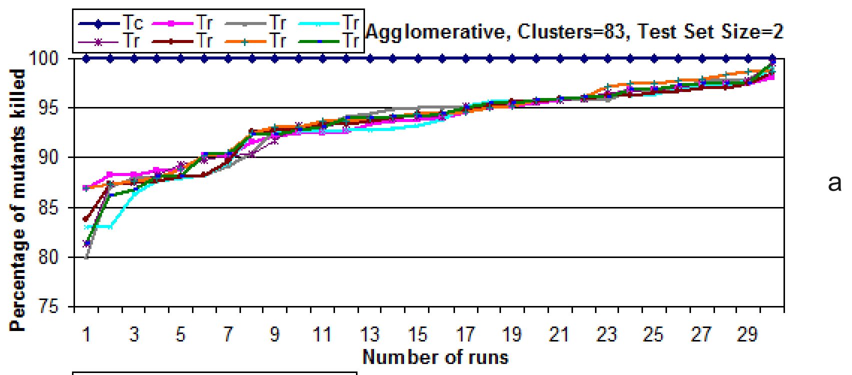


Figure 5.10: *K*-means clustering algorithm result on Schedule2 program for k=83 and k=20

The figure 5.11 shows the efficiency results of agglomerative algorithm for 83 and 24 clusters. The MS of *Tc* for 83 clusters is 100%, whereas for *Tr* it starts off at 79.9% and maximum MS achieved is 99.5%. On the other hand, lowering the cluster value in graph *b*, reveals a MS of maximum 99% and minimum 97.8% for *Tc*. However, the MS of *Tr* crosses the efficiency of *Tc* at one point. Again, the test suite size is reduced to 2 from a total of 1399 test cases.



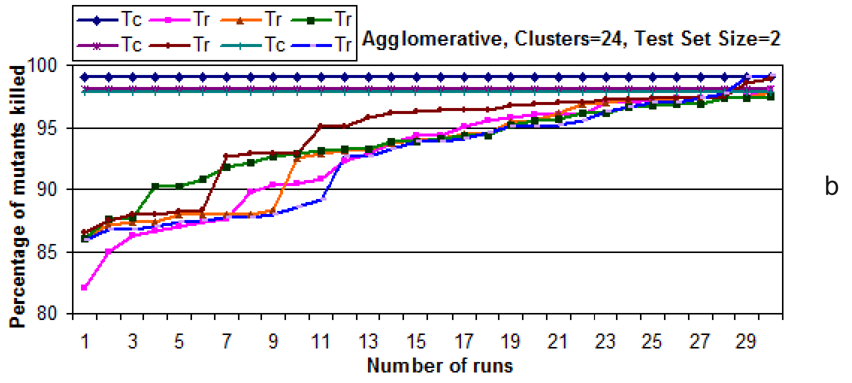


Figure 5.11: Agglomerative clustering algorithm result on Schedule2 program for k=83 and k=24

5.3.4 Results for the Totinfo Program

The efficiency results of *Tc* for both *k*-means algorithm and agglomerative algorithm are given in figures 5.12 and 5.13. The optimal value of the number of clusters is 173 for both of the algorithms. The MS of *Tc* in figure 5.12 *a* and figure 5.13 *a* is 100%. It is also eveident from the graphs *b* of the figures 5.12 and 5.13 that the MS of the test set *Tc* has reduced with reducing the number of mutants.

The test cases are reduced to 6 and 7, by *k*-means and agglomerative algorithms respectively, from a total size of 1052. Even though the test cases are reduced, but these test cases achieve strength by achieving a mutation score of 100%.

5.3.5 Results for the Printtokens Program

The MS of *Tc* for *k*-means and agglomerative algorithm is shown in figures 5.14 and 5.15. The *a* graphs in both the figures represent a MS of 100% for *Tc*. However, the MS for *Tc* is reduced in graphs *b*, although the reduction is still above 95%. Additionally, the test set size of 2021 is reduced to 4.

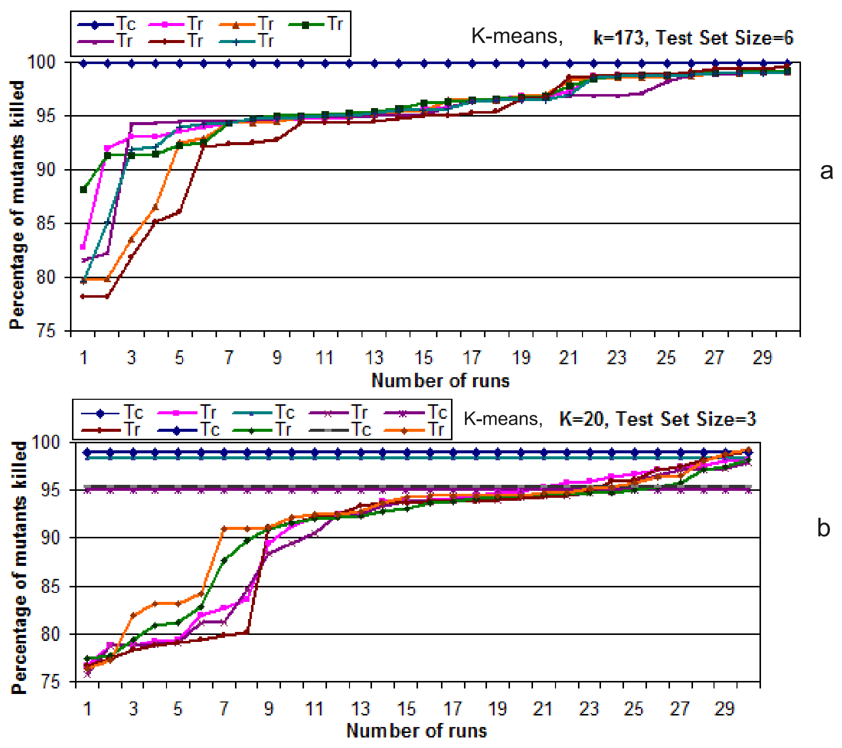


Figure 5.12: *K*-means clustering algorithm result on Totinfo program for k=173 and k=20

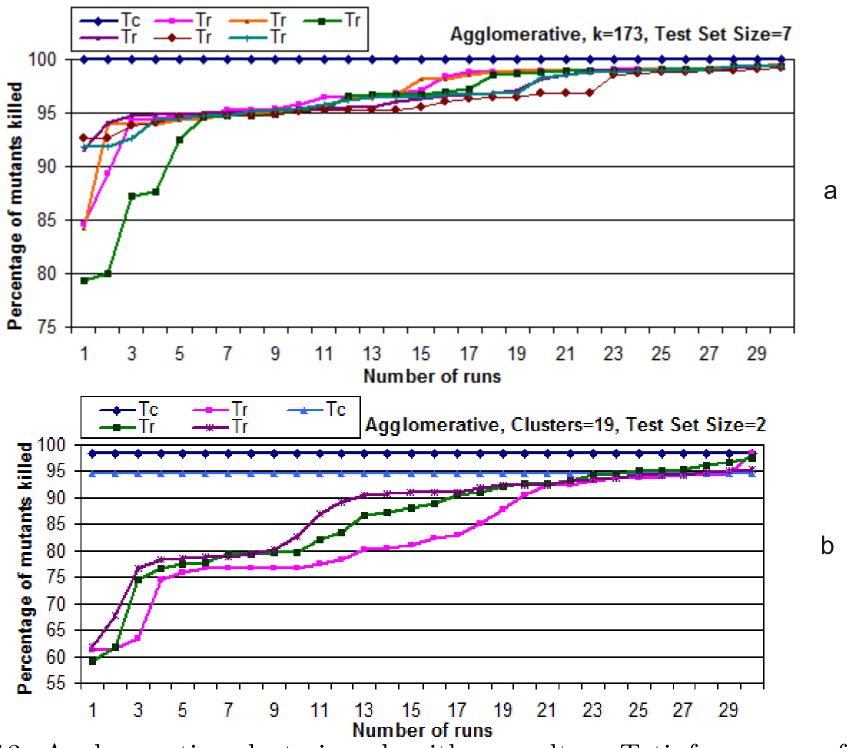


Figure 5.13: Agglomerative clustering algorithm result on Totinfo program for clusters= 173 and 19

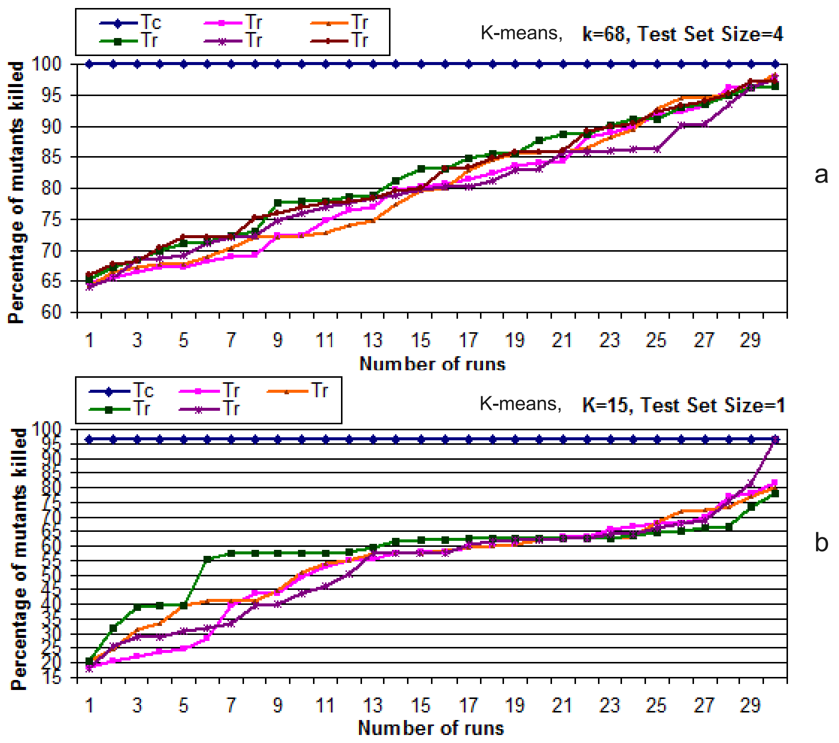


Figure 5.14: *K*-means clustering algorithm result on Printtokens for k= 68 and k=15

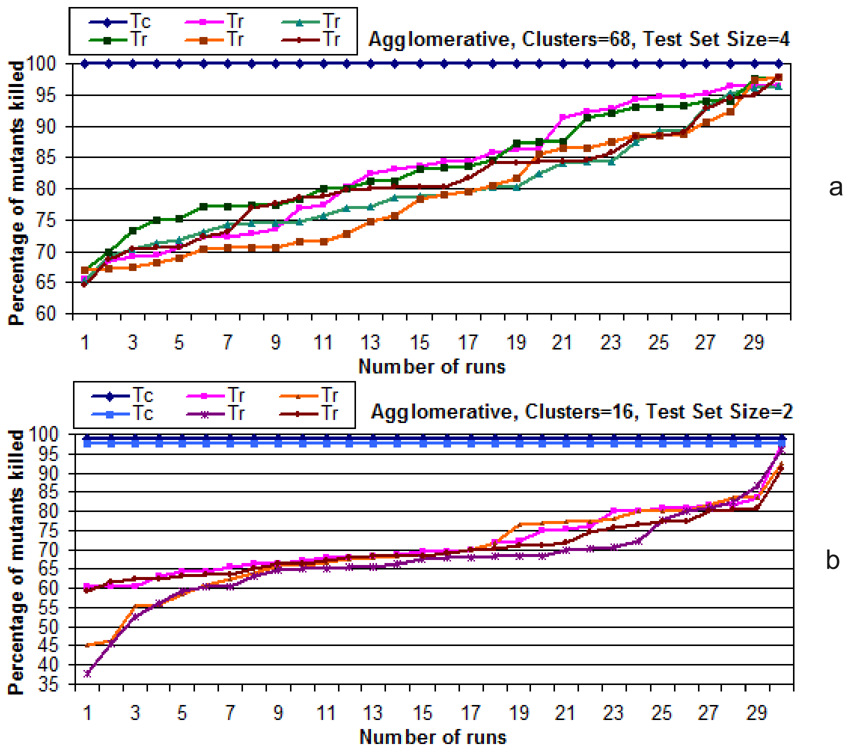


Figure 5.15: Agglomerative clustering algorithm result on Printtokens for clusters= 68 and 16

5.4 Results and Findings

The evaluation results shows that the test set from the clusters *Tc* are efficient in killing the mutants than the random test set *Tr*, this is particularly true for the TCAS, Schedule2, Totinfo and Printtokens. The *Tc* percentage of killing the mutant is always close to being 100% for the optimal cluster values of these programs, whereas the killing percentage of *Tr* starts off by killing somewhere around 80% of mutants. However, it is also significant to mention that by decreasing the number of the clusters, the efficiency of the *Tc* is greatly affected. Such that, in some cases it becomes below 95%. The MS below 95% is not desirable.

Program	Total # of mutants	Optimal # of clusters, reduced mutants	Total # of test cases	Reduced test cases	Average Mutation Score
Triangle	584	64	34	13	98.9%
TCAS	856	200	1608	19	99.3%
Schedule2	972	83	1399	2	100%
Totinfo	1879	173	1052	6	100%
Printtokens	506	68	2021	4	100%

Table 5.1: Summarized results of k-means algorithm on all five programs

Another important point worth mentioning is that the clustering works better on complex programs, as the evaluation shows that a MS of 100% is achieved in case of TCAS, Schedule2, Totinfo and Printtokens programs. With a considerably much smaller and less complex program like triangle program, the MS is never 100% for *Tc*.

Program	Total # of mutants	Optimal # of clusters, reduced mutants	Total # of test cases	Reduced test cases	Average Mutation Score
Triangle	584	62	34	12	95.6%
TCAS	856	212	1608	21	100%
Schedule2	972	83	1399	2	100%
Totinfo	1879	173	1052	7	100%
Printtokens	506	68	2021	4	100%

Table 5.2: Summarized results of agglomerative algorithm on all five programs

The evaluation results indicate that there are certain optimal values of the number of clusters, at which a mutation adequate test set is achieved. These values are optimal because of a number of reasons. Firstly, because these reduce the number of mutants, secondly a test set is produced with a 100% MS, thirdly, the size of this mutation adequate test set is less than the actual size of the test set. Hence, in addition to reducing the number of the clusters, the number of the test cases are also reduced. See tables 5.1 and 5.2, for a complete overview of the optimal values of the clusters and their effect on the mutants, test cases and their adequacy. The tables 5.1 and 5.2 summarizes our results and findings from the evaluation for both k-means and agglomerative algorithms.

Chapter 6

Conclusion

This section of the report elaborates the conclusion drawn from the empirical study conducted in this project.

This paper defines a novel approach to overcome one of the problems of mutation testing, i.e., large number of mutants. It is the originality of the research that makes it a significant contribution. Although work is done in this area previously, but that does not provide an overall optimal solution, as discussed in chapter 3.

Empirical results reveal that in spite of decreasing the large number of the mutants, the power of the set of the mutants is not reduced, which means the strength of the test set is not minimized. We get 100% mutation score for the test set generated from the clusters, this is what we hypothesized.

The empirical study was conducted on five different programs of different lengths and varying number of mutants and test cases. We have based our results and findings on the results of all the five programs. Here the size and complexity of the program is arguable. The test set Tc from large and complex programs, like TCAS, Schedule2, Totinfo and Printtokens, tend to do extremely well by achieving a mutation score of 100%, with an exception of TCAS clustered by k-means algorithm for which the MS is 99.3% Whereas, for small programs, like Triangle program of 50 LOC, the test set Tc never achieve a mutation score of 100% although it is close and below 95%. According to Offutt in [9], a test set with a mutation score below 95% is not effective. This leads us to say that this approach of mutation clustering, works extremely well for large and complex programs and better for small programs.

Another thing worth mentioning here is that by clustering the size of the mutants is reduced, but along with that the size of the test set is also minimized, see tables 5.1 and 5.2. Although the test set is minimized, yet it is strong and efficient enough to kill all the mutants, thereby achieving a mutation score of 100%. For example, in case of Schedule2 program, the k-means algorithm reduced the mutants to 83 from a total size of 972, consequently, the test cases are reduced to 2 from a test set of 1399.

We consider this project a great contribution in the particular research area. The reason for this is that it has provided a novel way to deal with the problem of mutation testing. This claim is backed up by the positive results from the evaluation.

Chapter 7

Future Enhancement

Due to the time constraint, only five programs were used. Which were evaluated to backup our results. However, the empirical study can be broadened by including and evaluating more programs. It would be interesting to see the results based on larger set of programs.

Furthermore, this project can be extended by implementing some other clustering algorithms, apart from k-means and agglomerative clustering techniques. The other possible implementation of algorithms could be divisive clustering algorithm. This would be an interesting enhancement to see how this technique works with other algorithms.

Bibliography

[1] A. J. Offutt, A Practical System for Mutation Testing:Help for the Common Programmer, Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years, Year of Publication:1994, Pages: 824-830, ISBN: 0-7803-2103-0

[2] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, A. Y. Wu, An Efficient k-Means Clustering Algorithm: Analysis and Implementation, IEEE Transactions on Pattern Analysis and Machine Intelligence 24(7), 2002,Pages:881-892.

[3] A. K. Jain, M. N. Murty, P. J. Flynn, Data Clustering: A Review, ACM Computing Surveys (CSUR), Volume 31 , Issue 3 (September 1999), Year of Publication:1999, Pages: 264-323, ISSN:0360-0300

[4] G. Fung. A Comprehensive Overview of Basic Clustering Algorithms, June 22, 2001

[5] M. J. Harrold. Testing: A Roadmap, College of Computing, In Future of Software Engineering, 22nd International Conference on Software Engineering, June 2000, Pages: 61-72.

[6] A. J. Offutt, R. H. Untch. Mutation 2000: Uniting the Orthogonal, Kluwer International Series On Advances In Database Systems, Mutation testing for the new century, Section: Mutation: cost reduction, Year of Publication:2001, Pages: 34-44

[7] A. P. Mathur, W. E. Wong, Reducing the cost of mutation testing: an empirical study, Journal of Systems and Software, Volume 31 , Issue 3 (December 1995), Pages: 185-196, Year of Publication:1995, ISSN:0164-1212

[8] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, Constrained mutation in C programs, In Proceedings o/ VIII Symposium on Software Engineering, Oct 1994. in Curitiba, Brazil.

[9] A. J. Offutt, G. Rothermel, C. Zapf, An experimental determination of sufficient mutation operators, ACM Transactions on Software Engineering and Methodology, vol. 5 (2), Pages: 99-118, 1996.

[10] W. E. Wong. On mutation and data flow, Phd Thesis, 1993

[11] T. A. Budd. Mutation analysis of program test data. Ph.D. thesis, Yale Univ., NewHaven CT, 1980.

[12] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In Proceedings of the 15th Annual International Computer Software and Applications Conference ,Tokyo, Japan, Sept. 1991, Pages: 604-605.

[13] A. T. Acree. On mutation. Phd thesis, Georgia Institute of Technology, Atlanta Georgia 1980, Technical Report GIT ICS 80/12.

[14] V. Faber. Clustering and the Continuous k-means Algorithm. Los Alamos Science, vol. 22, 1994, Pages: 138-144.

[15] M. R. Anderberg. Cluster Analysis for Applications. Academic Press, Inc., New York, NY, 1973.

[16] K. Daniels. Learning the Threshold in Hierarchical Agglomerative Clustering, Proceedings of the 5th International Conference on Machine Learning and Applications, Year of Publication:2006, Pages: 270-278, ISBN:0-7695-2735-3

[17] K. Adamopoulos, M. Harman, R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. Proc. Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103, Springer, Berlin Heidelberg New York, Pages: 1338-1349.

[18] A. J. Offutt, G. Rothermel, C. Zapf. An experimental evaluation of selective mutation. Proceedings of the Fifteenth International Conference on Software Engineering, IEEE Computer Society Press: Baltimore, Maryland, 1993; Pages: 100-107.

[19] Y. JiaandM. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing, 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08) Beijing, China, 28th-29th September 2008. To appear.

[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C.Stein. Introduction to Algorithms, Second Edition. The MIT Press, September 2001.

[21] Y. Jia and M. Harman. Milu: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language, TAIC PART'08, Windsor, UK, 29th-31st August 2008.

[22] K. Teknomo. Similarity Measurement, http:\\people.revoledu.com\kardi\tutorial\Similarity

Appendix A

Mutants data for Triangle program

- 584 mutants and a test set of 34 test cases.

```
0 0100111000101100000000001111100010
1 0100111000101100000000001111100010
2 0000000000000000000000000000000010
3 1111111111111111111111111111111111
4 1111111111111111111111111111111111
5 0100111000101100000000001111100010
6 0000000000000000000000000000000010
7 0100111000101100000000001111100010
8 0100111000101100000000001111100010
9 0000000000000000000000000000000010
10 0100111000101100000000001111100010
11 0100111000101100000000001111100000
12 0000000000000000000000000000000000
13 0100111000101100000000001111100010
14 0100111000101100000000001111100010
15 0100111000101100000000001111100010
16 0000010000000100000000000001000010
17 0100111000101100000000001111100000
18 0100111000101100000000001111100010
19 0100111000101100000000001111100010
20 0100111000101100000000001111100010
21 0000000000000000000000000000000000
22 0000000000000000000000000000000010
23 0000000000000000000000000000000000
24 0000000000000000000000000000000000
25 0000000000000000000000000000000010
26 0100111000101100000000001111100000
27 0000000000000000000000000000000010
28 0100111000101100000000001111100000
29 0100111000101100000000001111100010
30 0000000000000000000000000000000000
31 0100111000101100000000001111100010
32 0100111000101100000000001111100010
33 0100111000101100000000001111100000
34 0100111000101100000000001111100000
35 0000000000000000000000000000000000
36 0111111111101100001111111111111101
37 0111111111101100001111111111111101
38 0100111000101100000000001111100000
39 0000000000000000000000000000000000
40 0100111000101100000000001111100000
41 0100111000101100000000001111100000
42 0000000000000000000000000000000000
43 0100111000101100000000001111100000
44 0100111000101100000000001111100000
45 0000000000000000000000000000000000
46 0100111000101100000000001111100000
47 0100111000101100000000001111100000
48 0100111000101100000000001111100000
49 0000000000001000000000001010000000
50 0100111000101100000000001111100000
51 0100111000101100000000001111100000
52 0100111000101100000000001111100000
53 0100111000101100000000001111100000
54 0000000000000000000000000000000000
55 0000000000000000000000000000000000
56 0000000000000000000000000000000000
57 0000000000000000000000000000000000
58 0000000000000000000000000000000000
59 0100111000101100000000001111100000
60 0000000000000000000000000000000000
61 0100111000101100000000001111100000
62 0100111000101100000000001111100000
63 0000000000000000000000000000000000
64 0100111000101100000000001111100000
65 0100111000101100000000001111100000
66 0100111000101100000000001111100000
67 0100111000101100000000001111100000
68 0000000000000000000000000000000000
69 0101111111101100000001111111111101
70 0101111111101100000001111111111101
71 0100111000101100000000001111100000
72 0000000000000000000000000000000000
73 0100111000101100000000001111100000
74 0100111000101100000000001111100000
75 0000000000000000000000000000000000
76 0100111000101100000000001111100000
77 0100111000101100000000001111100000
78 0000000000000000000000000000000000
79 0100111000101100000000001111100000
80 0100111000101100000000001111100000
81 0100111000101100000000001111100000
82 0100111111001100000001110111111101
83 0100111111001100000001110111111101
84 0000000000000000000000000000000000
85 0100111111101100000001111111111101
86 0100111111101100000001111111111101
87 0100111111001100000001110111111101
88 0000000000000000000000000000000000
89 0100111111001100000001110111111101
```

```
90 0100111111001100000001111011111101
91 0100111111001100000001111011111101
92 0100001011001100000000111000101101
93 0100101011001100000000111010101101
94 0100001011001100000000111000101101
95 0000000001000100000000010000000101
96 0100101010001100000000011010101000
97 0100001011001100000000111000101101
98 0000000011001100000000110000001100
99 0100101011001100000000111010101101
100 0100001011001100000000111000101101
101 0100101010001000000000011010101000
102 0000100001000100000000100010000101
103 0100001010001000000000011000101000
104 0000000001000100000000010000000101
105 0100101011001100000000111010101101
106 0100001011001100000000111000101101
107 0000100000000000000000100010000101
108 0000000000000000000000000000000000
109 0000100000000000000000000010000000
110 0000100000000000000000000010000000
111 0000100000000000000000000010000000
112 0000100000000000000000000010000000
113 0000000000000000000000000000000000
114 0000100000000000000000000010000000
115 0000000000000000000000000000000000
116 0000100000000000000000000010000000
117 0000100000000000000000000010000000
118 0000100000000000000000000010000000
119 0000100000000000000000000010000000
120 0000100000000000000000000010000000
121 0000100000000000000000000010000000
122 0000000000000000000000000000000000
123 0000100000000000000000000010000000
124 0000000000000000000000000000000000
125 0000100000000000000000000010000000
126 0000000000000000000000000000000000
127 0000000000000000000000000000000000
128 0000100000000000000000000010000000
129 0000000000000000000000000000000000
130 0000100000000000000000000010000000
131 0000100000000000000000000010000000
132 0000000000000000000000000000000000
133 0000100000000000000000000010000000
134 0000100000000000000000000010000000
135 0100001101001100000001101000110101
136 0100011101001100000001101001110101
137 0100001101001100000001101000110101
138 0000000001000100000000010000000101
139 0100011101001100000001001001110000
140 0100001101001100000001101000110101
141 0100000001000100000000101000010100
142 0100011101001100000001101001110101
143 0100001101001100000001101000110101
144 0100011100001000000001001001110000
145 0000010001000100000000100001000101
146 0100001100001000000001001000110000
147 0000000001000100000000010000000101
148 0100011101001100000001101001110101
149 0100001101001100000001101000110101
150 0000010000000000000000000001000101
151 0000000000000000000000000000000000
152 0000010000100000000000000101000000
153 0000010000000000000000000001000000
154 0000010000100000000000000101000000
155 0000010000000000000000000001000000
156 0000000000000000000000000000000000
157 0000010000000000000000000001000000
158 0000000000000000000000000000000000
159 0000010000000000000000000001000000
160 0000010000100000000000000101000000
161 0000010000100000000000000101000000
162 0000010000000000000000000001000000
163 0000010000100000000000000101000000
164 0000010000000000000000000001000000
165 0000000000000000000000000000000000
166 0000010000100000000000000101000000
167 0000000000000000000000000000000000
168 0000010000000000000000000001000000
169 0000000000000000000000000000001000
170 0000000000000000000000000000001000
171 0000010000100000000000000101000000
172 0000000000000000000000000000001000
173 0000010000100000000000000101000000
174 0000010000100000000000000101000000
175 0000000000000000000000000000001000
176 0000010000000000000000000001000000
177 0000010000100000000000000101000000
178 0100110110001100000001011011011000
179 0100111101011000000001011111111001
180 0100110110001100000001011011011000
181 0000000010001000000000011010001000
182 0100111101011000000001011101110001
183 0100110110001100000001011011011000
184 0100110010001000000000010000011000
185 0100111101011000000001011111111001
186 0100110110001100000001011011011000
187 0100111001001000000001000101110001
188 0000001010101000000000011110101001
189 0100110100000100000001000001010000
190 0000000010001000000000011010001000
191 0100111101011000000001011111111001
192 0100110110001100000001011011011000
193 0000001000100000000000000110101001
194 0000000000000000000000000000000000
195 0000001000100000000000000100100000
196 0000001000000000000000000000100001
197 0000001000100000000000000100100001
198 0000001000100000000000000100100001
199 0000000000000000000000000000000000
200 0000001000100000000000000100100001
201 0000000000000000000000000000000000
202 0000001000000000000000000000100001
203 0000001000100000000000000100100001
204 0000001000100000000000000100100000
205 0000001000000000000000000000100001
206 0000001000100000000000000100100001
207 0000001000100000000000000100100001
208 0000000000100000000000000100000000
209 0000001000100000000000000100100001
210 0000000000100000000000000100000000
211 0000001000100000000000000100100001
212 0000000000100000000000000100000101
213 0000000000100000000000000100000101
214 0000001000100000000000000100100001
215 0000000000100000000000000100000101
216 0000001000100000000000000100100001
217 0000001000100000000000000100100001
218 0000000000100000000000000100000101
219 0000001000000000000000000000100001
220 0000001000100000000000000100100001
221 0100111000101100000000001111100001
```

222	0100111000101100000000001111100001
223	0100000000001100000000001000000000
224	0100111111101100000001111111111101
225	0100111111101100000001111111111101
226	0100111000101100000000001111100001
227	0100000000001100000000001000000000
228	0100111000101100000000001111100001
229	0100111000101100000000001111100001
230	0100000000001100000000001000000000
231	0100111000101100000000001111100001
232	0000000000000000000000000000000000
233	0000111000100000000000000111100001
234	0100111000101100000000001111100001
235	0100111000101100000000001111100001
236	0100111000101100000000001111100001
237	0100000000001100000000001000000000
238	0000000100000000000000000000000000
239	0100000000001100000000001000000000
240	0100000000001100000000001000000000
241	0100000000001100000000001000000000
242	0100000000000100000000000000000000
243	0100000000001100000001001000000000
244	0100000000000100000000000000000000
245	0100000000001100000000001000000000
246	0100000000001100000000001000000000
247	0100000000001100000000001000000000
248	0100000000001100000000001000000000
249	0100000000001100000000001000000000
250	0000000100000000000000000000000000
251	0100000000001100000001001000000000
252	0000000100000000000001000000000000
253	0100000000001100000000001000000000
254	0100000000001100000000001000000000
255	0100000000001100000000001000000000
256	0100000000001100000000001000000000
257	0100000000001100000000001000000000
258	0100000000001100000000001000000000
259	0100000000001100000000001000000000
260	0100000100000000000001001000000000
261	0100000000001100000000001000000000
262	0100000000001100000000001000000000
263	0100000100001100000001001000000000
264	0000000000000000000000000000000000
265	0100000100001100000001001000000000
266	0000000100000000000001000000000000
267	0100000000001100000000001000000000
268	0100000000001100000000001000000000
269	0000000100000000000001000000000000
270	0100000000001100000001001000000000
271	0100000000001100000001001000000000
272	0100000000001100000001001000000000
273	0100000000001100000001001000000000
274	0100000000001100000001001000000000
275	0000000000000000000000000000000000
276	0000000101000000000001100000000000
277	0000000000000000000000000000000000
278	0000000000000000000000000000000000
279	0100000000001100000001001000000000
280	0100000000001100000000001000000000
281	0100000000001100000001001000000000
282	0100000000001100000000001000000000
283	0100000101001100000001101000000000
284	0000000000000000000000000000000000
285	0100000000001100000001001000000000
286	0100000000001100000000001000000000
287	0100000000001100000000001000000000
288	0000000001000000000000100000000000
289	0100000000001100000000001000000000
290	0000000000001100000000001000000000
291	0100000000001100000000001000000000
292	0000000001000100000000100000000000
293	0100000000001100000000001000000000
294	0000000000000100000000000000000000
295	0100000000001100000000001000000000
296	0100000000001100000000001000000000
297	0100000000001100000000001000000000
298	0100000000001100000000001000000000
299	0100000000001100000000001000000000
300	0000000001000000000000100000000000
301	0100000000001100000000001000000000
302	0000000001000000000000100000000000
303	0100000000001100000000001000000000
304	0100000000001100000000001000000000
305	0100000000001100000000001000000000
306	0100000000001100000000001000000000
307	0100000000001100000000001000000000
308	0100000000001100000000001000000000
309	0100000000001100000000001000000000
310	0100000000001100000000001000000000
311	0100000000001100000000001000000000
312	0100000000001100000000001000000000
313	0100000001001100000000101000000000
314	0000000000000000000000000000000000
315	0100000001001100000000101000000000
316	0000000001000000000000100000000000
317	0100000000001100000000001000000000
318	0100000000001100000000001000000000
319	0100000001001000000000101000000000
320	0100000000001100000000001000000000
321	0100000000001100000000001000000000
322	0100000000001100000000001000000000
323	0100000000001100000000001000000000
324	0100000000001100000000001000000000
325	0000000000000000000000000000000000
326	0000000011000000000000110000000000
327	0000000000000000000000000000000000
328	0000000000000000000000000000000000
329	0100000000001100000000001000000000
330	0100000000001100000000001000000000
331	0100000000001100000000001000000000
332	0100000000001100000000001000000000
333	0100000011001100000000111000000000
334	0000000000000000000000000000000000
335	0100000000001100000000001000000000
336	0100000000001100000000001000000000
337	0100000000001100000000001000000000
338	0000000010000000000000000000000000
339	0100000000001100000000001000000000
340	0100000000001100000000001000000000
341	0100000000001100000000001000000000
342	0000000000001000000000000000000000
343	0100000000001100000000011000000000
344	0000000000001000000000001000000000
345	0100000000001100000000001000000000
346	0100000000001100000000001000000000
347	0100000000001100000000001000000000
348	0100000000001100000000001000000000
349	0100000000001100000000001000000000
350	0000000010000000000000000000000000
351	0100000000001100000000011000000000
352	0000000010000000000000010000000000
353	0100000000001100000000001000000000

```
354 0100000000001100000000001000000000
355 0100000000001100000000001000000000
356 0100000000001100000000001000000000
357 0100000000001100000000001000000000
358 0100000000001100000000001000000000
359 0100000000001100000000001000000000
360 0000000000001100000000001000000000
361 0100000000001100000000001000000000
362 0100000000001100000000001000000000
363 0100000010001100000000011000000000
364 0000000000000000000000000000000000
365 0100000010001100000000011000000000
366 0000000010000000000000010000000000
367 0100000000001100000000001000000000
368 0100000000001100000000001000000000
369 0000000010000100000000010000000000
370 0000111000000000000000000011111101
371 0000110000000000000000000011011000
372 0000111000000000000000000011111101
373 0000001000000000000000000000100101
374 0000110000100000000000000111011000
375 0000111000000000000000000011111101
376 0000111000000000000000000011111101
377 0000110000000000000000000011011000
378 0000111000000000000000000011111101
379 0000110000100000000000000111011000
380 0000111000100000000000000111111101
381 0000001000000000000000000000100101
382 0000001000100000000000000100100101
383 0000110000000000000000000011011000
384 0000111000000000000000000011111101
385 0000000000100000000000000100000000
386 0000000000000000000000000000001101
387 0000100000000000000000000010001101
388 0000000000000000000000000000001101
389 0000000000000000000000000000001101
390 0000100000000000000000000010000000
391 0000000000000000000000000000001101
392 0000000000000000000000000000000101
393 0000100000000000000000000010001101
394 0000000000000000000000000000011101
395 0000100000000000000000000010000000
396 0000100000000000000000000010001101
397 0000000000000000000000000000000000
398 0000000000000000000000000000001101
399 0000100000000000000000000010001101
400 0000000000000000000000000000001101
401 0000100000000000000000000010001101
402 0000000000000000000000000000000000
403 0000100000000000000000000010000000
404 0000000000000000000000000000000000
405 0000100000000000000000000000000000
406 0000100000000000000000000000000000
407 0000000000000000000000000000011101
408 0000000000000000000000000000000000
409 0000100000000000000000000010011101
410 0000000000000000000000000000011101
411 0000100000000000000000000010000000
412 0000100000000000000000000010000000
413 0000100000000000000000000010000000
414 0000100000000000000000000010000000
415 0000000000000000000000000000000000
416 0000100000000000000000000010011101
417 0000000000000000000000000000010000
418 0000100000000000000000000010000000
419 0000100000000000000000000010000000
420 0000000000000000000000000000000000
421 0000100000000000000000000010000000
422 0000100000000000000000000010000000
423 0000000000000000000000000000010000
424 0000100000000000000000000010000000
425 0000000000000000000000000000010000
426 0000100000000000000000000000000000
427 0000000000000000000000000000011000
428 0000100000000000000000000010000000
429 0000100000000000000000000010000000
430 0000100000000000000000000010000000
431 0000100000000000000000000010000000
432 0000100000000000000000000010000000
433 0000000000000000000000000000010000
434 0000000000000000000000000000000000
435 0000100000000000000000000010000000
436 0000000000000000000000000000010000
437 0000000000000000000000000000010000
438 0000000000000000000000000000010000
439 0000000000000000000000000000010000
440 0000000000000000000000000000010000
441 0000000000000000000000000000010000
442 0000000000000000000000000010010000
443 0000000000000000000000000000010000
444 0000000000000000000000000000011101
445 0000100000000000000000000010010000
446 0000100000000000000000000010010000
447 0000000000000000000000000000000000
448 0000100000000000000000000010000000
449 0000000000000000000000000000010000
450 0000000000000000000000000000010000
451 0000100000000000000000000000000000
452 0000000000000000000000000000010101
453 0000010000000000000000000001010101
454 0000000000000000000000000000010101
455 0000000000000000000000000000000101
456 0000010000000000000000000001010101
457 0000000000000000000000000000010101
458 0000000000000000000000000000000101
459 0000010000000000000000000001010101
460 0000000000000000000000000000011101
461 0000010000000000000000000001010000
462 0000010000000000000000000001000101
463 0000000000000000000000000000010000
464 0000000000000000000000000000000101
465 0000010000000000000000000001010101
466 0000000000000000000000000000010101
467 0000010000000000000000000001000000
468 0000010000000000000000000001010000
469 0000010000000000000000000001010000
470 0000010000000000000000000001010000
471 0000010000000000000000000001010000
472 0000010000000000000000000001010000
473 0000000000000000000000000000011101
474 0000000000000000000000000000000000
475 0000010000000000000000000001011101
476 0000000000000000000000000000011101
477 0000010000000000000000000001000000
478 0000010000000000000000000001000000
479 0000010000000000000000000001000000
480 0000010000000000000000000001000000
481 0000000000000000000000000000000000
482 0000010000000000000000000001011101
483 0000010000000000000000000001010000
484 0000010000000000000000000001000000
485 0000010000000000000000000001000000
```

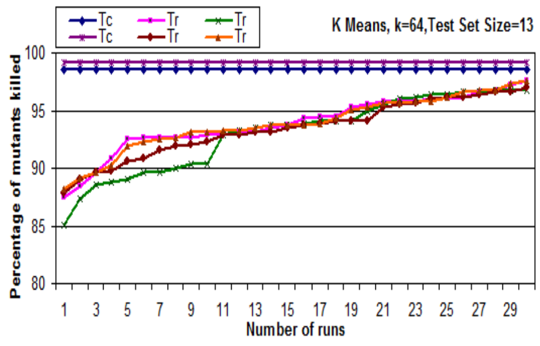
486	00000000000000000000000000000000000	519	00000010000000000000000000000111000	552	00000000000000000000000000000000000
487	00000100000000000000000000001000000	520	00000000000000000000000000000011000	553	00000010000000000000000000000100000
488	00000100000000000000000000001000000	521	00000000000000000000000000000000000	554	00000010000000000000000000000100000
489	00000000000000000000000000000001000	522	00000010000000000000000000000111000	555	00000000000000000000000000000000101
490	00000000000000000000000000001000000	523	00000000000000000000000000000011000	556	00000010000000000000000000000100000
491	00000000000000000000000000000001000	524	00000000000000000000000000000011000	557	00000000000000000000000000000000101
492	00000000000000000000000000000000000	525	00000010000000000000000000000111000	558	00000000000000000000000000000000000
493	00000000000000000000000000000011000	526	00000000000000000000000000000011101	559	00000000000000000000000000000010101
494	00000100000000000000000000001000000	527	00000010000000000000000000000111000	560	00000010000000000000000000000100000
495	00000100000000000000000000001000000	528	00000010000000000000000000000100000	561	00000010000000000000000000000100000
496	00000100000000000000000000001000000	529	00000000000000000000000000000011000	562	00000010000000000000000000000100000
497	00000100000000000000000000001000000	530	00000000000000000000000000000000000	563	00000010000000000000000000000100000
498	00000100000000000000000000001000000	531	00000010000000000000000000000111000	564	00000010000000000000000000000100000
499	00000000000000000000000000000001000	532	00000000000000000000000000000011000	565	00000000000000000000000000000000101
500	00000000000000000000000000000000000	533	00000010000000000000000000000100000	566	00000000000000000000000000000000001
501	00000100000000000000000000001000000	534	00000010000000000000000000000110000	567	00000010000000000000000000000100000
502	00000000000000000000000000000001000	535	00000010000000000000000000000110000	568	00000000000000000000000000000000101
503	00000000000000000000000000000001000	536	00000010000000000000000000000110000	569	00000000000000000000000000000000100
504	00000000000000000000000000000001000	537	00000010000000000000000000000110000	570	00000000000000000000000000000000101
505	00000000000000000000000000000001000	538	00000010000000000000000000000110000	571	00000000000000000000000000000000101
506	00000000000000000000000000000001000	539	00000000000000000000000000000011101	572	00000000000000000000000000000000101
507	00000000000000000000000000000001000	540	00000000000000000000000000000000000	573	00000000000000000000000000000000101
508	00000000000000000000000000001001000	541	00000010000000000000000000000111101	574	00000010000000000000000000000100101
509	00000000000000000000000000000001000	542	00000000000000000000000000000011101	575	00000000000000000000000000000000100
510	00000000000000000000000000000011101	543	00000010000000000000000000000100000	576	00000000000000000000000000000011101
511	00000100000000000000000000001001000	544	00000010000000000000000000000100000	577	00000010000000000000000000000100100
512	00000100000000000000000000001001000	545	00000010000000000000000000000100000	578	00000010000000000000000000000100101
513	00000000000000000000000000000000000	546	00000010000000000000000000000100000	579	00000000000000000000000000000000001
514	00000100000000000000000000001000000	547	00000000000000000000000000000000000	580	00000010000000000000000000000100001
515	00000000000000000000000000000001000	548	00000010000000000000000000000111101	581	00000000000000000000000000000000100
516	00000000000000000000000000000001000	549	00000010000000000000000000000110000	582	00000000000000000000000000000000101
517	00000000000000000000000000000000000	550	00000010000000000000000000000100000	583	00000000000000000000000000000000000
518	00000000000000000000000000000011000	551	00000010000000000000000000000100000		

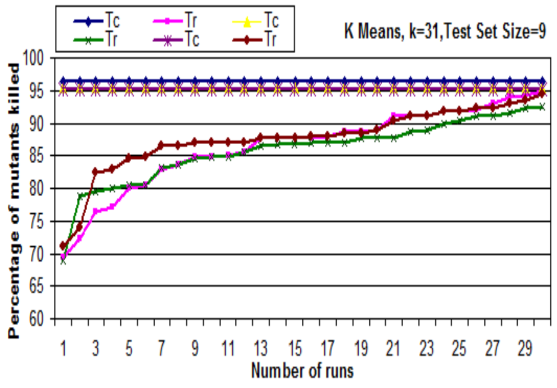
Appendix B

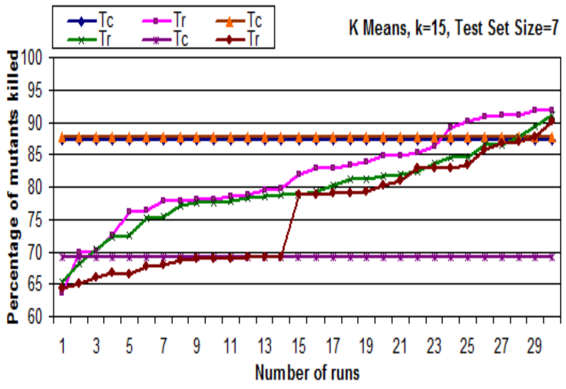
Experiment Results

Triangle Program

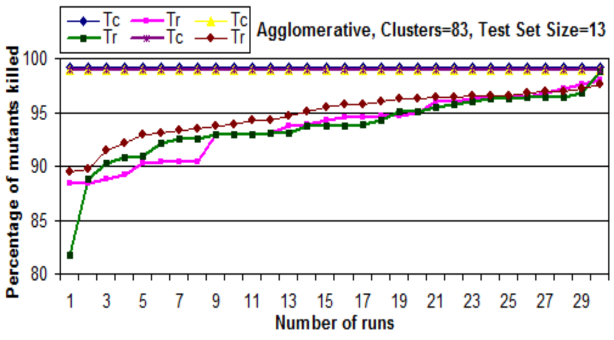
K-means clustering Algorithm

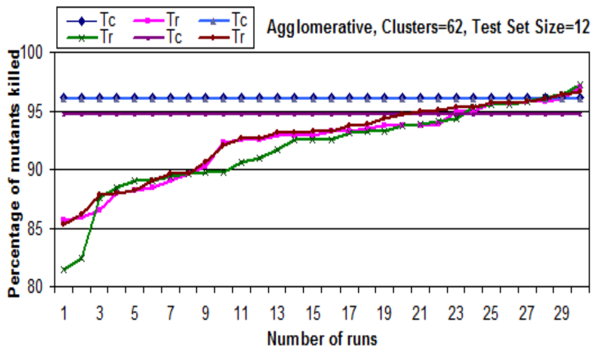


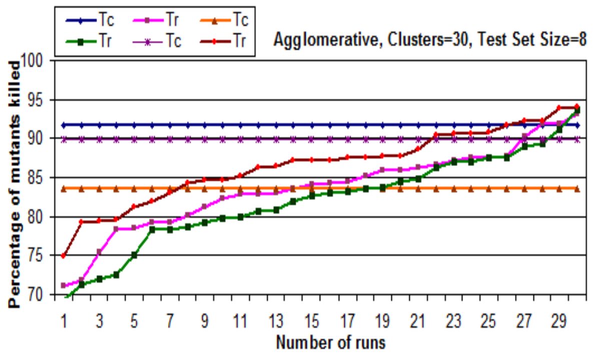




Agglomerative clustering algorithm

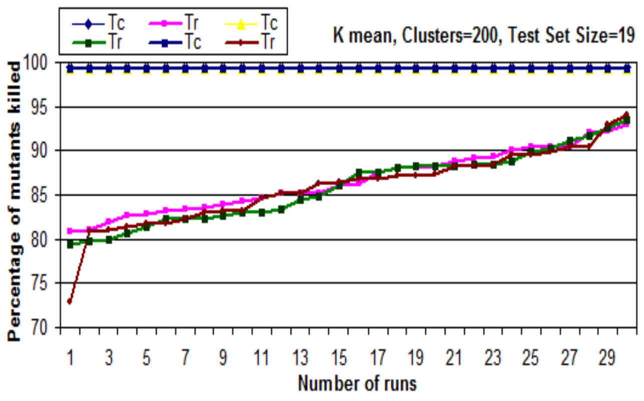


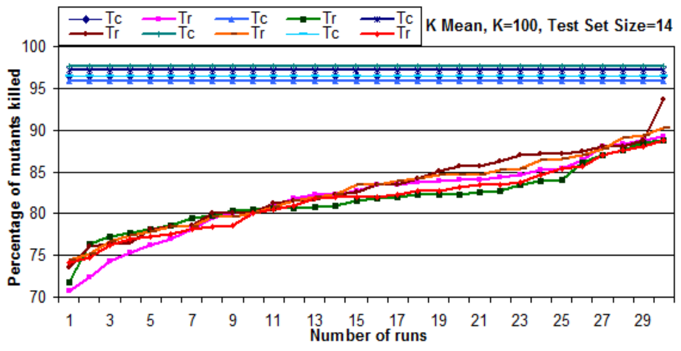


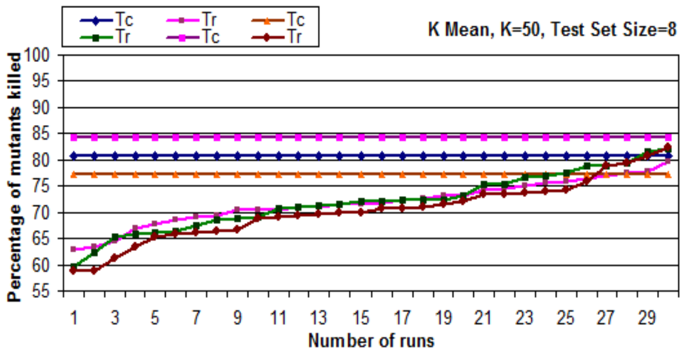


TCAS program

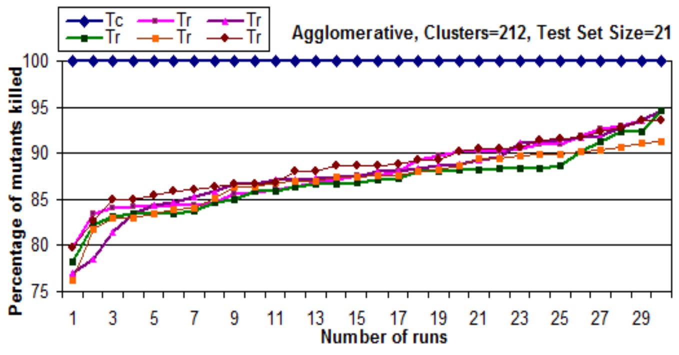
K-Means Clustering algorithm

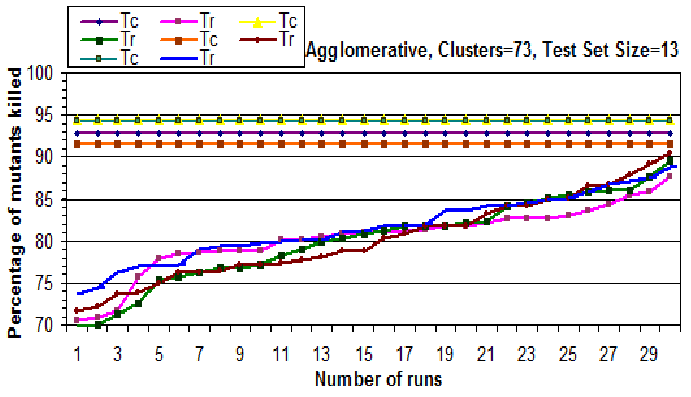


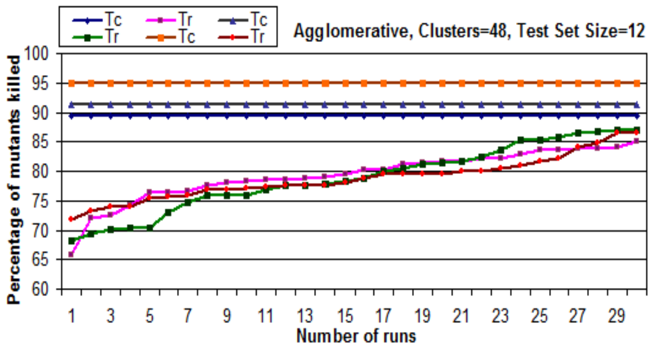




Agglomerative clustering algorithm



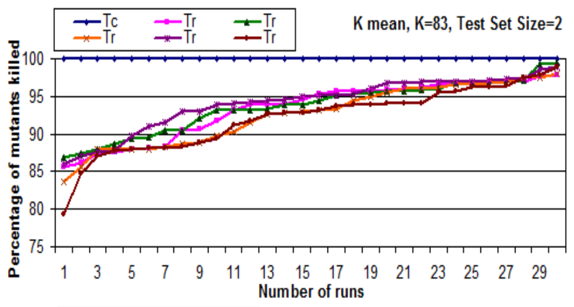


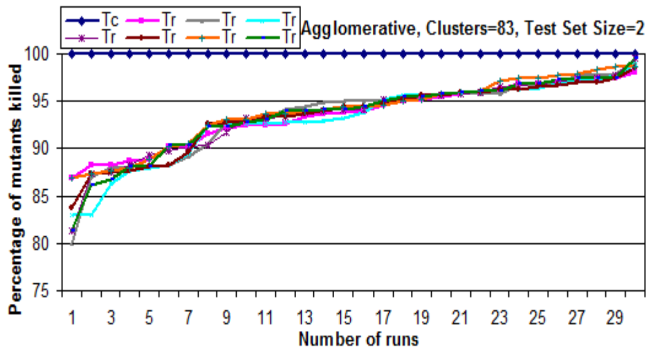


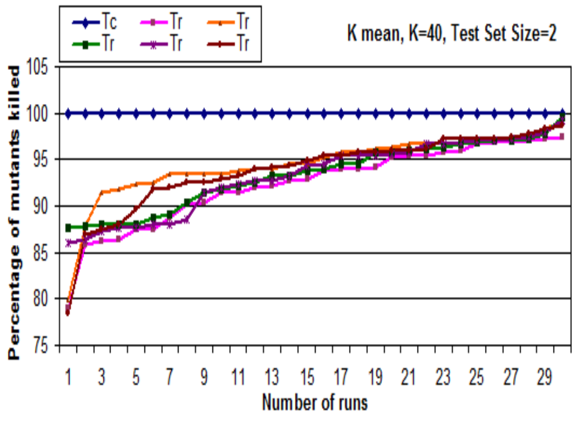
Schedule2 Program

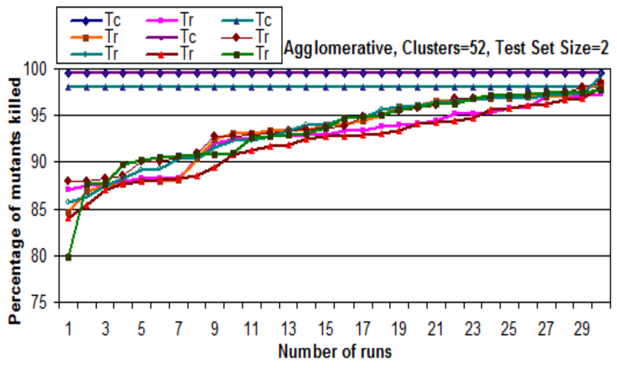
K-means clustering Algorithm

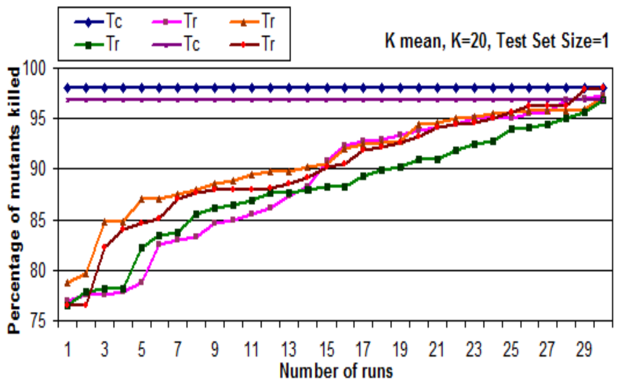
Agglomerative clustering algorithm

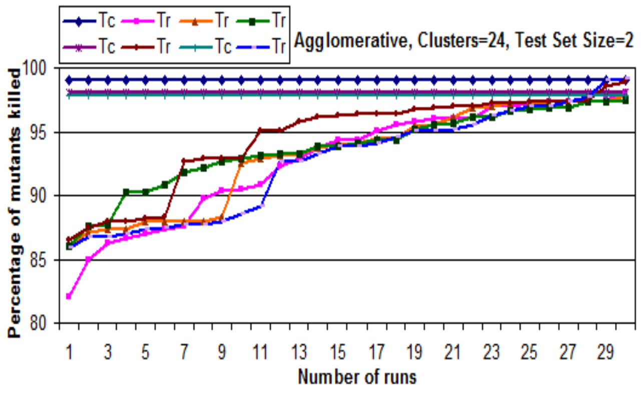






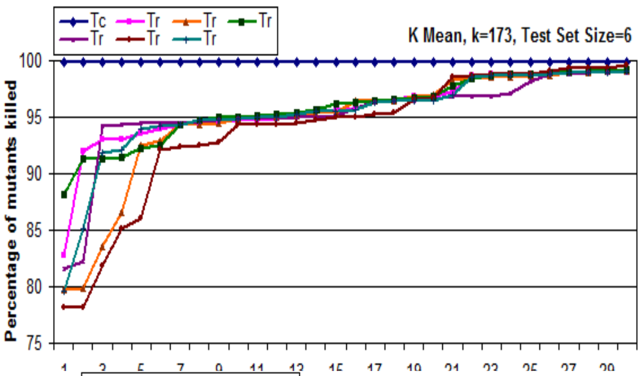


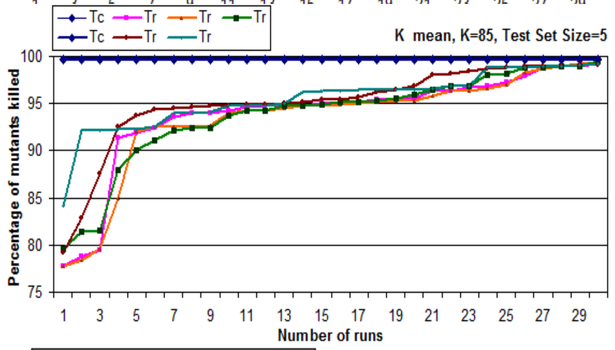


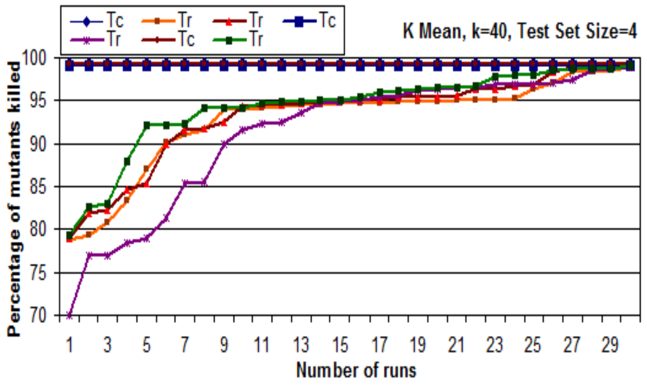


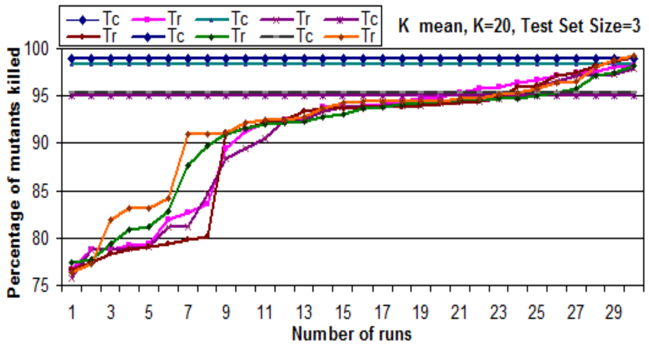
Totinfo Program

K-means clustering Algorithm

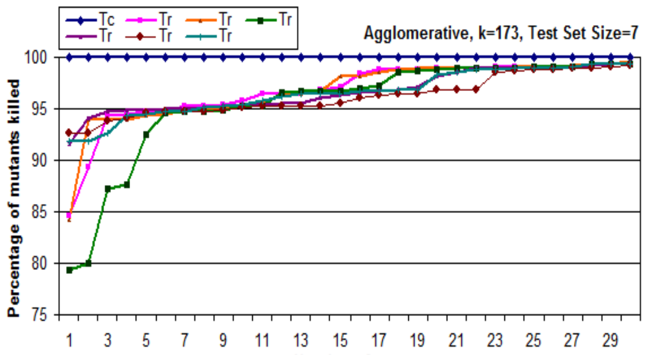


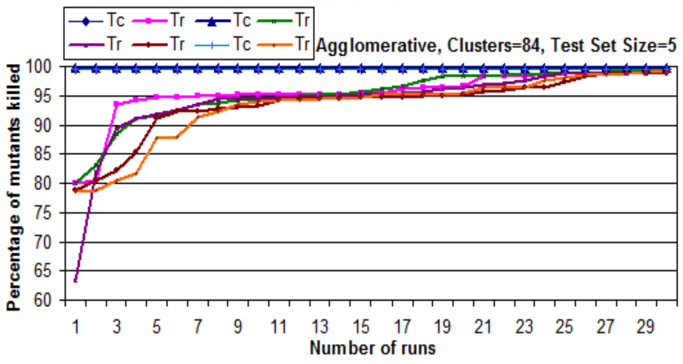


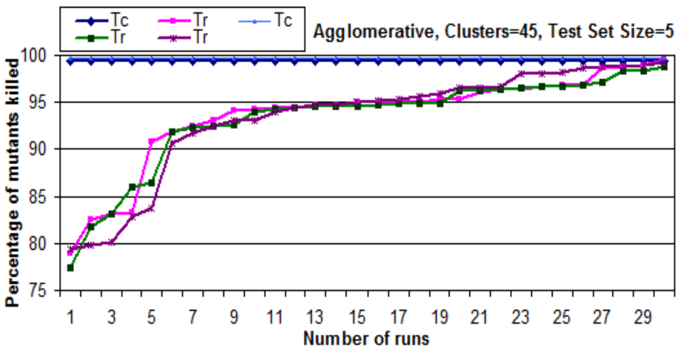


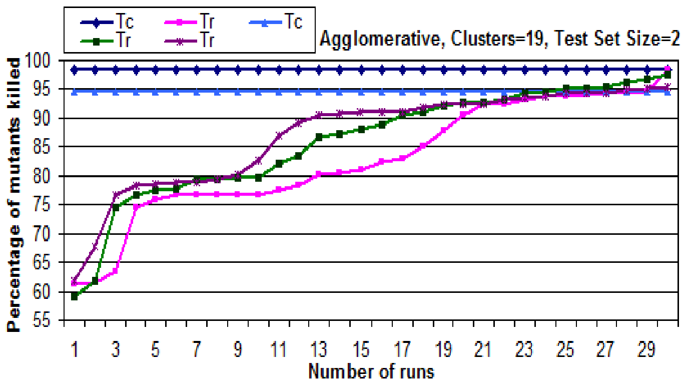


Agglomerative clustering algorithm



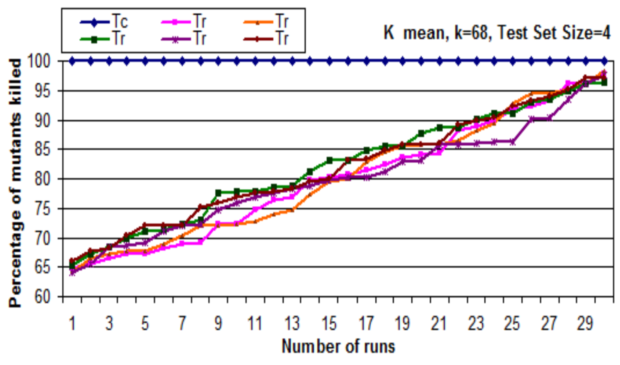


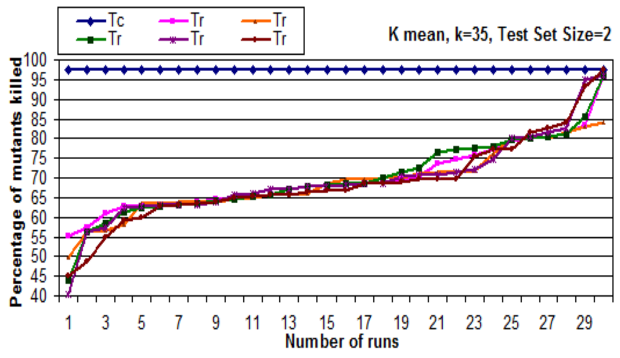


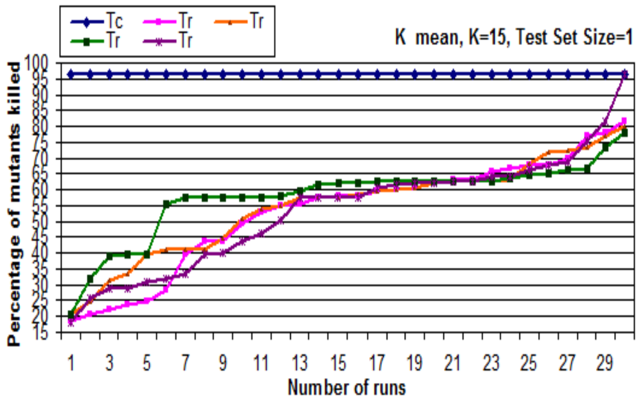


Printtokens Program

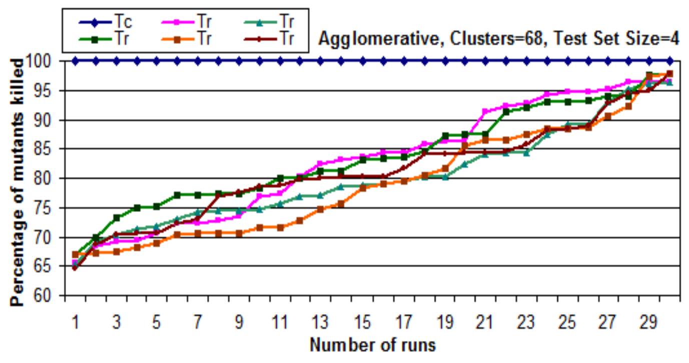
K-means clustering Algorithm

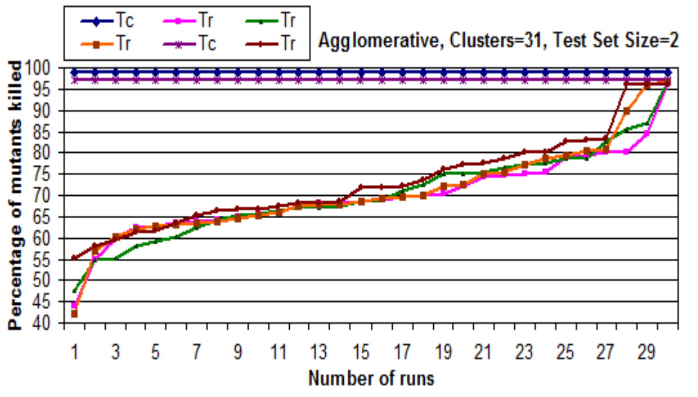


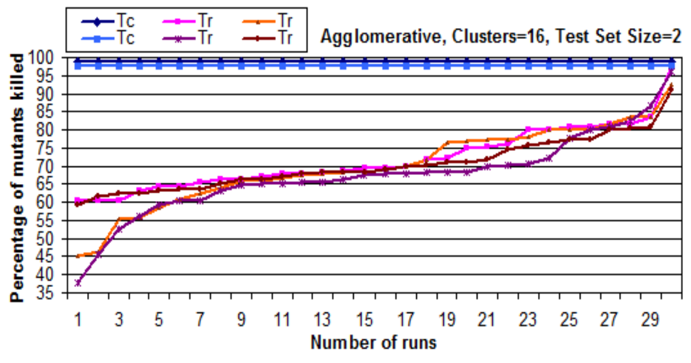




Agglomerative clustering algorithm







Appendix C

K-mean Clustering Class

The project is implemented in Visual Studio 2005 C#.Net.

This class has the following functions:

- ReadMutants(); Gets the list of the entire size of mutants.
- CalculateCentroid(): To calculate the centroids
- CalculateDistance(): To calculate the distance of each mutant to the centroid
- UpdateClusters(): Updates the clusters based on minimum distance
- WriteToFile(): Writes to file the clustered mutants

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.IO;

//////////////// K Mean Clustering Algorithm..////////////////////
/////// based on careful selection of initial  k points /////////

namespace GUI
{
    class KMeanAlgo
    {
        int NumberOfAlgoRuns=0;// specifies how many times we need to run the algo
        string[,] mutantsArray;
        int BitSize = 0;
        int NumberOfClusters;    // number of k
        int[,] newDistanceMatrix;
        int[,] oldDistanceMatrix;
        int[,] InitialDistance;
        float[,] HammingDistanceArray;
        float[,] Centroids;
        int CentroidIteration = 0;
        int NotEqualFlag = 0;
        int[] AddBitsArray; // used for calculating the centroid
        int[,] AllMutants;// this is the array that copies all the mutanst bit by bit
        Random rand = new Random(); // function for random numbers
// to avoid selecting same indexes for initial k values
        ArrayList NoRepetitionIndex = new ArrayList();
// saves different mutants for the initial K
        ArrayList NoRepititionMutant = new ArrayList();

//////////////////////////////////////////////////////
///// function that gets the mutants /////////////////

        public void ReadMutants(string[,] mutantsArray1, int ClusterNumber, int NumberOfAlgoRuns1)
        {
            NumberOfAlgoRuns = NumberOfAlgoRuns1;

            while (NumberOfAlgoRuns != 0) // runs the algorithm for specified number of times
            {
                NumberOfAlgoRuns = NumberOfAlgoRuns - 1;
                mutantsArray = new string[mutantsArray1.GetLength(0), mutantsArray1.GetLength(1)];
                Array.Copy(mutantsArray1, mutantsArray, mutantsArray1.Length);

                string mutantString = mutantsArray[1, 1];
// gives the length or number of bits of the mutant
                BitSize = mutantString.Length;

                // to get the size of the number of clusters by a loop
                NumberOfClusters = 0;
                for (int kNumber = 0; kNumber < ClusterNumber; kNumber++)
                {
                    NumberOfClusters = NumberOfClusters + 1;
                }
 // saves the smallest distance as 1's and rest as 0's
                newDistanceMatrix = new int[mutantsArray.GetLength(0), NumberOfClusters];
                oldDistanceMatrix = new int[mutantsArray.GetLength(0), NumberOfClusters];
                InitialDistance = new int[NumberOfClusters, mutantsArray.GetLength(0)];

                HammingDistanceArray = new float[ mutantsArray.GetLength(0), NumberOfClusters];
// 2-d array to, for k number of clusters and for the size of the mutant bits
                Centroids = new float[ClusterNumber, BitSize];

                AllMutants= new int[mutantsArray.GetLength(0), BitSize];

/////////////////////////////////////////////////////////////////////////
                //////// saves all the mutants bit by bit in the array AllMutants//////

                for (int rows = 0; rows < mutantsArray.GetLength(0); rows++)
                {
                    mutantString= mutantsArray[rows,1];

                    for (int col = 0; col < mutantString.Length; col++)
                    {
                        if (System.Convert.ToInt32(mutantString[col]) == 49)
                        {
                            AllMutants[rows, col] = 1;
                        }

                        else if (System.Convert.ToInt32(mutantString[col]) == 48)
                        {
                            AllMutants[rows, col] = 0;
                        }
                    }
                }
                Calculate_Centroid();
            }// ends while of the number of runs
        }
/////////////////////////////////////////////////////////////////////
//////////// FUNCTION TO CALCULATE THE CENTROID ////////////////////
///////////////////////////////////////////////////////////////////

public void Calculate_Centroid()
{
        int CentroidsSelected=0;
CentroidIteration = CentroidIteration + 1;
string MutantString;
       if (CentroidIteration == 1)    //initial K selection
       {
           int selectInitialCentroids;
           int select = 0;

          while (CentroidsSelected<NumberOfClusters)
          {
              selectInitialCentroids= rand.Next(mutantsArray.GetLength(0)-1);
              if (NoRepetitionIndex.Count > 0)
              {
                   MutantString= mutantsArray[selectInitialCentroids,1];

                  // this loop iterates and checks that the mutant has not already been selected of initial K
                  // it prevents repitition of mutant selection
                  for (int check = 0; check < NoRepetitionIndex.Count; check++)
                  {
                      if (System.Convert.ToInt32(NoRepetitionIndex[check]) != selectInitialCentroids &
      System.Convert.ToString(NoRepitionMutant[check]) != MutantString)
                      {
                            select = 1;
                      }
                      else
                      {
                            select = 0;
                            break;
                      }
                   }
              }
              else
              {
                 select = 1;
```

```
               }

               if (select == 1)
               {
                  NoRepetitionIndex.Add(selectInitialCentroids);
                  MutantString= mutantsArray[selectInitialCentroids,1];
                  NoRepititionMutant.Add( MutantString);
 for (int loop = 0; loop < MutantString.Length; loop++)
                 {
                    if (System.Convert.ToInt32(MutantString[loop]) == 49)
                    {
                        Centroids[CentroidsSelected, loop] = 1;
                    }
                 else if (System.Convert.ToInt32(MutantString[loop]) == 48)
                  {
                       Centroids[CentroidsSelected, loop] = 0;
                  }
                 }

                 CentroidsSelected = CentroidsSelected + 1;
                 select = 0;

              }
  }   // ends while
       }  // ends if

  else if (CentroidIteration > 1)   // more than 1 iteration
    {
        int countOnes = 0;
// iterates the columns, (cluster by cluster)
for (int distCol = 0; distCol < newDistanceMatrix.GetLength(1); distCol++)
        {
            countOnes = 0;
            AddBitsArray = new int[BitSize];
    // iterates the rows to find 1
           for (int distRow = 0; distRow < newDistanceMatrix.GetLength(0); distRow++)
           {
              if (newDistanceMatrix[distRow, distCol] == 1)
             {
                  countOnes = countOnes + 1;
          // now iterate the colums of ALLMUTANTS array at that row where it is 1
                  for (int AllMutantIndex = 0; AllMutantIndex < AllMutants.GetLength(1); AllMutantIndex++)
                  {
                       AddBitsArray[AllMutantIndex] =
       AddBitsArray[AllMutantIndex] + AllMutants[distRow, AllMutantIndex];
                  }
      }
           } // ends for loop for distRow

 ///////////// now here calculate the centroid by dividing by the CountOnes and save in
/////////////the array Centroid for each cluster
            for (int loop = 0; loop < AddBitsArray.Length; loop++)
            {
                 float CentroidValue = (float)AddBitsArray[loop] / (float)countOnes;
                 CentroidValue = (float)(Math.Truncate(CentroidValue * 10) / 10);

                //*********************  saves the calculated centroid column by column
                Centroids[distCol,loop]= CentroidValue;
            }

        }  // ends distCol

    } // ends else if (CentroidIteration > 1)

Calculate_Distance();
}

////////////////////////////////////////////////////////////////////
//////////////////////////  CALCULATE DISTANCE////////////////////
//////////////////////////////////////////////////////////////////

 public void Calculate_Distance()
{
   float  Difference = 0;
   float AddedValues = 0;
   int col;
   newDistanceMatrix= new int[mutantsArray.GetLength(0), NumberOfClusters];

   ////////////////   CALCULATE HAMMING DISTANCE/ //////////////////////

   for (int rows = 0; rows < AllMutants.GetLength(0); rows++)  // rows of the all mutants array
    {
     // rows of the Centroids Array
     for (int CentroidRow = 0; CentroidRow < Centroids.GetLength(0); CentroidRow++)
      {
   // iterates columns of both arrays(AllMutants and Centroids)
           for (col = 0; col < Centroids.GetLength(1); col++)
           {
               if ((float)AllMutants[rows, col] > Centroids[CentroidRow, col])
               {
                  Difference = (float)AllMutants[rows, col] - Centroids[CentroidRow, col];

               }

              else
              {
                 Difference = Centroids[CentroidRow, col] - (float)AllMutants[rows, col];
              }

              AddedValues = Difference + AddedValues;
           }  // ends for loop of col
          HammingDistanceArray[rows, CentroidRow] = AddedValues;
          AddedValues = 0;
          Difference = 0;
       }  //ends for loop of centroidRow

   }  //   ends the for loop for row

////////////// ASSIGN 0'S AND 1'S TO THE DISTANCE MATRIX  /////////////////

    float SmallestValue = 0;
    int SaveColumn = 0;

    for (int DistanceRow = 0; DistanceRow < HammingDistanceArray.GetLength(0); DistanceRow++)
    {
        for (int DistanceCol = 0; DistanceCol < HammingDistanceArray.GetLength(1); DistanceCol++)
        {
            if (DistanceCol == 0)
            {
                SmallestValue = HammingDistanceArray[DistanceRow, DistanceCol];
                SaveColumn = DistanceCol;
            }
            else
            {
               if (SmallestValue < HammingDistanceArray[DistanceRow, DistanceCol])
               {

               }
               else if (SmallestValue > HammingDistanceArray[DistanceRow, DistanceCol])
               {
                   SmallestValue= HammingDistanceArray[DistanceRow, DistanceCol];
                   SaveColumn= DistanceCol;
               }
           }
        }  // ends loop DistanceCol

        newDistanceMatrix[DistanceRow, SaveColumn]=1;

    }// ends loop DistanceRow

/////////////// NOW COPY THE NEWDISTANCEMATRIX INTO THE OLD DISTANCE MATRIX //////////

   // first check rhat the two matrix are not equal
    NotEqualFlag=0;
    for (int row = 0; row < newDistanceMatrix.GetLength(0); row++)
    {
       for (int coll = 0; coll < newDistanceMatrix.GetLength(1); coll++)
       {
          if (oldDistanceMatrix[row, coll] == newDistanceMatrix[row, coll])
          {
              NotEqualFlag=0;   // EQUALLLLLL
          }
         else if (oldDistanceMatrix[row, coll] != newDistanceMatrix[row, coll])
          {
              NotEqualFlag=1;
              break;
          }
       }
       if (NotEqualFlag == 1)
       {
            break;  // breaks from the row loop
       }
   }  // ends row

  if (NotEqualFlag==1)
  {
      // copy 2D Arrays
      Array.Copy(newDistanceMatrix, oldDistanceMatrix, oldDistanceMatrix.Length);
      // here update the cluster values and assign the cluster values to the mutants they belong to
      Update_Clusters();
  }
 else if (NotEqualFlag==0)
 {
    // means the distances has become same and now thet mutants willnot converge or move
    // write to file here
    //KMeanSecondMethod file
    Update_Clusters();    // update before writing to file
  }


}  // ends the function Calculate_distance

////////////////////////////////////////////////////////////////////////////////
///////////// FUNCTION TO UPDATE THE CLUSTER VALUES AND ASSIGN CLUSTER /////////
//////////////VALUES TO THE MUTANTS THEY BELONG TO///////////////////////////
///////////////////////////////////////////////////////////////////////////////

public void Update_Clusters()
{
   // means which cluster it belongs to... 1, 2, 3 ,... so on
   int ClusterValue = 0;
   // iterates the columns, (cluster by cluster)
   for (int distCol = 0; distCol < newDistanceMatrix.GetLength(1); distCol++)
   {
       ClusterValue = ClusterValue+1;
       for (int distRow = 0; distRow < newDistanceMatrix.GetLength(0); distRow++)
       {
           if (newDistanceMatrix[distRow, distCol] == 1)
           {
                mutantsArray[distRow, 0] = System.Convert.ToString(ClusterValue);
           }

        }
    }// ends distCol

   if (NotEqualFlag == 1)
   {
      Calculate_Centroid();
    }
    else if (NotEqualFlag == 0)   // if the oldDistanceMatrix and newDistanceMatrix are same
    {
         WriteToFile();
 SanityFileWrite();
    }
}



///////////////////////////////////////////////////////////////////////
//////////////////// WRITE THE CLUSTERED MUTANTS TO FILE ///////////
////////////////////////////////////////////////////////////////////

public void WriteToFile()
{
     // Specify file, instructions, and privelegdes
     FileStream file = new FileStream("KMeanSecondMethod.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
     int ClusterValues=1;
     StreamWriter sw = new StreamWriter(file);
     string line;

     if (file.Length == 0)    // if file is initially empty them write on it
     {
        sw.WriteLine("*******Initial K values **********");
        for (int ind = 0; ind < NoRepititionMutant.Count; ind++)
        {
            sw.WriteLine(NoRepititionMutant[ind]);   // write all the initially selected k mutants
        }
        sw.WriteLine("Clustered mutants");
        while (ClusterValues <= NumberOfClusters)
        {
            sw.WriteLine(ClusterValues);
            for (int rows = 0; rows < mutantsArray.GetLength(0); rows++)
            {
                if (mutantsArray[rows, 0] == System.Convert.ToString(ClusterValues))
                {
                     sw.WriteLine(mutantsArray[rows, 1]);
                }
            }
            ClusterValues = ClusterValues + 1;
       }
     }// ends if (file.length ==0)

     else
     {
        StreamReader sr = new StreamReader(file);
        line = sr.ReadLine();
        while (line != null)  // read the file until blank line is read
        {
            line = sr.ReadLine();
        }
        sw.WriteLine("*******Initial K values **********");
        for (int ind = 0; ind < NoRepititionMutant.Count; ind++)
        {
            sw.WriteLine(NoRepititionMutant[ind]);
        }
        sw.WriteLine("Clustered mutants");
        while (ClusterValues <= NumberOfClusters)
        {
            sw.WriteLine(ClusterValues);
            for (int rows = 0; rows < mutantsArray.GetLength(0); rows++)
            {
               if (mutantsArray[rows, 0] == System.Convert.ToString(ClusterValues))
               {
                   sw.WriteLine(mutantsArray[rows, 1]);
               }
            }
             ClusterValues = ClusterValues + 1;
        }
         sw.WriteLine("*************************************");
       }  // ends else
        sw.Close();
 }

////////////////////////////////////////////////////////////////////
//////////////////// FUNCTION TO COUNT THE NUMBER ///////////////
///////////////////OF MUTANTS IN EACH CLUSTER///////////////////
/////////////////////////////////////////////////////////////////

public void SanityFileWrite()
{
   // Specify file, instructions, and privelegdes
    FileStream filee = new FileStream("Sanity_Check.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
    string linee;
    StreamWriter Sww = new StreamWriter(filee);
    int pass_Mutants_Length;
    pass_Mutants_Length = newDistanceMatrix.GetLength(0);
    int TotalClusters = newDistanceMatrix.GetLength(1);

   //************      Write to file     *****************//
   if (filee.Length == 0)    // if file is initially empty them write on it
   {
   Sww.WriteLine(TotalClusters);
   for (int col = 0; col < TotalClusters; col++)
   {
      for (int row = 0; row < newDistanceMatrix.GetLength(0); row++)
      {
         Sww.WriteLine(newDistanceMatrix[row, col]);
       }
    }
     Sww.WriteLine("***");    // used for seperation
     Sww.Close();
   }
   else   // if file is not empty then look for a blank line and then write after that
  {
    StreamReader srr = new StreamReader(filee);
    linee = srr.ReadLine();
     while (linee != null)  // read the file until blank line is read
     {
          linee = srr.ReadLine();
      }

     Sww.WriteLine(TotalClusters);
     for (int col = 0; col < TotalClusters; col++)
     {
        for (int row = 0; row < newDistanceMatrix.GetLength(0); row++)
        {
           Sww.WriteLine(newDistanceMatrix[row, col]);
        }
      }
     Sww.WriteLine("***");    // used for seperation
     Sww.Close();   // this closes the file
   }
  }
 }
}
```

Agglomerative Clustering Algorithm Class

Has the following functions:

- ReadMutants(); Gets the list of the entire size of mutants.
- CalculateCentroid(): To calculate the centroids
- CalculateDistance(): To calculate the distance of each mutant to the centroid
- Agglomerate(): To merge the two clusters based on min value.
- WriteToFile(): Writes to file the clustered mutants

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.IO;


 ////////////////////////////////////////////////////////
//////////Agglomerative Threshold/ Minimum Value//////////
/////// This algorithm first agglomerate based on a //////
/////////threshold value 1 then it agglomerate based /////
//on minimum value until all are merged into one cluster//

namespace GUI
{
   class AggloThresholdPLUSmin
    {
// all the variables of the classs
        int ClusterNumber = 1;
/// saves how many total clusters are there
int TotalClusters;
        string[,] mutantsArray;
        int BitSize = 0;
        int CentroidIteration = 0;
        int ClusterIteration = 1;
/// clustering is done based on this threshold
        int Threshold = 2;
// this will have the cluster numbers against eash mutant
        int[] Clusters;
        int[] OldClusters;
        int checkClusterValue = 0;
        int MinValueCriteria = 0;
        int FinallySame = 0;
        float[,] Distance;    // saves the actual distance
        int[,] DistanceMatrix; // saves the 0's and 1's
        float[,] Centroids;   // saves the Centroid of the clusters
        int WithoutThreshold = 0;

/////////////////////////////////////////////////////////
////////// gets the total  mutants of a particular program
//////////////////////////////////////////////////////////

        public void ReadMutants(string[,] mutantsArray1)
        {
            mutantsArray = mutantsArray1;
            string MutantString = mutantsArray[0, 1];
            BitSize = MutantString.Length;    // gives the length of bits or mutants

            OldClusters = new int[mutantsArray.GetLength(0)];

            Centroids = new float[mutantsArray.GetLength(0), BitSize];

            Calculate_Centroid();   // call the method to calculate centroids

        }

     /////////////////////////////////////////////////////////////////
     ////////////// CALCULATE THE CENTROID OF THE CLUSTERS////////////
     ////////////////////////////////////////////////////////////////

     public void Calculate_Centroid()
     {
        CentroidIteration = CentroidIteration + 1;
// first iteration of calculating the centroid
        if (CentroidIteration == 1)
        {
          for (int rows = 0; rows < mutantsArray.GetLength(0); rows++)
          {
             string mutant = mutantsArray[rows, 1];

              for (int cols = 0; cols < BitSize; cols++)
              {
                 if (System.Convert.ToInt32(mutant[cols]) == 49)
                 {
                     Centroids[rows, cols] = 1;
                  }
                else if (System.Convert.ToInt32(mutant[cols]) == 48)
                {
                            Centroids[rows, cols] = 0;
                }
             }
          }
        }
       else   /// for second time iteration
      {
 // this saves the indexes of the mutants with same cluster numbers
        ArrayList SameClusters = new ArrayList();
        TotalClusters = ClusterNumber - 1;
        Centroids = new float[TotalClusters, BitSize];
        int StartClusters = 1;
int calculated_cent_of_all = 0;
        int CentRow = 0;  // row index of the array centroids[]
        while (calculated_cent_of_all != TotalClusters)
        {
           for (int Xindex = 0; Xindex < Clusters.Length; Xindex++)
           {
               if (Clusters[Xindex] == StartClusters)
                {
                    SameClusters.Add(Xindex);
               }
           } //ends for loop

         StartClusters = StartClusters + 1;
  //// now calculate the cemtroid of a particualar cluster////
         string SameClusterMutantString;
         int AddBits = 0;
        for (int cols = 0; cols < BitSize; cols++) // traverse the columns
        {
          for (int rowss = 0; rowss < SameClusters.Count; rowss++)
          {
             int row = System.Convert.ToInt32(SameClusters[rowss]);
             SameClusterMutantString = mutantsArray[row, 1];
    if (System.Convert.ToInt32(SameClusterMutantString[cols]) == 49)
             {
                  AddBits = 1 + AddBits;
             }
            else if (System.Convert.ToInt32(SameClusterMutantString[cols]) == 48)
            {
                 AddBits = 0 + AddBits;
            }

          }
           float CentroidValue = (float)AddBits / (float)SameClusters.Count;
           CentroidValue = (float)(Math.Truncate(CentroidValue * 10) / 10);
           Centroids[CentRow, cols] = CentroidValue;
           AddBits = 0;
         }
           CentRow++;
           SameClusters.Clear();
           calculated_cent_of_all = calculated_cent_of_all + 1;
        }// ends while

          int r = 0;
       }// ends else
// call method to calculate the distance of clusters from other clusters
        Calculate_Distance();
   }


    ///////////////////////////////////////////////////////////////
    ////////////  CALCULATE THE DISTANCE OF MUTANTS /////////////
    //////////////////////////////////////////////////////////////

   public void Calculate_Distance()
   {
      /// this is the  mutant with which we will calculate the distance with others
      float[] CalculateDistanceWith = new float[BitSize];
      float AddedDistance = 0;
      float Difference = 0;
      Distance = new float[Centroids.GetLength(0), Centroids.GetLength(0)];
      int AllCentroidsTraversedFlag = 0;   // this keeps check that all the centroids have been traversed and calculated
      int row = 0;
      while (AllCentroidsTraversedFlag != Centroids.GetLength(0))
      {
        for (int coll = 0; coll < Centroids.GetLength(1); coll++)
        {
           CalculateDistanceWith[coll] = Centroids[row, coll];
        }
         row++;   // moves onto next row for next iteration

       //// now using the CalculateDistanceWith Array, calculate
       ////////the distance with other centroids////
        for (int IterateRows = 0; IterateRows < Centroids.GetLength(0); IterateRows++)
        {
          for (int IterateCol = 0; IterateCol < Centroids.GetLength(1); IterateCol++)
          {
             if (CalculateDistanceWith[IterateCol] > Centroids[IterateRows, IterateCol])
             {
                 Difference = CalculateDistanceWith[IterateCol] - Centroids[IterateRows, IterateCol];
             }
             else
             {
                 Difference = Centroids[IterateRows, IterateCol] - CalculateDistanceWith[IterateCol];
              }

            AddedDistance = AddedDistance + Difference;   // adds the distance
           }

          Distance[AllCentroidsTraversedFlag, IterateRows] = AddedDistance;
           AddedDistance = 0;
         }
        AllCentroidsTraversedFlag = AllCentroidsTraversedFlag + 1;
         AddedDistance = 0;
      } // ends while
       Agglomerate();

}

 ////////////////////////////////////////////////////////////////////////
 //////////////  CKUSTER/ AGGLOMERATE BASED ON SOME THRESHOLD VALUE////////
 ////////////////////////////////////////////////////////////////////////

public void Agglomerate()
{
   ClusterNumber = 1;
   DistanceMatrix = new int[Distance.GetLength(0), Distance.GetLength(1)];
   Clusters = new int[mutantsArray.GetLength(0)]; // saves the cluster numbers against the mutants
   int ClusterRow = 0;
   int ClusterCol = 0;
   int BothArraysSame = 0;
    ////******////// first look for 2   0's /////*****//////
    for (int row = 0; row < Distance.GetLength(0); row++)
   {
      for (int col = 0; col < Distance.GetLength(1); col++)
     {
        // if there is 0 at a location other then at row ==row
        if (Distance[row, col] == (float)0 & row != col)
        {
        if (DistanceMatrix[row, col] == 0 & DistanceMatrix[row, row] == 0 & DistanceMatrix[col, col] == 0)
        {
           ClusterRow = row;
           ClusterCol = col;
           DistanceMatrix[row, col] = 1;   // put 1 in place of the other 0
           DistanceMatrix[row, row] = 1;   // put 1 on place of the row==row
           DistanceMatrix[col, row] = 1;
           DistanceMatrix[col, col] = 1;
           if (ClusterIteration == 1)
           {
               Clusters[row] = ClusterNumber;
               Clusters[col] = ClusterNumber;
           }
           else if (ClusterIteration > 1)
           {
              for (int check = 0; check < Clusters.Length; check++)
               {
                  if (OldClusters[check] == ClusterCol + 1)
                   {
                       Clusters[check] = ClusterNumber;
                   }
                  if (OldClusters[check] == ClusterRow + 1)
                  {
                       Clusters[check] = ClusterNumber;
                  }
               }

            }

            ClusterNumber = ClusterNumber + 1;

          }
        }
      }
     }// ends for loop

/// ***//// now look for smaller distances, other than 0, less than the threshold //***///

     int OneCluster = 0;

     float TemporaryClusterValue = 0;
     float SmallestValue = 0;
      int SaveCol = 0;
      int WentINTOLoop = 0;
      int OK1 = 0;
      int OK2 = 0;
      if (WithoutThreshold == 0)
      {

       for (int roww = 0; roww < Distance.GetLength(0); roww++)
       {
           OneCluster = 0;
           OK1 = 0;
           OK2 = 0;
           for (int coll = 0; coll < Distance.GetLength(1); coll++)
           {
               ClusterRow = roww;
               ClusterCol = coll;
               if (Distance[roww, coll] < Threshold & Distance[roww, coll] != 0)
               {
                    TemporaryClusterValue = Distance[roww, coll];
                    if (DistanceMatrix[roww, coll] == (float)0 & DistanceMatrix[roww, roww] ==
                    (float)0 & DistanceMatrix[coll, coll] == (float)0)
                    {
                       if (ClusterIteration == 1)
                        {
                          if (Clusters[coll] == 0)
                           {
                               WentINTOLoop = 1;
                               {
                                   SmallestValue = TemporaryClusterValue;
   // saves the column where the value is the smallest
                                   SaveCol = coll;
                               }
                           else
                           {
                              if (SmallestValue < TemporaryClusterValue)
                                 {

                                 }
                               else if (SmallestValue > TemporaryClusterValue)
                                {
                                    SmallestValue = TemporaryClusterValue;
    // saves the column where the value is the smallest
                                    SaveCol = coll;
                                }
                            }

                          }   // ends if (Clusters[coll] == 0 )
                        }

                      else if (ClusterIteration > 1)
                     {
                        for (int check = 0; check < OldClusters.Length; check++)
                        {
                            if (OldClusters[check] == ClusterRow + 1)
                            {
                               if (Clusters[check] == 0)    // checks if the value is 0
                               OK1 = OK1 + 1;

                             }
                            if (OldClusters[check] == ClusterCol + 1)
                            {
                              if (Clusters[check] == 0)    // checks if the value is 0
                                                 OK2 = OK2 + 1;
                            }
                         }
                       if (OK1 > 0 & OK2 > 0)  // means the values at particular indexes in Cluster array is 0
                      {
                            WentINTOLoop = 1;
                          if (SmallestValue == (float)0)
                          {
                              SmallestValue = TemporaryClusterValue;
                             SaveCol = coll;
                          }
                          else
                         {
                             if (SmallestValue < TemporaryClusterValue)
                             {

                             }
                             else if (SmallestValue > TemporaryClusterValue)
                             {
                                 SmallestValue = TemporaryClusterValue;
                                 SaveCol = coll;
                             }
                          }
                      } // ends if (OK > 1)

                   }
                 }
              }
          } // ends for loop

      if (WentINTOLoop == 1)   // if the control went into the loop above
      {
         DistanceMatrix[roww, SaveCol] = 1;   // put 1 in place of the other 0
         DistanceMatrix[roww, roww] = 1;   // put 1 on place of the row==row
         DistanceMatrix[SaveCol, roww] = 1;
         DistanceMatrix[SaveCol, SaveCol] = 1;
         ClusterRow = roww;
         ClusterCol = SaveCol;
         if (ClusterIteration == 1)
         {
             Clusters[roww] = ClusterNumber;
             Clusters[SaveCol] = ClusterNumber;
         }

        else if (ClusterIteration > 1)
        {
            for (int check = 0; check < Clusters.Length; check++)
            {
               if (OldClusters[check] == ClusterCol + 1)
               {
                  Clusters[check] = ClusterNumber;
               }

               if (OldClusters[check] == ClusterRow + 1)
               {
                   Clusters[check] = ClusterNumber;
               }

            }
        }

         ClusterNumber = ClusterNumber + 1;
         OneCluster = 1;
     }  // ends if (WentINTOLoop == 1)

     if (OneCluster == 0 & DistanceMatrix[roww, roww] == 0)
     {
         if (ClusterIteration == 1)
         {
              DistanceMatrix[roww, roww] = 1;
              Clusters[roww] = ClusterNumber;
         }
         else if (ClusterIteration > 1)    // more than 1 iterations
        {
              DistanceMatrix[roww, roww] = 1;
              for (int find = 0; find < OldClusters.Length; find++)
              {
                     if (OldClusters[find] == ClusterRow + 1)
                     {
                            Clusters[find] = ClusterNumber;
                      }
               }
          }
          ClusterNumber = ClusterNumber + 1;   // increments the cluster value

       } // ends if(OneCluster==0 & DistanceMatrix[roww, roww]==0)

        SaveCol = 0;
        SmallestValue = 0;
        TemporaryClusterValue = 0;
         WentINTOLoop = 0;
      }  // ends for loop

    }  // end of WithoutThreshold

  else if (WithoutThreshold == 1)
   {

         int ColCheckk = 0;
         int findSomeOtherMinValue = 0;
         ArrayList NotThisCol = new ArrayList();
         int Other = 0;
         int NoSmallValue = 0;
         MinValueCriteria = 1;
         for (int roww = 0; roww < Distance.GetLength(0); roww++)
         {
              OneCluster = 0;
              OK1 = 0;
              OK2 = 0;
              findSomeOtherMinValue = 0;
      // loops through the columns
              for (int coll = 0; coll < Distance.GetLength(1); coll = coll + 0)
              {
                 ClusterRow = roww;
                 ClusterCol = coll;
                 Other = 0;
                 ColCheckk = coll;
                 if (Distance[roww, coll] != 0)
                 {
                     TemporaryClusterValue = Distance[roww, coll];

                     if (DistanceMatrix[roww, coll] == (float)0 & DistanceMatrix[roww, roww]
                     == (float)0 & DistanceMatrix[coll, coll] == (float)0)
                     {
                         for (int checkkk = 0; checkkk < NotThisCol.Count; checkkk++)
                         {
                            if (coll == System.Convert.ToInt32(NotThisCol[checkkk]))
                            {
                                     Other = 1;
                            }

                          }
                          if (Other == 0)
                          {
                             if (ClusterIteration == 1)
                              {
                                   if (Clusters[coll] == 0)
                                   {
                                       WentINTOLoop = 1;
                                       if (SmallestValue == (float)0)
                                        {
                                            SmallestValue = TemporaryClusterValue;
              // saves the column where the value is the smallest
                                             SaveCol = coll;
                                         }
                                        else
                                        {
                                           if (SmallestValue < TemporaryClusterValue)
                                           {

                                           }
                                           else if (SmallestValue > TemporaryClusterValue)
                                           {
                                                    SmallestValue = TemporaryClusterValue;
     // saves the column where the value is the smallest
                                                    SaveCol = coll;
                                           }
                                         }
                                      }   // ends if (Clusters[coll] == 0 )
                                  }

                                 else if (ClusterIteration > 1)
                                 {
                                    for (int check = 0; check < OldClusters.Length; check++)
                                     {
                                        if (OldClusters[check] == ClusterRow + 1)
                                        {
                                           if (Clusters[check] == 0)    // checks if the value is 0
                                            OK1 = OK1 + 1;
                                         }
                                         if (OldClusters[check] == ClusterCol + 1)
                                         {
                                             if (Clusters[check] == 0)    // checks if the value is 0
                                                    OK2 = OK2 + 1;
                                         }
                                     }
     // means the values at particular indexes in Cluster array is 0
                                    if (OK1 > 0 & OK2 > 0)
                                    {
                                       WentINTOLoop = 1;
                                       if (SmallestValue == (float)0)
                                       {
                                          SmallestValue = TemporaryClusterValue;
                                           SaveCol = coll;
                                       }
                                        else
                                       {
                                          if (SmallestValue < TemporaryClusterValue)
                                          {

                                           }
                                          else if (SmallestValue > TemporaryClusterValue)
                                          {
                                               SmallestValue = TemporaryClusterValue;
```

```
                                               SaveCol = coll;
                                          }
                                      }

                                    } // ends if (OK > 1)

                               }
                           }// ends "other" variable
                       }
                 }
               coll++;

         // now here check that the cminimum column value u have selected,
         //does it also have some othet minimum value
             if ((ColCheckk + 1) == Distance.GetLength(1))   // one before the length of column
             {
                if (SmallestValue == 0)
                {

                }
                else
                {
                   for (int iterateColl = 0; iterateColl < Distance.GetLength(1); iterateColl++)
                   {
                      if (SmallestValue < Distance[SaveCol, iterateColl])
                      {

                      }
                      else if (SmallestValue > Distance[SaveCol, iterateColl] && SaveCol != iterateColl)
                      {
                         findSomeOtherMinValue = 1;
                         NotThisCol.Add(SaveCol);
                         SaveCol = 0;
                         SmallestValue = 0;
                         coll = 0;
                          break;
                       }
                    }
                 }
               } //ends if

     } // ends for loop for col

       if (WentINTOLoop == 1)   // if the control went into the loop above
        {
            if (SmallestValue == 0)
            {
DistanceMatrix[roww, roww] = 1;   // put 1 on place of the row==row
            }
            else
            {
                DistanceMatrix[roww, SaveCol] = 1;   // put 1 in place of the other 0
                DistanceMatrix[roww, roww] = 1;   // put 1 on place of the row==row
                DistanceMatrix[SaveCol, roww] = 1;
                DistanceMatrix[SaveCol, SaveCol] = 1;
            }
            ClusterRow = roww;
            ClusterCol = SaveCol;
            if (ClusterIteration == 1)
            {
                Clusters[roww] = ClusterNumber;
                Clusters[SaveCol] = ClusterNumber;
             }
             else if (ClusterIteration > 1)
             {
             for (int check = 0; check < Clusters.Length; check++)
             {
                if (OldClusters[check] == ClusterCol + 1)
                 {
                   Clusters[check] = ClusterNumber;
                  }

                 if (OldClusters[check] == ClusterRow + 1)
                 {
                      Clusters[check] = ClusterNumber;
                 }
               }
             }
            ClusterNumber = ClusterNumber + 1;
            OneCluster = 1;
          }  // ends if (WentINTOLoop == 1)

        if (OneCluster == 0 & DistanceMatrix[roww, roww] == 0)
         {
          if (ClusterIteration == 1)
          {
               DistanceMatrix[roww, roww] = 1;
               Clusters[roww] = ClusterNumber;
          }
         else if (ClusterIteration > 1)    // more than 1 iterations
         {
             DistanceMatrix[roww, roww] = 1;
              for (int find = 0; find < OldClusters.Length; find++)
              {
                    if (OldClusters[find] == ClusterRow + 1)
                    {
                            Clusters[find] = ClusterNumber;
                    }
              }
           }
           checkClusterValue = ClusterNumber;
           ClusterNumber = ClusterNumber + 1;   // increments the cluster value
         } // ends if(OneCluster==0 & DistanceMatrix[roww, roww]==0)

           SaveCol = 0;
           SmallestValue = 0;
           TemporaryClusterValue = 0;
           WentINTOLoop = 0;
           NotThisCol.Clear();
         }  // ends for loop
       }

        ClusterIteration = ClusterIteration + 1;

         TotalClusters = ClusterNumber - 1;
        for (int check = 0; check < Clusters.Length; check++)     // loop to check the two cluster arrays
        {
           if (OldClusters[check] == Clusters[check])   // if the content at a particular index are same
           {
               BothArraysSame = 1;      // both cluster arrays are same
           }
           else
           {
              BothArraysSame = 0;
              break;
           }

         }

        if (BothArraysSame == 0 && WithoutThreshold == 0)    // if the clusters has changed
       {
           Clusters.CopyTo(OldClusters, 0);
           WriteToFile();
           Calculate_Centroid();
       }
      else if (BothArraysSame == 1 && WithoutThreshold == 0)
      {
            WithoutThreshold = 1;
      }
       if (WithoutThreshold == 1 && checkClusterValue != 1 && FinallySame == 0)
      {
          if (MinValueCriteria == 1 && checkClusterValue != 1)
          {
                WriteToFile();
          }

          if (checkClusterValue != 1)
          {
                Clusters.CopyTo(OldClusters, 0);
                Calculate_Centroid();
          }
       }

//if both oldCluster and Cluster Arrays are same, means
// no new cluster has been formed
       if (checkClusterValue == 1 && MinValueCriteria == 1 && FinallySame == 0)
       {
             FinallySame = 1;
       }

    }


   ///////////////////////////////////////////////////////////////////////////
   /////////// WRITE THE CLUSTERED MUTANTS TO FILE ///////////////////////////
   //////////////////////////////////////////////////////////////////////////
   public void WriteToFile()
   {
      // Specify file, instructions, and privelegdes
      FileStream file = new FileStream("AggloClustersM1.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
      int ClusterNumberr = 1;
      float dist = 0;
      StreamWriter sw = new StreamWriter(file);
      string line;
      if (file.Length == 0)    // if file is initially empty them write on it
      {
```

```
 while (ClusterNumberr <= TotalClusters)
 {
     sw.WriteLine(ClusterNumberr);
     for (int find = 0; find < Clusters.Length; find++)
       {
           if (Clusters[find] == ClusterNumberr)
            {
               sw.WriteLine(mutantsArray[find, 1]);
            }
        }

 ClusterNumberr = ClusterNumberr + 1;   // increments the cluster to be assigned

  }

 sw.WriteLine("the distances***************");
 for (int row = 0; row < Distance.GetLength(0); row++)
 {
    for (int col = 0; col < Distance.GetLength(1); col++)
    {
        dist = Distance[row, col];
        dist = (float)(Math.Truncate(dist * 10) / 10);
        sw.Write(dist);
        sw.Write(" ");
     }
     sw.WriteLine("");
  }

 sw.WriteLine("");
 sw.WriteLine("the distances matrixxxx***************");
 for (int row = 0; row < Distance.GetLength(0); row++)
 {
    for (int col = 0; col < Distance.GetLength(1); col++)
    {
       dist = DistanceMatrix[row, col];
       dist = (float)(Math.Truncate(dist * 10) / 10);
       sw.Write(dist);
       sw.Write(" ");
    }
      sw.WriteLine("");
 }
}
else
{
   StreamReader sr = new StreamReader(file);
   line = sr.ReadLine();
   while (line != null)             // read the file until blank line is read
  {
       line = sr.ReadLine();
 }
 sw.WriteLine("*********************************");
while (ClusterNumberr <= TotalClusters)
{

sw.WriteLine(ClusterNumberr);
for (int find = 0; find < Clusters.Length; find++)
{
   if (Clusters[find] == ClusterNumberr)
   {
        sw.WriteLine(mutantsArray[find, 1]);
    }
 }

          ClusterNumberr = ClusterNumberr + 1;
 }

 sw.WriteLine("the distances***************");
 for (int row = 0; row < Distance.GetLength(0); row++)
       {
           for (int col = 0; col < Distance.GetLength(1); col++)
           {
               dist = Distance[row, col];
               dist = (float)(Math.Truncate(dist * 10) / 10);
               sw.Write(dist);
               sw.Write(" ");
           }
           sw.WriteLine("");
       }
       sw.WriteLine("");
       sw.WriteLine("the distances matrixxxx***************");
       for (int row = 0; row < Distance.GetLength(0); row++)
       {
           for (int col = 0; col < Distance.GetLength(1); col++)
           {
               dist = DistanceMatrix[row, col];
               dist = (float)(Math.Truncate(dist * 10) / 10);
               sw.Write(dist);
               sw.Write(" ");
```

```
                    }
                    sw.WriteLine("");
                }
            }

            sw.Close();
        }
    }
}
```

Evaluation Class

Has the following functions:

- ReadFiles(): Reads all the mutants
- GreedyClusterTestSet(): Generates the test set Tc by greedy approach
- RandomTestSetFunction(): Generate test set Tr from random mutants of same size as Tc
- KillabilityEfficiency(): Calculates the mutation score of both test sets

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.IO;


     ///////////////////////////////////////////////////////////////////////////////
     //////////////     Evaluation using the greedy alorithm    ////////////////////
     /////////////     Find the cluster test set Tc by greedy algo ////////////////
     ///////////////////////////////////////////////////////////////////////////////



namespace GUI
{
  class GreedyEvaluation
  {
        ArrayList SelectFromClusters = new ArrayList();   // mutants selected from clusters
        ArrayList AllMutants4 = new ArrayList();
        ArrayList AllMutants = new ArrayList();
        int[] AllMutants1;// = new ArrayList(); // same as AllMutants saves all the 512 mutants
        int[] AllMutants2;   // = new ArrayList(); // same as AllMutants saves all the 512 mutants
        Random rand = new Random();
        int HowManyMutants = 0;
        ArrayList MutantsPerCluster = new ArrayList();
        int MutantPerCluster = 2;
        int RandomNumber = 0;
        ArrayList ClusterTestSet = new ArrayList();
        ArrayList RandomTestSet = new ArrayList();
// this will save the selected mutants from the clusters in the form of bits
        int[,] SelectFromClustersArray;
// this saves all the mutants in the form of bits, o's and 1's
        int[,] AllMutantsArray;
        int[] ClusterKillability;
        int NumberOfRuns;
        int LengthOfMutant;
// saves the test set generated form the clustered mutants
        ArrayList ClusterTestSet1 = new ArrayList();
// saves teh test set generated from the random mutants
        ArrayList RandomTestSet2 = new ArrayList();
        ArrayList RandomEfficiency = new ArrayList();
        ArrayList ClusterEfficiency = new ArrayList();

     public void ReadFiles(int numberOfRuns1, int LengthOfMutant1, int MutantPerCluster1)
     {

        NumberOfRuns = numberOfRuns1;
        LengthOfMutant = LengthOfMutant1;
        MutantPerCluster = MutantPerCluster1;

 // agglomerative clustering with threshold
        //System.IO.StreamReader file = new System.IO.StreamReader("AggloEvaluationFileM1.txt");
// agglomerative clustering with minimum value
        System.IO.StreamReader file = new System.IO.StreamReader("AggloEvaluationFileM2.txt");
// k mean method 2, careful selection of initial k
        // System.IO.StreamReader file = new System.IO.StreamReader("EvaluationFileM2.txt");

        string line;

       /////////// read and select 2 mutants from each cluster/////////////
       while ((line = file.ReadLine()) != null)
       {
          if (line.Length < LengthOfMutant)  // means it is the cluster number like 1, 2, 3, 4...
          {
             if (MutantsPerCluster.Count > 0 & MutantsPerCluster.Count >= MutantPerCluster)
```

```
              {
          // specifies the maximum value for the random function to generate
                  int MaxValue = MutantsPerCluster.Count;
                  // this loop is to select 2 randon mutants from
                  for (int select = 0; select < MutantPerCluster; select = select + 0)  each cluster
                  {
                      RandomNumber = rand.Next(MaxValue);
      // if the index does not have a 0 value instead of mutant
                     if (System.Convert.ToString(MutantsPerCluster[RandomNumber]) != "48")
                     {
                        string SizeCheck = System.Convert.ToString(MutantsPerCluster[RandomNumber]);
                        if (SizeCheck.Length < LengthOfMutant)
                        {
                            int t = 0;
                        }
                        else
                        {
                           SelectFromClusters.Add(MutantsPerCluster[RandomNumber]);
                           MutantsPerCluster[RandomNumber] = 0;   // once the mutant has been selected and
    //added to the arraylist , put a 0 in its place
                           HowManyMutants = HowManyMutants + 1;
                           select = select + 1;

                         }
                      }
                     else  // if 0 value then decrement and get another random integer.
                  {
                       select--;
                  }
              }
             }

            MutantsPerCluster.Clear();

          } //ends if (MutantsPerCluster.Count > 0 & MutantsPerCluster.Count> MutantsPerCluster)


          // if the number of mutants to select from each cluster is grater than the size of the array
          // then select all the mutants from that cluster
          else if (MutantsPerCluster.Count > 0 & MutantsPerCluster.Count < MutantPerCluster)
          {
             for (int loopList = 0; loopList < MutantsPerCluster.Count; loopList++)
              {
                   SelectFromClusters.Add(MutantsPerCluster[loopList]);
                   HowManyMutants = HowManyMutants + 1;
              }
             MutantsPerCluster.Clear();
            }

        }   // ends if (line.Length < LengthOfMutant)
        else
       {
            MutantsPerCluster.Add(line);  // adds mutants of each cluster in array list
       }

     }
// this part is for the mutants just before the end
        if (MutantsPerCluster.Count > 0 & MutantsPerCluster.Count >= MutantPerCluster)   of file
        {

           int MaxValue1 = MutantsPerCluster.Count;  // specifies the maximum value for the random function to generate
    // this loop is to select 2 randon mutants from each cluster
           for (int select = 0; select < MutantPerCluster; select = select + 0)
           {
               RandomNumber = rand.Next(MaxValue1);
 // if the index does not have a 0 value instead of mutant
               if (System.Convert.ToString(MutantsPerCluster[RandomNumber]) != "48")
               {
                   string SizeCheckk = System.Convert.ToString(MutantsPerCluster[RandomNumber]);
                   if (SizeCheckk.Length < LengthOfMutant)
                   {

                   }
                   else
                   {
                       SelectFromClusters.Add(MutantsPerCluster[RandomNumber]);
// once the mutant has been selected and added to the arraylist , put a 0 in its
                       MutantsPerCluster[RandomNumber] = 0;    place
                       HowManyMutants = HowManyMutants + 1;
                       select = select + 1;

                    }
                }
                else  // if 0 value then decrement and get another random integer.
               {
                   select--;
               }
            }/// end for
        }
        else if (MutantsPerCluster.Count > 0 & MutantsPerCluster.Count < MutantPerCluster)
        {
           for (int loopList = 0; loopList < MutantsPerCluster.Count; loopList++)
           {
                 SelectFromClusters.Add(MutantsPerCluster[loopList]);
                 HowManyMutants = HowManyMutants + 1;
            }

              MutantsPerCluster.Clear();
        }

      //**/// Read all the 512 mutants from the file and save in the AllMutants ArrayList   //***//

     System.IO.StreamReader file1 = new System.IO.StreamReader("Data.txt");
     while ((line = file1.ReadLine()) != null)
      {
           string[] splitData = line.Split(new Char[] { ' ' });
           AllMutants4.Add(splitData[1]);                    // all the 512 mutants added to the array list
      }
      // save the selected mutants from the clusters in the form
             SelectFromClustersArray = new int[HowManyMutants, LengthOfMutant];    of bits

  ////***///// Copy mutants bit by bit in the array SelectFromClustersArray  ////****/////

       string mutant = System.Convert.ToString(SelectFromClusters[0]);
       for (int row = 0; row < SelectFromClusters.Count; row++)
       {
            mutant = System.Convert.ToString(SelectFromClusters[row]);

            for (int col = 0; col < LengthOfMutant; col++)
            {
               if (mutant[col] == 49)
               {
                    SelectFromClustersArray[row, col] = 1;
               }
               else if (mutant[col] == 48)
               {
                    SelectFromClustersArray[row, col] = 0;
               }
             }
        }

///***///// copy all the mutants bit by bit in the AllMutantsArray  ///***////

       AllMutantsArray = new int[AllMutants4.Count, LengthOfMutant];
       string mutantt;
       for (int Rowsss = 0; Rowsss < AllMutants4.Count; Rowsss++)
       {
          mutantt = System.Convert.ToString(AllMutants4[Rowsss]);

           for (int Colll = 0; Colll < LengthOfMutant; Colll++)
           {
              if (mutantt[Colll] == 49)
              {
                  AllMutantsArray[Rowsss, Colll] = 1;
              }
               else if (mutantt[Colll] == 48)
               {
                     AllMutantsArray[Rowsss, Colll] = 0;
               }
            }
        }

      ///*****/// here write in file the selected mutants from the clusters ///***///

       // Specify file, instructions, and privelegdes
       FileStream file22 = new FileStream("GreedyEfficiencyK0.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

        StreamWriter sw = new StreamWriter(file22);
        string line1;
        if (file22.Length == 0)    // if file is initially empty them write on it
        {

          //&&&&&&&&&&&&&&&&&& just checking&&&&&&&&&&&&&&&&&&&
          for (int y = 0; y < SelectFromClusters.Count; y++)
           {
               sw.WriteLine(SelectFromClusters[y]); ;
            }

        }
        else
        {
           StreamReader sr = new StreamReader(file22);
           line1 = sr.ReadLine();
           while (line1 != null)     // read the file until blank line is read
           {
                 line1 = sr.ReadLine();
```

```
         }
         //&&&&&&&&&& writing the selected mutants onto the FILE &&&&&&&&&&&
         for (int y = 0; y < SelectFromClusters.Count; y++)
          {
               sw.WriteLine(SelectFromClusters[y]); ;
          }

          //&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
      }

      sw.Close();
       GreedyClusterTestSet();

   }   // ends the function readFile()


  ////////////////////////////////////////////////////////////////////////////////
  ///////////  Generate test set from clusters by greedy algorithm  ///////////////
  ///////////////////////////////////////////////////////////////////////////////

public void GreedyClusterTestSet()
{
    ArrayList ClusterTestSet = new ArrayList();   // saves the test set generated form the clustered mutants
    // this saves the total number of mutants killed by rach test case
    int [] HowManyMutantsKilled  = new int[LengthOfMutant];
     int MaxValue = 0;   // saves the test set that kills the most

   /// /** *///  get the CLUSTER TEST SET Greedily for the mutants from the clusters  ///****////

    int KillFlag = 0;
    int PreventTestCaseRepitition = 0;
    ClusterKillability = new int[HowManyMutants];
    int countZeros = 0;
    int TestCase = 0;

    while (KillFlag != 1)
    {
        // this loop counts all the mutants that can be killed by a particular test case
        for (int col = 0; col < SelectFromClustersArray.GetLength(1); col++)  // iterates the column of the array
        {
             for (int row = 0; row < SelectFromClustersArray.GetLength(0); row++)
             {
               if (SelectFromClustersArray[row, col] == 0)
               {
                     countZeros = countZeros + 1;
               }
              }// end row for loop

          HowManyMutantsKilled[col] = countZeros;
          countZeros = 0;

         }// end col for loop

       // once all the data has been saved in tha array HowManyMutantsKilled,
       // look for the test case that kills the most mutants
       // so now look for the maximum number
      for (int loop = 0; loop < HowManyMutantsKilled.Length; loop++)
      {
          if (MaxValue == 0)
          {
             MaxValue= HowManyMutantsKilled[loop];
             TestCase = loop;      // saves the index of the test case
          }
          else if (MaxValue < HowManyMutantsKilled[loop])
          {
              MaxValue= HowManyMutantsKilled[loop];
              TestCase = loop;  // saves the index of the test case
           }
       } // ends for loop

      MaxValue = 0;

     // now at the location of the maximum test case see which ones are killed by it
     // and put 1 at the place of killed mutants

      for (int rows = 0; rows < SelectFromClustersArray.GetLength(0); rows++)   // iterates the rows
      {
         if (SelectFromClustersArray[rows, TestCase] == 0)   // if the mutant is kileld by the test case
         {
               ClusterKillability[rows]=1;
         }

       }

     /// add the maximum value test case to the test set array
     ClusterTestSet.Add(TestCase);
     // now put all 0's in the rows where mutants are killed, in this array SelectFromClustersArray[]
     // this is done so that in next iteration when the max number of mutant killings are calculated, the
```

```
      // ones that are already counted are excluded(by putting 0's)

      for (int loopThrough = 0; loopThrough < ClusterKillability.Length; loopThrough++)
      {
          if (ClusterKillability[loopThrough] == 1)
          {
             for (int coll = 0; coll < SelectFromClustersArray.GetLength(1); coll++)
             {
                 SelectFromClustersArray[loopThrough, coll]=1;
             }

         }

      }
        // now iterate through ClusterKillability[] Array and see if all the mutants have been killed

        for (int loopp = 0; loopp < ClusterKillability.Length; loopp++)
        {
           if (ClusterKillability[loopp] == 1)
           {
                KillFlag = 1;
           }
            else if (ClusterKillability[loopp] == 0)
           {
                  KillFlag = 0;
                  break;
          }
       }

     } // while (KillFlag != 1)

   int yy = 0;

   ClusterTestSet1 = ClusterTestSet;
   int u = 0;
   while (NumberOfRuns != 0)   // this is used to run the algorithm for specified number of runs
  {
       NumberOfRuns = NumberOfRuns - 1;
       AllMutants = new ArrayList(AllMutants4); // this copies the array list AllMutants4 into AllMutants
       AllMutants1 = new int[AllMutants.Count];
       AllMutants2 = new int[AllMutants.Count];
      RandomTestSetFunction();
  }
} // ends function ClusterTestSetFunction()


  ///////////////////////////////////////////////////////////////////////
  /////////  Generate RANDOM TEST SET same number as generated by //////
  ////////////////////greedy clusterTest SET//////////////////////////
  ////////////////////////////////////////////////////////////////////////

 public void RandomTestSetFunction()
 {
    ////******/// here get the same number of RANDOM TEST SET as the CLUSTER TEST SET ////***///
    int TotalNumberOfTestCases = ClusterTestSet1.Count;  // we need same number of test cases from the RandomMutants
    int TestCaseCount = 0;
    int[] MutantBitRow = new int[LengthOfMutant];
    int NotAdded = 0;
    int killed = 0;
    ArrayList RandomTestSet = new ArrayList();    // saves teh test set generated from the random mutants
      // until the test cases are not equal to the cluster test set count
     while (TestCaseCount != TotalNumberOfTestCases)
     {
         int RandomRow = rand.Next(AllMutants.Count);
       string RandomMutant = System.Convert.ToString(AllMutants[RandomRow]);
//    the length of the selected random mutant should be equal to the length mutant bits
       if (RandomMutant.Length == LengthOfMutant)
       {
   // save this random mutant in the array bit by bit
           for (int index = 0; index < RandomMutant.Length; index++)
           {
               if (RandomMutant[index] == 49)
               {
                     MutantBitRow[index] = 1;
               }
                else if (RandomMutant[index] == 48)
                {
                           MutantBitRow[index] = 0;
                }
            }

            while (killed != 1)
           {
              int KillingBit = rand.Next(RandomMutant.Length);
              NotAdded = 0;
             if (MutantBitRow[KillingBit] == 0)// if the test case kills it
             {
                  for (int check = 0; check < RandomTestSet.Count; check++)
                  {
// checks that the test case has not already been added to the test set
                      if (System.Convert.ToInt32(RandomTestSet[check]) != KillingBit)
                      {
                             NotAdded = 0;
                      }
                      else
                      {
                         NotAdded = 1;
    // put zero in its place if the row has been selected randomly
                         AllMutants[RandomRow] = 0;
                         killed = 1;
                         break;
                       }
                   }
     // only proceed if the test case has not already been added to the test set
                  if (NotAdded == 0)
                  {
                      AllMutants[RandomRow] = 0;   // put zero in its place if the row has been selected randomly
                      RandomTestSet.Add(KillingBit);
                      TestCaseCount = TestCaseCount + 1;
                      killed = 1;
                   }
               }   // ends if
           }// ends while, killed ==1

            killed = 0;
          }   // ends if (RandomMutant.Length > 0)
          else
           {
                 int hello = 0;   // length less then the length of the mutant bits
           }

         }  // ends while(TestCaseCount != TotalNumberOfTestCases)

        RandomTestSet2 = RandomTestSet;

       KillabilityEfficiency();

    }   // ends the random test set function


    ////////////////////////////////////////////////////////////////////////
    //////////// CALCULATES THE EFFICIENCY OF BOTH TEST SETS ////////////////
    ////////////////////////////////////////////////////////////////////////

 public void KillabilityEfficiency()
 {
      /////********////first see the killability of the CLUSTER TEST SET ////***/////
      int CTestCase;
      int KillCounts = 0;
      for (int Cloop = 0; Cloop < ClusterTestSet1.Count; Cloop++)
      {
         CTestCase = System.Convert.ToInt32(ClusterTestSet1[Cloop]);
          for (int row = 0; row < AllMutantsArray.GetLength(0); row++)
           {
               if (AllMutantsArray[row, CTestCase] == 0)
               {
                  if (AllMutants1[row] != 1)
                   {
                        AllMutants1[row] = 1;
                        KillCounts = KillCounts + 1;

                    }
                  }
                }
            }
       //// calculate the killability percentage ///

        int countZeros = 0;
         float CkillabilityPercentage;
         for (int loooop = 0; loooop < AllMutants1.Length; loooop++)
         {
            if (AllMutants1[loooop] == 1)
             {
                  countZeros = countZeros + 1;
              }
         }

          CkillabilityPercentage = ((float)countZeros / (float)AllMutants1.Length) * 100;
  // this is the percentage of killability
         CkillabilityPercentage = (float)(Math.Truncate(CkillabilityPercentage * 10) / 10);
         ClusterEfficiency.Add(CkillabilityPercentage);

        ////******//first see the killability of the RANDOM TEST SET ///****////
        KillCounts = 0;
        int RTestCase = 0;
       for (int Rloop = 0; Rloop < RandomTestSet2.Count; Rloop++)
```

```
{
    RTestCase = System.Convert.ToInt32(RandomTestSet2[Rloop]);
   for (int roww = 0; roww < AllMutantsArray.GetLength(0); roww++)
   {
       if (AllMutantsArray[roww, RTestCase] == 0)
       {
           if (AllMutants2[roww] != 1)
           {
                 AllMutants2[roww] = 1;
                  KillCounts = KillCounts + 1;
            }
        }
    }

}

/// now calculate the percentage killability of mutants by andom test set/////
int Ones = 0;
float RkillabilityPercentage;
for (int loop = 0; loop < AllMutants2.Length; loop++)
{
     if (AllMutants2[loop] == 1)
      {
          Ones = Ones + 1;
      }
}
 RkillabilityPercentage = ((float)Ones / (float)AllMutants2.Length) * 100;
 RkillabilityPercentage = (float)(Math.Truncate(RkillabilityPercentage * 10) / 10);
RandomEfficiency.Add(RkillabilityPercentage);

   ////////***** //////WRITE ON FILE ////////*******////////

   // Specify file, instructions, and privelegdes
   FileStream file = new FileStream("GreedyEfficiencyK0.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

   StreamWriter sw = new StreamWriter(file);
   string line;
   if (file.Length == 0)    // if file is initially empty them write on it
   {
         sw.WriteLine("TEST SET FROM CLUSTERS( Tc )");
         sw.Write("{");
         for (int loop = 0; loop < ClusterTestSet1.Count; loop++)
         {
             sw.Write(ClusterTestSet1[loop]);
             sw.Write(",");
             sw.Write("  ");
         }
         sw.Write("}");
         sw.WriteLine("");
         sw.Write("Killed mutants by Tc = ");
         sw.Write(CkillabilityPercentage);
         sw.WriteLine(" %");

         sw.WriteLine("RANDOM TEST SET( Tr )");
         sw.Write("{");
         for (int loop = 0; loop < RandomTestSet2.Count; loop++)
         {
             sw.Write(RandomTestSet2[loop]);
             sw.Write(",");
             sw.Write("  ");
         }
         sw.Write("}");

         sw.WriteLine("");
         sw.Write("Killed mutants by Tr = ");
         sw.Write(RkillabilityPercentage);
         sw.WriteLine(" %");
         sw.WriteLine("");
         sw.WriteLine("......................................");
         sw.WriteLine("");
     }
     else
     {
         StreamReader sr = new StreamReader(file);
         line = sr.ReadLine();

         while (line != null)  // read the file until blank line is read
         {
             line = sr.ReadLine();
         }
         ////&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

         sw.WriteLine("TEST SET FROM CLUSTERS( Tc )");
         sw.Write("{");
         for (int loop = 0; loop < ClusterTestSet1.Count; loop++)
         {
             sw.Write(ClusterTestSet1[loop]);
             sw.Write(",");
```

```
                sw.Write("  ");
            }
            sw.Write("}");

            sw.WriteLine("");
            sw.Write("Killed mutants by Tc = ");
            sw.Write(CkillabilityPercentage);
            sw.WriteLine(" %");

            sw.WriteLine("RANDOM TEST SET( Tr )");
            sw.Write("{");
            for (int loop = 0; loop < RandomTestSet2.Count; loop++)
            {
                sw.Write(RandomTestSet2[loop]);
                sw.Write(",");
                sw.Write("  ");
            }
            sw.Write("}");

            sw.WriteLine("");
            sw.Write("Killed mutants by Tr = ");
            sw.Write(RkillabilityPercentage);
            sw.WriteLine(" %");
            sw.WriteLine("");
            sw.WriteLine("................................");
            sw.WriteLine("");
    }

    sw.Close();
    if (NumberOfRuns == 0)
    {
       // Specify file, instructions, and privelegdes
       FileStream filee = new FileStream("SortedEfficiencyKO.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

            StreamWriter sww = new StreamWriter(filee);
            string linee;
            if (filee.Length == 0)    // if file is initially empty them write on it
            {
                sww.WriteLine("Cluster Killability");

                for (int xx=0; xx< ClusterEfficiency.Count; xx++)
                {
                    sww.WriteLine(ClusterEfficiency[xx]);
                }

                sww.WriteLine("Random Killability");
                RandomEfficiency.Sort(0, RandomEfficiency.Count, new Desc());

                for (int x=0; x< RandomEfficiency.Count; x++)
                {
                    sww.WriteLine(RandomEfficiency[x]);
                }
            }
            else
            {
                StreamReader srr = new StreamReader(filee);
                linee = srr.ReadLine();

                while (linee != null)             // read the file until blank line is read
                {
                    linee = srr.ReadLine();
                }
                sww.WriteLine("*************************************");
                sww.WriteLine("Cluster Killability");

                for (int xx=0; xx< ClusterEfficiency.Count; xx++)
                {
                    sww.WriteLine(ClusterEfficiency[xx]);
                }

                sww.WriteLine("Random Killability");
                RandomEfficiency.Sort(0, RandomEfficiency.Count, new Desc());

                for (int x=0; x< RandomEfficiency.Count; x++)
                {
                    sww.WriteLine(RandomEfficiency[x]);
                }



            }
            sww.Close();

      }

  public class Desc : IComparer
  {
      public int Compare(object x, object y)
```

```
            {

                if ((float)x > (float)y) return 1;
                if ((float)x == (float)y) return 0;
                if ((float)x < (float)y) return -1;
                // MODIFY: Code below added to ensure all values returned.
                else
                return 0;
            }
        }
    }
}
```