Optimal Regression Test Case Prioritization using genetic algorithm

T. Prem Jacob [1], Dr. T. Ravi [2]

[1.] Research Scholar, Department of Computer Science and Engineering, Sathyabama University, Chennai, India
[2.] Principal, Srinivasa Institute of Engineering and Technology, Chennai, India
premjac@yahoo.com

Abstract: Regression testing is an essential and expensive activity in the maintenance phase to show that the code has not been affected by the changes. It consumes 80% of the maintenance cost. Hence optimizing the regression testing will be the prime motives for the software testers. We prioritize the test case based on number of the modified lines the test case covers. The test case that covers the maximum number of the modified lines is given highest priority, and executed first. Hence even if testing is not completed we can cover maximum number of modified lines. The test cases prioritization is done by using genetic algorithm. It takes the test case information as input and it produces a sequence of the test case that has to be executed so that maximum number of the modified code gets covered.
[T. Prem Jacob, T. Ravi. **Optimal Regression Test Case Prioritization using genetic algorithm**. *Life Sci J* 2013;10(3):1021-1033] (ISSN:1097-8135). http://www.lifesciencesite.com. 149

Keywords: Regression Testing; Test Case; Genetic Algorithm.

1. Introduction

Software testing requires resources and consumes 30-50% of the total cost of development. Testing is often done in time to market pressure and is supposed to test whole software in a systematic manner to achieve quality as much as possible. Testing also includes many other expectations such as delivering error free versions and checking thorough software iteration in available time and other resources. It may not be possible for testers provide quality product free of bugs to customers, so it ultimately raises the possibility of potential risks in software, while on the other hand time slippage occurs for delivering the satisfactory quality assessment of software. Testing has been traditionally performed in value neutral approach in which all software parts are given same testing resources to test but this eventually does not satisfy the end customer as approximate 36% of software functions are only often used .Therefore it is meaningless to test the whole software in this way. One type of testing is a regression testing in which software is tested after making some changes to it. Regression testing is considered to be very expensive due to repeated execution of existing test cases. Regression testing involves execution of a large number of test cases and is time consuming. It is impractical to repeatedly test the software by executing a complete set of test cases under resource constraints. Because of these reason researches have considered various methods for reducing the cost of regression testing, this includes test case minimization, and regression test selection, test suite minimization techniques lower cost by reducing a test suite to a minimal subset that maintains equivalent coverage of the original test suite with respect to a particular test adequacy criterion, regression test selection method reduces the cost of regression testing by selecting an appropriate subset of the existing test suite based on information about the program, modified version.

Test suite minimization methods and Regression test selection, however, can have drawbacks. For example, although some empirical evidence indicates that, in certain cases, there is little or no loss in the ability of a minimized test suite to reveal faults in comparison to its non-minimized original other empirical evidence shows that the fault detection capabilities of test suites can be severely compromised by minimization. Similarly, although there are safe regression test selection techniques that can ensure that the selected subset of a test suite has the same fault detection capabilities as the original test suite, the conditions under which safety can be achieved do not always hold. Therefore, there is a need to schedule the test cases based on some criteria. This process is called test case prioritization. There are many criteria, based on which one can prioritize the test cases. For example, ordering test cases based on, total coverage of the code components, ordering the test cases based on, coverage of code components which not previously covered, ordering test cases will be based on their ability to reveal the faults in the code they cover. We use prioritization based on a number of code covered or modified lines. The test case with maximum number of modified lines is executed the earliest and the one with the least number of modified lines executed at the last.

When the time required to re-execute an entire test suite is short, test case prioritization may not be cost-effective, it may be sufficient simply to

schedule test cases in any order. When the time required to execute an entire test suite is sufficiently long, however, test-case prioritization may be beneficial because, in this case, meeting testing goals earlier can yield meaningful benefits. Because test case prioritization techniques do not themselves discard test cases, they can avoid the drawbacks that can occur when regression test selection and test suite minimization discard test cases. Alternatively, in cases where the discarding of test cases is acceptable, test case prioritization can be used in conjunction with regression test selection or test suite minimization techniques to prioritize the test cases in the selected or minimized test suite. Further, test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time will have been spent more beneficial than if test cases were not prioritized.

2. Literature Survey

Yu-Chi Huang (2010) has proposed a cost cognizant test case prioritization technique based on the use of historic records and genetic algorithm. They run a controlled experiment to evaluate the proposed technique's effectiveness. This technique however does not take care of the test cases similarity.

Sangeeta Sabharwal (2011) has proposed a technique for prioritization test case scenarios derived from activity diagram using the concept of basic information flow metric and genetic algorithm. Sangeeta Sabharwal (2011) has generated prioritized test case in static testing using genetic algorithm. They have applied a similar approach as to prioritize test case scenarios derived from source code in static testing.

James H. Andrews (2011) has applied genetic algorithm for randomized unit testing to figure out the best suitable test cases.

Mohsen FallahRad (2011) has applied common genetic and bacteriological algorithm for optimizing testing data in mutation testing.

RuchikaMalhotra (2011) has developed an adequacy based test data generation technique using genetic algorithms.

3. Problem Definition

Prioritizations (orderings) of T and f are a function that, applied to any such ordering, yields an award value for that ordering. For simplicity, and without loss of generality, the definition assumes that higher award values are preferable to lower ones.

For given T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real number. Our aim is to find T' $\in$ PT such that

$$(\forall T'')\,(T'' \in PT')$$
$$(T'' \neq T')\,[\,f(T') \geq f(T'')]$$

There are several aspects of the test case prioritization problem that are worth describing further. There are many possible goals of prioritization. To measure the success of a prioritization technique in meeting any such goal, however, we must describe the goal quantitatively. In Definition, f represents such quantification. We will precisely define one particular function f for use in quantifying the first of these goals. Depending upon the choice of f, the test case prioritization problem may be intractable. For example, given a function f that quantifies whether a test suite achieves statement coverage at the fastest rate possible, an efficient solution to the test case prioritization problem would provide an efficient solution to the knapsack problem. Similarly, given a function f that quantifies whether a test suite detects faults at the fastest rate possible, a precise solution to the test case prioritization problem would provide a solution to the halting problem. In such cases, prioritization techniques must be heuristics. Test case prioritization can be used either in the initial testing of software or in the regression testing of software. One difference between these two applications is that, in the case of regression testing, prioritization techniques can use information gathered in previous runs of existing test cases to help prioritize the test cases for subsequent runs. It is useful to distinguish two varieties of test case prioritization, general test case prioritization and version specific test case prioritization. In general test case prioritization, given a program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering of test cases that will be useful over a succession of subsequent modified versions of P. Thus, general test case prioritization can be performed following the release of any version of the program during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases. It is hoped that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, on average over those subsequent releases. In contrast, in version-specific test case prioritization, given a program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering that will be useful on a specific Version P' of P. Version-specific prioritization is performed over a set of changes have been made to P and prior to regression testing P'. Because this prioritization is accomplished after P' is available, care must be taken to keep the cost of performing the prioritization from excessively delaying the very regression testing activities it is intended to facilitate. The prioritized test suite may be more effective at meeting the goal of the prioritization for P' in particular than would a test suite resulting from general test case prioritization, but may be less effective on average over a succession of subsequent releases. Typically though not necessarily the general test case prioritization does not use

information about specific modified versions of P, whereas version specific prioritization does use such information. Of course, it is possible for general test case prioritization techniques to incorporate information about expected modifications to improve the average performance of prioritized test suites over a succession of program versions, and it is possible to use prioritization techniques that ignore the modified program as version-specific techniques. It is also possible to integrate test case prioritization with regression test selection or test suite minimization techniques for example, by prioritizing a test suite selected by a regression test selection algorithm, or by prioritizing the minimal test suite returned by a test suite minimization algorithm. Finally, given any prioritization goal, various prioritization techniques may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase the rate of the fault to be detected in the test suites, we might prioritize test cases in terms of the extent to which they execute modules that, measured historically, have tended to fail. Alternatively, we might prioritize test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirements specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than would an ad hoc or random ordering of test cases. We restrict our attention, focusing on general test case prioritization in application to regression testing, independent of regression test selection and test suite minimization.

4. Genetic Algorithm

Genetic algorithms (GAs) are search methods based on principles of natural selection and genetics. GAs encodes the decision variables of a search problem into finite-length strings of alphabets of certain cardinality. The strings which are candidate solutions to the search problem are referred to as chromosomes, the alphabets are referred to as genes and the values of genes are called alleles. For example, in a problem such as the travelling salesman problem, a chromosome represents a route, and a gene may represent a city. In contrast to traditional optimization techniques, GAs work with coding of parameters, rather than the parameters themselves. To evolve good solutions and to implement natural selection, it needs a measure for distinguishing good solutions from bad solutions. The measure could be an objective function that is a mathematical model or a computer simulation, or it can be a subjective function where humans choose better solutions over worse ones. In essence, the fitness measure must determine a candidate solution's relative fitness, which will subsequently be used by the GA to guide the evolution of good solutions. Another important concept of GAs is the notion of population. Unlike traditional search methods, genetic algorithms rely on a population of candidate solutions. The population size, which is usually a user-specified parameter, is one of the important factors affecting the scalability and performance of genetic algorithms. For example, small population sizes might lead to premature convergence and yield substandard solutions.

Once the problem is encoded in a chromosomal manner and a fitness measure for discriminating good solutions from bad ones has been chosen, we can start to evolve solutions to the search problem using the following steps:

4.1. Initialization

The initial population of candidate solutions is usually generated randomly across the search space. However, domain-specific knowledge or other information can be easily incorporated.

4.2. Evaluation

Once the population is initialized or an offspring population is created, the fitness values of the candidate solutions are evaluated.

4.3. Selection

Selection allocates more copies of those solutions with higher fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection, some of which are described in the next section.

4.4. Recombination

Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring). There are many ways of accomplishing this and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner.

4.5. Mutation

While recombination operates on two or more parent chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.

4.6. Replacement

The offspring population created by selection, recombination, and mutation replaces the original parental population. Many replacement techniques such as elitist replacement, generation-wise

replacement and steady-state replacement methods are used in GAs. Repeat steps from evolution to replace until a terminating condition is met. Goldberg (1983, 1999, and 2002) has likened GAs to mechanistic versions of certain modes of human innovation and has shown that these operators when analyzed individually are ineffective, but when combined together they can work well as in Figure 1. This aspect has been explained with the concepts of the fundamental intuition and innovation intuition. The same study compares a combination of selection and mutation to continual improvement (a form of hill climbing), and the combination of selection and recombination of innovation (cross-fertilizing).

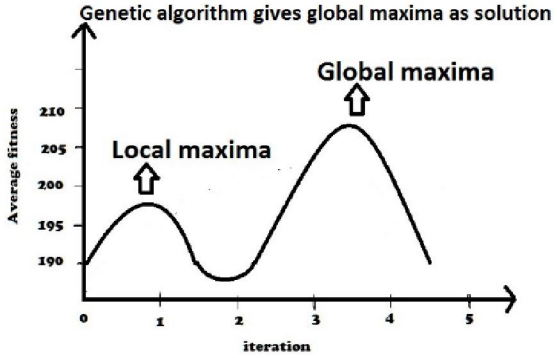

Figure.1. Genetic Algorithm Giving Global Maxima

5. Proposed Methodology

Genetic algorithm is stochastic search technique, which is based on the idea of selection of the fittest chromosome. In genetic algorithm, population of chromosome is represented by different codes such as binary, real number, permutation etc. genetic operators(i.e. selection, crossover, mutation) is applied on the chromosome in order to find more fittest chromosome. Fitness of the chromosome can be defined by a suitable objective function. As a class of stochastic method genetic algorithm is different from a random search. While genetic algorithm carry out a multidimensional search by maintaining population of potential user, random methods consisting of a combination of iterative search methods and simple random search methods can find a solution for a given problem. One of the genetic method's most attractive features is to explore the search space by considering the entire population of the chromosome.

The steps of genetic algorithm are:

1. Generate population (chromosome).
2. Evaluate the fitness of generated population.
3. Apply selection for individual.
4. Apply crossover and mutation.
5. Evaluate and reproduce the chromosome.

5.1. Generate Population

Initially population is randomly selected and encoded. Each chromosome represent the possible solution of the problem (in our case the sequence of test cases is chromosome and our aim is to optimize this sequence). For example- for 12 test cases T1, T2, T3…….T12 the sequence is

T1→T2→T4→T6→T9→T10→T12→T3→T5→T7

5.2. Evaluate the Fitness

Fitness of the chromosome can be defined by the objective function. An objective function tells how 'good' or 'bad' a chromosome is. This objective function generates a real number from the input chromosome. Based on this number two or more chromosome can be compared.

5.3. Apply Selection

In general the selection is depending on the fitness value of the chromosome. The chromosome with higher or lower value will be selected based on the problem definition.

5.4. Apply Crossover And Mutation

Parents are chosen and randomly combined. This technique for generating random chromosome is called crossover.

There exist two type of crossover.

(i). Single point crossover.

(ii). Multiple point crossover.

For example- suppose two sequences for test cases is

P1: T1→T2→T3→T4→T5→T6→T7→T8→T9

P2: T4→T2→T5→T7→T8→T1→T6→T9→T2

When we use one point crossover offspring can be

C1: T1→T2→T3→T4→T8→T6→T9→T5→T7

C2: T4→T3→T5→T7→T6→T8→T9→T1→T2

For C1 write first part of the P1 as it is and then write second part of P2 with constraint that a test case has not been added in to C1. For doing mutation two genes selected randomly along the chromosome and swapped with each other.

For example- when T3 and T9 get selected randomly

T1→T2→T3→T4→T8→T6→T9→T5→T7

T1→T2→T9→T4→T8→T6→T3→T5→T7

6. Test Case Optimization Using GA

This paper provides technique for test case prioritization using genetic algorithm. Let's say a program has test case suite T, now if one can make modification in the program p, suppose modified program is P', so in order to test program P' one can generate a prioritize sequence of test cases from test case suite T , on the basis of the line of code modified. Here the following genetic parameter will be used.

6.1. Fitness Function

The following objective function (fitness function) will be used.

Fitness value (F) = Σ {order * (number of modified lines covered by test cases)}

For example- a test case sequence is T1_T2_T3_T4 and T1, T2, T3 and T4 covers 2,1,5,3 modified lines of code respectively. Then fitness value for this sequence will be

F= (2*4) + (1*3) + (5*2) + (3*1) 16

In this T1 has order 4 and it covers 2 lines of code,T2 has order 3 and it contains 1 line of code , T3 has order 2 and it covers 5 line of code and T4 has order 1 and it covers 3 lines of code.

6.2. Crossover

Here one can use one point cross over with crossover probability Pc=0.33.

Crossover Probability=Fitness Function of Chromosomes/∑Fitness Function.

6.3. Mutation

Here we will use mutation probability Pm=0.2. It means that 20% of the genes will be muted within a chromosome. Example: Test cases with execution history.

Table1 tells us which test case covers which line code of the code being tested, one can see that test case with test case ID one covers statement eight, nine, ten, eleven, twelve and thirteen like case, one can find what are the statement numbers covered by particular software. This is helpful because later on when we know the number of modified lines, we can compare the number of modified lines with above information and sort out which test case covers most modified lines of code as in Table 2.

Each test case has to be also associated with its implicit properties such as the code functions that they parse through, within the development code, and the complexity of the tested code. Assume that lines 5, 8,10,15,20,23,28,35 are modified and the modified lines of code covered by each test case are shown in the table 3.

It shows the test cases which does not at all cover modified lines of code though they cover lines, now we use genetic algorithm because that is one of the best search problem that over comes the problem in hill climbing but if we already know that there is going to be one local maxima then hill climbing becomes more efficient.

Table 1.Test Case Execution History

Test Case ID	A	B	C	Expected Output	Execution History
T1	30	20	40	Obtuse angle triangle	8,9,10,11,12,13
T2	30	20	40	Obtuse angle triangle	8,9,10,11,12,13,14,15,16,17
T3	30	20	40	Obtuse angle triangle	10,11,12,13
T4	30	20	40	Obtuse angle triangle	10,11,12,13,14,15,16,20,21,22
T5	30	20	40	Obtuse angle triangle	12,13,14,15,16,20,21,22
T6	30	20	40		22,23,24,25,28
T7	30	20	40	Obtuse angle triangle	5, 6, 7, 8, 9, 10,11, 12, 13,14, 15, 16, 20,21, 15, 16,20, 21, 35
T8	-	-	-		
T9	30	20	40		5, 6, 7, 8, 9, 10,11, 12, 13,14, 15, 16, 20,21, 15, 16,20, 21, 35
T10	30	20	40		18, 19, 20, 21,35
T11	30	20	40	Obtuse angle triangle	24, 25
T12	30	20	40	Obtuse angle triangle	15, 16, 20, 21

Table 2.Test Case Code Coverage

Statement	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8	Test case 9	Test case 10	Test case 11	Test case 12
5							X		X			
6							X		X			
7							X		X			
8	X	X					X		X			
9	X	X					X		X			
10	X	X	X	X			X		X			
11	X	X	X	X			X		X			
12	X	X	X	X	X		X		X			
13	X	X	X	X	X		X		X			
14		X		X	X		X		X			
15		X		X	X		X		X			X
16		X		X	X		X		X			X
17		X										
18										X		
19										X		
20				X	X		X		X	X		X

21				X	X				X	X		X
22				X	X	X						
23						X						
24						X					X	
25						X					X	
26												
27												
28						X						
29												
30												
31												
32												
33												
34												
35							X		X	X		

Gregg Rothermel suggests that one can also prioritize the test cases based on the number of branch code the test case covers, the number of additional code that the test case covers. But we limit only to prioritize the test cases based on number of modified lines a test case covers.

Now we apply genetic algorithm, on this data find one can represents this information in a matrix, the first column would be order, second column would be number of lines modified by each test cases, and then one can generate random number without repetition and put it in the following column, these pattern of random number would represent chromosomes and we would have chromosomes, e1, e2, …. and so on in the

following column of the matrix and then we find the fitness of each chromosomes, find probability, perform selection and recommend which chromosomes to be taken in to the population. Based on the random number we came to know that the first random number recommends the chromosome 1 which is represented as

(T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12)

Because the selected random number lies between 0-0.342. Second random number recommends the chromosome 2 which is represented as

(T2→T4→T6→T8→T10→T12→T1→T3→T5→T7→T9→T11)

Because the random number lies between 0.342-0.671.The third random number recommends the chromosome 1 which is represented as

(T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12)

Because the selected random number lies between 0-0.342.So now we have the following member in our mating pool:

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

T2→T4→T6→T8→T10→T12→T1→T3→T5→T7→T9→T11

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

Now we will apply the one point cross over on these chromosome and will generate the new off springs.

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

T2→T4→T6→T8→T10→T12→T1→T3→T5→T7→T9→T11

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

When we apply one point crossover to the selected population then we get these offspring's

T1→T2→T3→T4→T5→T6→T7→T9→T11→T8→T10→T12

T2→T4→T6→T8→T10→T12→T1→T9→T11→T3→T5→T7

T1→T2→T3→T4→T5→T6→T7→T9→T11→T8→T10→T12

Table 3. Number of Modified Lines Covered by the Test Case

Test case	Number of modified lines
T1	2
T2	4
T3	1
T4	3
T5	2
T6	2
T7	5
T8	2
T9	4
T10	1
T11	0
T12	2

Suppose if the crossover probability is 0.3 then we select 2 chromosomes from the offspring and one from the parents based on the fitness function value.

This process is repeated certain fixed number of iterations, on repeating this procedure multiple times, we will get the nearly optimum solution as in Table 4.

7. Steps by Step Procedure for Genetic Algorithm

The GA mainly consists of five modules. The modules are GA Initialization, Fitness Evaluation, Selection, Crossover, and Mutation. Each module is described separately.

7.1. GA Initialization

In this module sample population is initialized. It is generated randomly or heuristically. Individuals are represented by a fixed-length string over a finite alphabet. Population is a collection of chromosomes. Each chromosome consists of genes in it. Gene type includes number of method calls, lower bound, upper bound, value pool value etc.

Here order is the priority of the test case, if the test case is to be executed first then the order of the test case will be n, where n is the number of test case, NML is number of lines modified, this information is matched with the test execution history to derive which test case covers how many numbers of modified lines.

E1, E2,.. are the chromosomes, they are made of some pattern of test case execution history, to generate this random pattern we use rand() present in stdlib of c language, one has to generate the random number such that it should be within one to N , or in other words if "K" the random number generated it should satisfy this condition K ≤ N, the other condition is that the number should not repeat, thus if we calculate the total number of possibilities then one will have to calculate the value of N X (N-1) X (N-2) X (N-3)….1 this value will be very large if N is large, thus genetic algorithm would much optimize the load of find such a possibilities.

Order	**NML**	E1	E2	E3	E4	E5
12	2	5	9	.	.	.
11	4	4	4	.	.	.
10	6	8	2	.	.	.
9	7	9	10	.	.	.
8	6	1	5	.	.	.
7	1	2	11	.	.	.
6	0	10	12	.	.	.
.	.			.	.	.

Table 4. Using genetic algorithm on the same data

Chromosomes	**Fitness Value**	**Normalized Value**	**Cumulative Probability**	**Selection Of Random Numbers**	**Recommendation**
T1->T2-> T3-> T4-> T5->T6-> T7->T8-> T9-> T10-> T11-> T11-> T12	196	196/573=0.342	0.342	0.3	Chromosomes e1
T2->T4->T6->T8->T10 ->T12-> T1->T3->T5-> T7->T9->T11	189	189/573=0 329	0.671	0.4	Chromosomes e2
T5->T6->T8-> T9->T12 ->T1-> T7->T11->T2-> T3->T4->T10	188	188/573=0.328	1	0.2	Chromosomes e1

7.2. GA Evaluation

Once the population is initialized or an offspring population is created, the fitness values of the candidate solutions are evaluated. This is where we attempt to identify the most successful members of the population, and typically we accomplish this using a fitness function. The purpose of the fitness function is to rank the individuals in the population.

The fitness calculation is done for each chromosome using the following formula

$$\textit{fitness value of each chromosomes} = \sum_{1}^{n} Order \times NML$$

Where n is the number of test case to be prioritized. In the above equation order is the priority of a particular test case, and NML is the number of modified lines, here one can find the order and number of modified lines of each test cases in a test

case pattern present in a chromosomes, addition of each of these order and number of modified lines give us the fitness value of a particular chromosomes.

Here for instance if one takes the first chromosome e1, then one has test case 5 scheduled to be executed first, test case 4 comes second thus, for first test case We take the value 5 and index it in the array of matrix, this gives as the order and number of the particular test case in column one and two, we find the product of order and number of modified line test case 5 and it comes out to be 48 as 8 x 6 then one can proceed with test case 4 it comes out to be 63 and then we add 48+63, this process continues till then end of all the test cases finally we get the fitness of chromosomes e1 and we calculate for e1-e5.

7.3. GA Selection

In the selection step we *choose* the individuals whose *traits* we want to install in the next generation. In the selection process typically we call the *fitness function* to identify the individuals that we use to create the next generation. In the biological world, usually two parents contribute chromosomes to the offspring.

Order	**NML**	E1	E2	E3	E4	E5
12	2	5	9	.	.	.
11	4	4	4	.	.	.
10	6	8	2	.	.	.
9	7	9	10	.	.	.
8	6	1	5	.	.	.
7	1	2	11	.	.	.
6	0	10	12	.	.	.
.	.			.	.	.

Of course, in software testing, we are free to use any combination of parents. For example, we are free to combine the traits of the top two, five, 10, or any other number of individuals.

	Order	**NML**
	12	2
	11	4
	10	6
	9	7
Index 5	8	6
	7	1
	6	0

There are various selection technique, random wheel selection technique, tournament selecting technique, we implement only the random wheel selection technique.

In order to perform random wheel selection we calculate the probability and cumulative probability of the population, the formula for calculating the probability and cumulative probability are:

$$probability\ of\ a\ chromosome = \frac{fitness\ of\ the\ particular\ chromosomes}{\Sigma(fitness\ of\ all\ chromosomes)}$$

random number generator to generate a new random number and check where the point lies in cumulative probability, the chromosomes with most fitness values covers the most sector in the wheel.

7.4. GA Crossover

Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring).There are many ways of accomplishing this, and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner (Goldberg, 2002).

There are many crossover methods, one better than another, namely, one point crossover, two point crossover, uniform crossover etc, uniform crossover is considered in many situations to be one of the best methods, but for the ease of implementation we considers only the one point crossover. Consider that the following two chromosomes (e1, e2) were selected to be the fittest amongst the five chromosomes. One also has the execution sequence of these two chromosomes.

In one point cross over one generates the a random number smaller than the number of test cases, then one can take that random number of point of crossover, we calculate the cross over probability, which tells us the amount of changes that will be done on the chromosomes.

E2	**E4**
T8	T3
T7	T4
T3	T1
T5	T10
T1	T12
T12	T11
T4	T9
T11	T8
T6	T7

T10	T2
T2	T5
T9	T6

Assume 8 be the number which is generated by the random functions, the cross over probability comes out to be 0.33 which means that 33% of the chromosomes will be changed following is what we get after cross over.

E2	**E4**
T8	T3
T7	T4
T3	T1
T5	T10
T1	T12
T12	T11
T4	T9
T11	T8
T7	T6
T2	T10
T5	T2
T6	T9

7.5. GA Mutation

While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution.

Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.

Mutation is done to bring a change in structure after crossover, and it is advised to perform mutation only after certain iteration, because genetic algorithm follows nature, and making changes unnaturally brings about an opposition to the nature.

E2	**E4**
T8	T3
T7	T4
T3	T1
T5	T10
T1	T12
T12	T11
T4	T9
T11	T8
T7	T6
T2	T10
T5	T2
T6	T9

For mutation one can generate random number based on the mutation probability and then the structure at those random numbers are changed.

E2	**E4**
T8	T3
T7	T4
T11	T8
T5	T10
T1	T12
T12	T11
T4	T9
T3	T1
T7	T6
T2	T10
T5	T2
T6	T9

Considering the above chromosomes where cross over is already performed, and suppose the mutation probability is 0.16 then one can generate two random numbers and then brings changes about those structure, if 3 and 8 are then number generated then the above chromosomes becomes.

The structure that is at the index 3, index 8 that are swapped as a process of mutation, it is believed to improve the fitness if mutation is done once in certain iteration and not all.

8. Pseudo-Code for Genetic Algorithm

```
Begin
T<-0
Initialise P(t)
while (not termination condition)
    Evaluate P(t)
    Select P(t+1) from P(t)
    Crossover P(t+1)
    Mutate P(t+1)
    t<-t+1
end while
end procedure
```

8.1. Evaluation Operation

Test info is an array that stores all the necessary information of a test case represents the chromosomes. Fitness is variable that stores fitness value of chromosomes. Fitar is an array that stores the fitness value of each chromosome. Order is the priority of test case. TID is the test case number we get from test case information.

```
while(e not 7)
Fitness<-0
Order starts from number of test cases
for (each number of test case)
TID<-testinfo[i][e]
Fitness<-fitness+ (order*testinfo[TID-1][1])
Order decremented by one
End for
Put the fitness value in fitar;
increment j
increment e
```

```
end while
```

8.2. Selection Operation

```
for(number of chromosomes times)
calculate the probability for each chromosomes ;
sum of probability of each chromosomes;
calculate the cumulative probability of;
end for
for (two parents)
do
until
generate a random number;
check where the number lies in roulette wheel
Convert the generated number such that it lies in between 0-1
for(the number of chromosomes times)
if(the number lies in between 0 and first cumulative probability)
break;
else if(check where it lies in cumulative probability)
break;
end for
check or number is already used:
 Set Check true;

for (j<-0;j<i; increment j)
if (if number is already used)
set check to false
break; //no need to check other elements of crom[]
end if
end while if check is not true
end for
```

8.3. Crossover Operation

```
for(the number of test case times)
if( until the point of cross over)
new matrix first column=elements of selected chromosomes
end if
else
do
until
initialize the n;
if(n crosses the number of test cases)
then set n to zero
set check true;
for(j from 0 to current index)
 if(current chromosomes element is already in the new matrix)
set check to false
break;
end if
end for
end while if check is not true
new matrix first column=element of selected chromosomes
end else
end for

//second child
for(the number of test case times)
if( until the point of cross over)
 new matrix second column=elements of selected chromosomes
end if
else
do
until
initialize the n;
if(n crosses the number of test cases)
then set n to zero
set check true;
for(j from 0 to current index)
 if(current chromosomes element is already in the new matrix)
set check to false
break;
end if
end for
end while if check is not true
new matrix second column=element of selected chromosomes
end else
end for
```

8.4. Mutation Operation

```
srand(time(NULL));
generate first random number
do
until
set check true
generate second random number;
if(the two random numbers are same)
set check to false
break;
end while if check is not true
//swap
Swap the execution order of selected random number for first child
Swap the execution order of selected random number for the second child
```

9. Performance Analysis

Generally in regression testing we use certain algorithms to get various information, there are regression test case algorithm to detect number of lines modified or number of lines covered, we take the advantage of the these information to find the best execution sequence of test case that would maximize the code coverage or that would cover maximum

modified lines, we consider a matrix to represent the test case information, the test case information are the test case id, test case order, number of lines modified or number of lines covered by a test case, and the chromosomes are set of test cases in some random execution order.

For performance analysis we use some random chromosomes it then uses a fitness function and checks how at an average is the fitness of each chromosomes, we observe that in the beginning or otherwise called first generation, at an average the fitness value of the chromosomes is very poor, in order to improve the fitness at an average it uses the genetic algorithm, the idea of genetic algorithm originated from Darwin theory of evolution, its main postulate being "the survival of the fittest", this algorithm mimics the nature and produces the best optimum solution, thus genetic algorithm does not grantee best solution, instead it gives the best optimum solution available.

Amongst many operations available in the genetic algorithm cross over and mutation are the two that is implemented, the two produces a fairly good outcome. There are many crossover methods, one better than another, namely, one point crossover, two point crossover, uniform crossover etc., uniform crossover is considered in many situations to be one of the best methods, but for the ease of implementation we consider only the one point crossover.

Table 5. First generation

E1	**E2**	**E3**	**E4**	**E5**
T5	T8	T9	T3	T2
T2	T7	T1	T4	T4
T7	T3	T8	T1	T6
T8	T5	T2	T10	T8
T9	T1	T7	T12	T10
T3	T12	T3	T11	T12
T1	T4	T6	T9	T1
T10	T11	T4	T8	T3
T12	T6	T5	T7	T 5
T11	T10	T12	T2	T7
T4	T2	T10	T5	T9
T 6	T9	T 11	T6	T11

While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution. The first generation has following chromosomes as in Table5.

The output which is produced by the chromosome has the fitness function as in Table 6.If the average fitness value of the chromosomes are found it comes out to be 190.6 fitness values. With above fitness value we search two best parents and perform cross over for fixed amount of times, for instance with five iteration we get the following output as in Table 7. The chromosomes fitness values are in Table 8.

Table 6. Fitness function

Chromosomes	E1	E2	E3	E4	E5
Fitness value	208	178	216	162	189

Table 7. Fitness values

Chromosomes	E1	E2	E3	E4	E5
Fitness value	208	216	202	206	196

Table 8. Final generation

E1	**E2**	**E3**	**E4**	**E5**
T5	T9	T9	T5	T9
T2	T1	T1	T2	T1
T7	T8	T8	T7	T8
T8	T2	T7	T8	T7
T9	T7	T12	T12	T11
T3	T3	T4	T4	T4
T1	T6	T11	T11	T12
T10	T4	T6	T6	T10
T12	T5	T10	T9	T 6
T11	T12	T2	T3	T5
T4	T10	T3	T10	T2
T 6	T11	T 5	T1	T3

Table 9. Fitness value

Iteration	1	2	3	4	5	6
Average Fitness	190.6	201.6	205	205.6	200	205.6

Now after the implementation of genetic algorithm if we find the average fitness value of the below execution sequence the fitness value comes out to be 205.6The best execution sequence of the chromosomes is as follows. On performing five iterations and finding the fitness value we get the following result as in Table 9.Plotting graph for the above result we get the following curve, which suggest the genetic algorithm does not always guarantee the answer as in Figure 2.

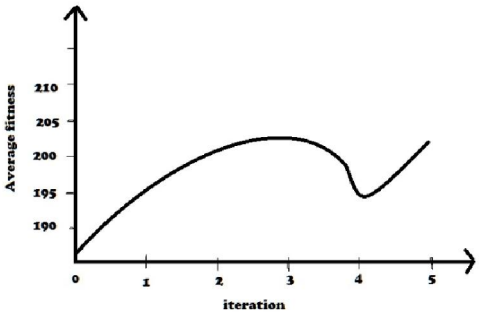

Figure 2: Fitness plot for each iteration

10. Conclusion and Future Enhancement

Here the genetic algorithm is applied on the test cases with their execution history. We used a fitness function which gives higher value if a test case covers more line of code, and a test case which has higher fitness value is provide higher priority in ordered sequence. When we applied genetic algorithm a large number of time we will get a nearly optimized solution. As we know that genetic algorithm does not always gives optimum solution, but if we run this algorithm fairly large number of time then we will get nearly optimum solution.

The input given to the genetic algorithm is a set of chromosomes and the chromosomes are set of test cases with the execution history, below is an instance of chromosome.

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→ T11→T12

We consider a random execution sequence generated by random number generator function available in stdlib library(c language) the sequence so generated becomes one chromosomes, we use five chromosomes, generates the fitness of each chromosomes, and then the average fitness value is found. In the first generation the average fitness value comes out to be 190.6, we use iteration value five as a fixed terminating condition, after the fifth iteration we find that the average fitness value of the population becomes 205.6 a much better one then the first generation. This means that the final population has a set of chromosomes, whose execution sequence is nearly the best optimum solution. We considers a random terminating value, we can perform analysis on bench mark problems and derive the terminating criteria by which we can find the least iteration value that will provide guarantee the near optimal solution.

References

1. Smith (2009). An empirical study of incorporating cost into test suite reduction and prioritization. In Proceedings of the 24th Symposium on Applied Computing, 2009.
2. T. Prem Jacob, Dr.T.Ravi,(2013) Regression Testing: Tabu Search Technique for Code Coverage, Indian Journal of Computer Science and Engineer-ing, Vol. 4, No.3.
3. Zhong (2008). An experimental study of four typical test suite reduction techniques. Information and Software Technology, 50(6).
4. Semantics Guided Regression Test Cost Reduction David Binkley, Loyola College in Maryland.
5. T.Prem Jacob, Dr.T.Ravi,(2013) Detecting of Soft-ware Source Code Defects using Test Case Prioritization Rules, 2nd ICLCT'13, London (UK).
6. Generic Chromosome Representation and Evaluation for Genetic Algorithms Kristian Guillaumier Department of Computer Science and AI, University of Malta.
7. Kapfhammer(2007). A Comprehensive Framework for Testing Database-Centric Applications. PhD thesis, University of Pittsburgh, Pittsburgh, Pennsylvania.
8. Prioritizing Test Cases For Regression Testing Gregg Rothermel, Member IEEE Computer Society.
9. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization SriramanTallam, Dept. of Computer Science. The University of Arizona Tucson.
10. T.Prem Jacob, Dr.T.Ravi,(2013),An Efficient Method for Regression Test Selection, International Journal of Software Engineering and Technology, ISSN 0974 – 9632.
11. Sampath(2008). Prioritizing user-session-based test cases for web applications testing. In Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation.
12. Venkatesh, Priyesh Cherurveettil, Thenmozhi. S, Balasubramanie. P. Predicting test effectiveness

using performance models in Life Science IT projects. *Life Sci J* 2012;9(4):96-100 (ISSN:1097-8135)

T.PREM JACOB received the B.E degree in Computer Science and Engineering from C.S.I Institute of Technology, Manonmaniam Sundaranar University, Nagercoil, India in 2004 and M.E degree in Computer Science and Engineering from Sathyabama University, Chennai, India in 2006, where he is currently working towards the Ph.D. degree in Computer Science and Engineering at Sathyabama University, Chennai, India. He is an Assistant Professor of Computer Science and Engineering in Sathyabama University and he has more than 7 Years of Teaching Experience. He has participated and presented many Research Papers in International and National Conferences. His area of interests includes Software Engineering, Data mining and Data warehouse.

Dr. T. Ravi, Principal of Srinivasa college of Engineering & Technology, Chennai. He has graduated in computer science and Engineering from Madurai Kamaraj University, Masters and Ph.D in computer Science and Engineering from Jadavpur University, Kolkata. He has more than 20 years of teaching experience in various engineering institutions in Tamil Nadu. More than 25 research papers are published in International & National Journals and conferences and also 5 text books are published through various publications. He is the Recognised Research Supervisor in Anna University and Sathyabama University Chennai and MS university, Tirunelveli.

6/20/2013