International Journal of Computer Applications (0975 – 8887)
Volume 75– No.8, August 2013

Analysis of Test Case Prioritization in Regression Testing using Genetic Algorithm

Pradipta Kumar Mishra
CUTM, Bhubaneswar
Odisha,India

B.K.S.S Pattanaik
GITA,Bhubaneswar
Odisha,India
Pin – 752054

ABSTRACT

Testing is an accepted technique for improving the quality of developed software with the increase in size and complexity of modern software products, the importance of testing is rapidly growing. Regression testing plays a vital role for software maintenance when software is modified. The main purpose of regression testing is to ensure the bugs are fixed and the new functionality that are incorporated in a new version of a software do not unfavorably affect the correct functionality of the previous version. So to revalidate the modified software, regression testing is the right testing process. Though it is an expensive process which requires executing maintenance process frequently but it becomes necessary for subsequent version of test suites. To evaluate the quality of test cases which are used to test a program.Testing requires execution of a program. In this paper it is proposed a new test case prioritization technique using genetic algorithm. The proposed technique separate the test case detected as severe by customer and among the rest test case prioritizes subsequences of the original test suite so that the new suite, which is to run within a time-constrained execution environment. It will have a superior rate of fault detection when compared to rates of randomly prioritized test suites. This experiment analyzes the genetic algorithm with regard to effectiveness and time overhead by utilizing structurally-based criterion to prioritize test cases.

Keywords

Regression testing, object-oriented, software testing, regression test selection, software maintenance, Genetic algorithm.

1. INTRODUCTION

It is a basic phenomenon that changes in software is one of the most foreseeable part when computer based system are made. So often it has been observed that the functionalities which were working in the previous version are still working in the new version also. With the existing errors and necessity for changing requirements propels the software to be reworked. Through the new uses of the previous version software, one can obtain new functionality that is not originally conceived in the requirements. To properly manage these changing concepts play a crucial role in the enduring efficacy of the software.

The basic purpose of regression testing is to revalidate the old functionality that was inherited from the previous version. Once the new functionality is added to the system then it can be accommodated by the distinctive software development process. The new version is required to behave exactly same in the previous version except the new included part. In other way, regression tests for a system may be perceived as partial operational requirements intended for the new version of a system.
Let's take an example of a sequence of time intervals during the life cycle of software which is depicted in the Fig. 1. It is already a known fact that the regression testing intervals take up a considerable portion of the system's life time for which the importance of regression testing cannot be overlooked. Due to time constraint a complete regression testing cannot be always possible for an updated version of the software. It may be in the software revalidation process one cannot completely omit or randomly reduce the regression testing interval.

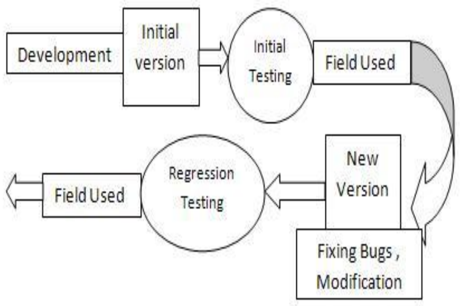


Fig. 1. Sequence of time intervals during a software's life cycle

When a software/program is modified, it should not only look at whether the modifications work properly or not but it should be checked whether there have been any adverse side effect in the unmodified parts of the software/program because even if a small change in one part of a program can affect with the unrelated part of the program. It may happen the modified program may yield correct results on specially designed test cases for modification testing whereas it may produce incorrect results on other test cases on which the original program produce right outputs. To ensure the modified program's effectiveness, the modified program is executed on all existing regression tests to ascertain that it still works the same way as the original program except the modified parts of the program where the change is expected.

Like other application programs, object oriented application programs also requires testing. Therefore for proper testing of object oriented software, specific type of testing classes is required. In class testing technique, in the first step, a sequence of methods with varying orders are invoked. Then after each sequence, it is verified whether the resulting state of the objects manipulated by the method is correct or not as proposed in [10], [11] and [12].

By utilizing a distinctive approach for performing class test, a test driver is used which invokes a sequence of methods. The test driver performs its task in different steps. In the first step, the driver performs setup tasks which is comprised of task like constructor routines calling as well as different initialization methods are utilized also. In the second step, the sequence of methods under test is invoked by the test driver. At the end, a special method called oracle method is invoked by the test driver to ascertain whether objects have attained proper states or not. As specified in [11], [13] when a class is modified it is required to retest the class itself as well classes derived by that class. As found in [13], [14] a function which has been effectively tested in segregation may not be sufficiently tested in combing with other functions for which it is suggested to test the application programs which use the modified class. But reality is that it is impossible to retest all such application programs. Therefore the testers should be well aware about the bottleneck. To overcome this problem, the testers can be able to reduce the associated risks by regression testing those application programs which test cost will be economical.

In the last two decades solutions to these problems for traditional programs have been proposed by researchers Fischer, Harrold, Laski, Leung, Prathar, and white in [15], [1],[16],[29],[42],[18] and [28] respectively. However less attention had given to testing of OO programs and regression testing of object oriented programs has not considered at all.

Many new concepts like inheritance, encapsulation, polymorphism and dynamic binding are introduced by object oriented model for software development process. Due to these new concepts a complex relationship occurs in between classes and their members.

As a result of which new testing problems find their place in the software field which are acknowledged by the different researchers as proposed in [24], [43], [38] and [35]. Along with that it also raises a tough challenge for conducting regression testing for object-oriented programs. Although the existing results can be functional to regression testing of member functions of a class at the unit and integration levels. They lack suitability for testing object oriented components at higher levels like a class, a group of classes, or class libraries. Basically the traditional approaches lack three measure difficulties while handling object oriented features.

i) The complex relationships and dependencies, such as inheritance, aggregation, and association properties which exist between different classes cannot be addressed by traditional approaches.
ii) As most traditional approaches are control flow model oriented whereas class models exhibit state dependant behavior which has various changing nature for which traditional approaches cannot be applied to the class testing.
iii) Traditional approaches use test stubs to simulate the modules that are invoked but in OO programs this is difficult and costly because it requires understanding of many related classes, their member functions and how the member functions are invoked in [38] and [27].

The organization of this paper is as follows. In section II, it is specified the existing regression testing technique. In section III, briefly identified some importance about previous related work. In section IV it is describe about some available soft computing approach. In section V it is elaborately discussed proposed model prioritization technique using genetic algorithm. It followed with conclusion .

2. EXISTING TEST CASE PRIORITIZATION TECHNIQUES

As the regression testing is quite expensive for reducing the cost, researchers have done many works other than test selection technologies. The test case prioritization technology is addressed in [25],[26] and [21]. They prioritize the test cases according to certain measures. After which in the regression testing cycle, the test cases will be used to test the modified program P' in accordance with the same order.So that the “better” test cases can able to run first. The purpose of the prioritization is either to increase the rate of fault detection or increase the rate of code coverage.
Here it is taken example-1 for better understanding about this technique.

Example 1
Let T1, T2, T3 and T4 are four test cases.
Suppose, T1 has the coverage of 60%
T2 has the coverage of 15%
T3 has the coverage of 35% and
T4 has the coverage of 45%
According to the second goal, by applying such technology, the test cases can be run in the order of T1, T4, T3, and T2. Like that according to the first goal, the order of the four test cases will depend on their ability to expose the fault.
As it is proposed in [30] test case prioritization techniques can be divided into three categories as depicted in Fig. 2.

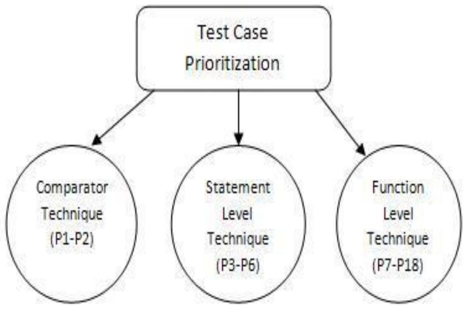

Test Case Prioritization
Comparator Technique (P1-P2)
Statement Level Technique (P3-P6)
Function Level Technique (P7-P18)

Fig. 2. Phases of Prioritization Test Case Techniques

But there are 18 different test case prioritizations techniques which are numbered from P1 to P18 in these three categories which are discussed below.

2.1 Comparator Techniques

Random ordering (P1): Here the test cases in test suite are randomly prioritized.
Optimal ordering (P2): In this case the test cases are prioritized to optimize rate of fault detection.

2.2 Statement level Techniques (Fine Granularity)

Total statement coverage prioritization (P3): Here the test cases are prioritized in terms of total number of statements according to sorted order of coverage achieved. If test cases are having same number of statements they can be ordered pseudo randomly.

Additional statement coverage prioritization (P4): It is likely similar to total coverage prioritization which depends upon feedback about coverage attained to focus on statements that are not yet covered.

Total FEP prioritization (P5): Here prioritization is done on the probability of exposing faults by test cases. Mutation analysis is used for approximation of the Fault-Exposing-Potential (FEP) of a test case. **Additional FEP prioritization (P6):** In this case, the total FEP prioritization is extensive to additional FEP prioritization due to the total statement coverage prioritization is also extended to additional statement coverage prioritization.

2.3 Function level Techniques: (Coarse Granularity)

Total function coverage prioritization (P7): Though it is similar to total statement coverage but here functions are used. As it has possessed coarse granularity level so the process of collecting function level traces is cheaper as compared to the process of collecting statement level traces in total statement coverage.
Additional function coverage prioritization (P8): Though it is similar to additional statement coverage prioritization with a slight difference i.e. it considers function level coverage instead of statements.

Total FEP prioritization (function level) (P9): It is equivalent to Total FEP prioritization with one difference that is instead of using statements, functions are used here.

Additional FEP prioritization (function level) (P10): This technique also similar to additional FEP prioritization. Instead of using statements it is using functions.

Total Fault Index (FI) prioritization (P11): In this technique a measurable software attribute called fault proneness is used.

Additional Fault Index prioritization (P12): Here, the total function coverage prioritization is extended to additional function coverage prioritization and the total Fault Index prioritization is extended to additional Fault Index prioritization similarly also.

Total Fault Index with FEP coverage prioritization (P13): It combines both total Fault Index and FEP coverage prioritization to achieve a superior rate of fault detection.

Additional Fault Index with FEP coverage prioritization (P14): Here also, the total function coverage prioritization is extended to additional function coverage prioritization as well as the total Fault Index with FEP coverage prioritization is extended to Additional Fault Index with FEP coverage prioritization in a similar manner.

Total Diff prioritization (P15): It is similar to Total Fault Index prioritization but here, the total FI prioritization requires collection of metrics whereas total Diff prioritization requires only the calculation of syntactic differences between the program and the modified program. Diff means that merely syntactic differences are given consideration.

Additional Diff prioritization P16: Here the total Diff prioritization is extended to additional Diff prioritization in the same manner as the total function coverage prioritization is extended to additional function coverage prioritization.

Total Diff with FEP prioritization (P17): It is precisely similar to total FI with FEP coverage prioritization but it is dependent upon changed data that is derived from Diff.

Additional Diff with FEP prioritization (P18): The total Diff with FEP prioritization is extended to additional Diff with FEP prioritization in a similar approach as Total function coverage prioritization is extended to additional function coverage prioritization.

3. RELATED WORK

Different research work on regression testing proposed by various researchers have briefly analyzed in this section.
Harroldet al. in [37] proposed a technique which emphasizes on changing effects within a module when the software is modified. Data flow graphs are used for identifying the pretentious definition use pairs. Also sub paths are used not necessarily. As, retesting is performed here for affected define use paths and newer paths, so the test effort is reduced comparatively which is an added advantage of this technique. According to [39] the technique was expanded so that it can also be used to identify affected procedures at inter procedural level.

The three other researchers named Laski, Benedusi, and Prather have also proposed techniques in [29], [17] and [18] which is based upon control flow graph techniques that works on for both procedures and functions for identifying the affected control paths in a module.
H. K. J. Leung and L. White have introduced firewall concept which enclosed the affected modules that arises due to module modification in [42]. A call graph is defined according to the concept of a control related firewall. Here test effort is reduced by augmenting the retesting factor of modules and links in the firewall of the changed module.
In [20] significant test approaches are introduced like top-down, bottom-up, and sandwich approach. In the approach the tester performs selections to minimize test effort and cost.
The researcher L. White in [28] introduced data related firewall. It was based on a data flow graph which enclosed all affected modules where coupling arises due to global data.
In [18], an adaptive path prefix software testing strategy was proposed where previous test paths were utilized as a guide in the selection of consequent paths. In this approach, branch coverage is ensured and a less number of computational resources were utilized.

In [24] an incremental object oriented testing methodology was presented according to class inheritance hierarchy. In this approach, the researcher suggested about that testing of base class in proximity before derived class such that the base class' test cases and relevant information can be reused in testing the derived classes.

In [19] and [18] to fulfill coverage criteria factor of regression testing with different types of retesting criteria were proposed where path coverage of a procedure or function is utilized.

In [19] researcher Fischer discussed a test case selection strategy that deals with the set covering problem syndrome. The indispensable plan was to use the 0-1 integer programming models where minimum test cases were found out and that covers one of the path criteria especially in case of unit regression testing.
In [40] a specific 0-1 integer programming model was also used on a test matrix for minimizing the test efforts in functional regression testing.
In [23] a retest strategy was proposed by Leung and White to perform corrective regression testing. Here, they have planned to outlook the regression testing as collected of two sub problems i.e. the test selection problem and the test plan update problem. As a result the re-testing process is alienated into two phases i.e. test classification and the test plan update. In the classification, the existing test cases are classified into three tests i.e. i) reusable tests, ii) obsolete tests and iii) re-testable tests. In the new regression test plan the re-testable test cases and new test cases are considered as tests.

The existing methods can be functional to regression testing of member functions as proposed in [41]. Conventionally testing uses test stubs and drivers for simulating the calling functions and called functions.
But they are difficult as well as costly in object oriented testing as the tester is required to understand a set of member functions and classes before constructing a driver or stub.

To reduce the test effort, a technique was presented in [22] and [24] which was required to test subclasses. But the drawback of this approach is that, it requires the hand analysis ought to be performed and it does not reuse test cases that are associated with the parent class.

In the design of test suites for derived classes, three researchers called Harrold, McGregor, and Fitzpatrick presented a technique in [24] where reusing testing information associated with a parent class was utilized. But this technique does not deal with the problems in regression test selection for modified classes or derived classes, or problems in test selection for modified object oriented application software. It does not overcome problems of that software which use modified classes. Apart from that, their technique reuses test cases which are based on their association with modified methods and attributes.

Kung et al. and Hsia et al. have presented a technique in [22], [19] and [16] for selecting regression tests for class testing which is based on the firewalls concept that was presented originally in [42] first for procedural software and later applied for object oriented software [15]. They have defined a concept called construction of object relation diagram which establishes static relationship among classes. This relationship comprises inheritance, aggregation i.e. use of composite objects and association which deals with data dependence, control dependence or message passing relationship among classes. The object relation datagram instruments code to account the classes which are exercised by test cases. The firewall for a class C is defined as the set of classes which are directly or transitively dependent on C with the virtue of all three properties i.e. inheritance, aggregation, and association in object relation datagram concept. When the C class is adapted, the object relation datagram technique selects almost all test cases which were determined through instrumentation to put into effect, one or more classes within the firewall for C.

4. SOFT COMPUTING APPROACHES

There are several steps involved in order to automate a test case. They are automation of test case selection criteria, test case minimization, test execution and test case evaluation

4.1 Genetic Algorithm

Basically, Genetic Algorithms are adaptive heuristic search algorithm based on the evolutionary ideas of expected collection of genetics and Genetic algorithms are based on the mechanics of natural selection and natural genetics [8]. Genetic algorithms have been applied in different areas such as machine learning, search and optimization. These algorithms have been considered an efficient and robust method for solving complex problems [9].
Based on Natural Selection, a genetic algorithm uses the following operators:

- Selection
- Reproduction
- Mutation

Selection: It is the process of picking out a suitable individual from the population. Suitable individuals are individuals with a good fitness. Selection at times faces problem called **crowding**. Crowding occurs when the fit individuals are selected quickly to reproduce. Which leads to a large percentage of the entire population looking very similar. Diversity in the population is greatly reduced and may hinder the long-run progress of the algorithm.
Reproduction: During reproduction new chromosomes are created out of existing chromosomes in the population.
Mutation: The mutation operator modifies one or several genes' value (e.g. if an individual is a bit string, mutation means changing a 1 to 0 and vice versa). The reproduction and crossover operators are so powerful in improving the search that the mutation operator usually plays a secondary role, i.e., it modifies the value of the test methods.
When there are these three operators and the fitness function, a genetic algorithm can be easily formulated as follows:

```
1.Randomly initialize population(t)
2. Determine fitness of population(t)
3.repeat
  i. Select parents from population(t)
  ii. Perform crossover on parents creating
     population(t+1)
  iii. Perform mutation of population(t+1)
4. Replace old with new population as the new
   generation.
5. Test problem criterion.
6. Back to step 2.
```

Therefore by using the genetic algorithms, it can generate better test cases. This improves the difficulty of having to run many test cases against mutants. This shows that, genetic algorithms improve the generation of effective test cases. This way they also reduce the cost as well as time of executing the test cases over mutated programs .
There are two popular swarm inspired methods in computational intelligence areas: Ant colony optimization (ACO) and particle swarm optimization (PSO). ACO was inspired by the behaviors of ants.

4.2 Particle swarm optimization (PSO):

Basically it is a population based stochastic optimization technique that incorporates swarm intelligence with respect to socio-behaviour phenomenon. It shares many similar features of evolutionary computation techniques like Genetic Algorithms (GA). The system is initialized with a population of random solutions which is known as particles and searching for optima is done by updating generations. However, unlike GA, PSO does not utilize genetic operators like crossover and mutation. In PSO, the potential solutions (particles), fly/move through the problem search space by following the current optimum particles or solutions.

4.3 An artificial neural network (ANN):

It is an analysis paradigm which is modeled after human Brain's powerful computing ability. Though a set of different types of algorithms used, one of the best algorithm used to train the artificial neural network is known as Back Propagation algorithm which is performed effectively. A significant research works are going on which utilizes evolutionary computation techniques and in turn it helps to evolve more and more aspects of artificial neural networks. Different models of ANN employs soft computing platform for solving a lot of applications in diversified areas.

5. PROPOSED PRIORITIZATION TECHNIQUE

The proposed prioritization technique is based on both testing time and potential code coverage information to intelligently reorder a test suite using Genetic Algorithm as shown in Fig. 3. Our prioritization algorithm reorders the tests in any sequence that maximizes the suite's ability to cover the code.

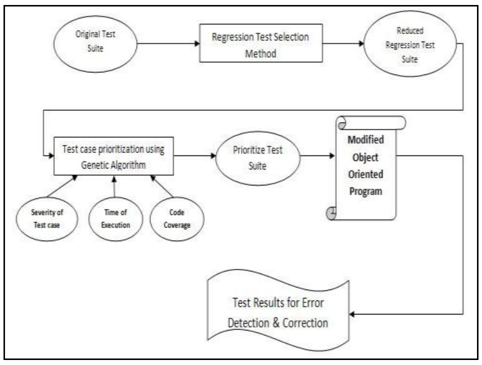

Original Test Suite
Regression Test Selection Method
Reduced Regression Test Suite
Test case prioritization using Genetic Algorithm
Prioritize Test Suite
Modified Object Oriented Program
Severity of Test case
Time of Execution
Code Coverage
Test Results for Error Detection & Correction

Fig. 3. Proposed model from test case reduction to test result generation.

When customer detected some of the test case as sever then that test case are automatically part of the new test suit. Only it is to detect the remaining test cases according to code coverage and time. For example if there are 100 test cases and the customer choose<T9,T6,T73,T60,T89> as sever than these test cases must be present in our new test suit after prioritization. From the remaining test cases it is to choose them which take less time and cover more code. The remaining number of test case to be chosen must be pre decided. which is depicted in Fig. 4.

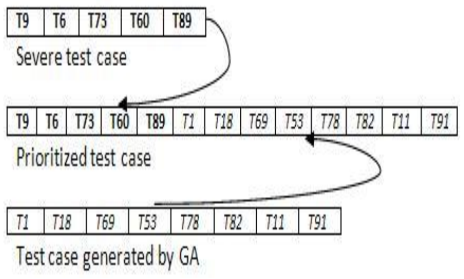

T9 T6 T73 T60 T89
Severe test case
T9 T6 T73 T60 T89 T1 T18 T69 T53 T78 T82 T11 T91
Prioritized test case
T1 T18 T69 T53 T78 T82 T11 T91
Test case generated by GA

Fig. 4. Test case prioritization based on Severity, Code coverage & Time using Genetic Algorithm

Here utilization of Genetic algorithm in the proposed prioritization technique to select among the left test case after severity test case is identify by customer. It is to first record the execution time of each test case, because time constraint could be very short. Test case execution times must be exact in order to properly prioritize. Only the execution time of the test case is included in the recorded time and not that of class loading. Timing information is additionally includes any initialization and shutdown time required by a test. Inclusion of initial time and shutdown time is necessary because these operations can greatly increase the execution time required by the test case. The program *P* and each Test case in a test suite are input into the genetic algorithm, along with the following user specified parameters:

- Test suite – *T*
- Number of initial population-n
- Number of test suites to be created - *s*
- Maximum iterations – *dmax*
- Crossover probability - *pC*
- Mutation probability - *pM*
- Initial Population I

Because two test cases may cover all or part of the same code, one cannot simply calculate the coverage of each test case and sum the products of the time required by each test case and the associated coverage amounts. Rather the secondary fitness measurement is partially calculated by summing the products of the test case time and the incremental coverage and algorithm is shown in the Fig. 5.

```
Algorithm
GAPRIORITIZE(T,s,n, dmax,pc,pm)
Input: program
Initial Papulation P
Output : Minimum fitness combination of test case
i ← 1
repeat
        P_i ← null;
        repeat
                P_i ← P_i U { Randomly from T }
        until | P_i |=S
i ← i+1
until i=n
g ← 1
repeat
        i ← 1
        repeat
                F_i ← CalculateFitness(P_i)
                i ← i+1
        until i ← n
        P_1 ← ChooseParent(P_[random()]).
        P_2 ← ChooseParent(P_[random()]).
        C_1, C_2 ← Crossover(pC, P_1, P_2)
        C_1 ← Mutation(pM, C_1)
        C_2 ← Mutation(pM, C_2 )
g ← g+1
until g=dmax
δmin ← FindMinFitnessTuple(P,F)
return δmin
```

Fig. 5. Genetic Algorithm For prioratisation

5.1 Fitness function

The calculation of the fitness function is broken up into three parts. The primary component of the fitness function is calculated by measuring the test adequacy of the final test suite prioritization. In these experiments, this value was multiplied by 100 to mark the total adequacy as the principle measurement. Next, favor is given to test suites ordered in such a way that test cases most likely to reveal faults in the program under test are executed first. Coverage versus time are considered. Because two test cases may cover all or part of the same code, one cannot simply calculate the coverage of each test case and sum the products of the time required by each test case and the associated coverage amounts. Rather the secondary fitness measurement is partially calculated by summing the products of the test case time and the incremental coverages. For example, suppose P = <T1, T2, T3> where T1 takes 5 seconds, T2 requires 3 seconds, and T3 needs 1 second. Assume <T1>, <T1, T2 >, and <T1, T2, T3> have calculated coverages of 0.3, 0.4, and 0.6 respectively. Then the principle component of the fitness value is Fp = 0.6 *100 = 60. Next, favor is given to faster fault detection. In order to weight the prioritizations appropriately, the amount of code covered by <T1>, <T1; T2>, and finally <T1; T2; T3> related to time must all be considered. The secondary fitness measurement based on the prioritized test suite, Fst , is calculated as follows:

$$Fst(P) = (5\ X .3) + (3\ X .4) + (1\ X. 6)$$
$$= 3.3$$

Fst is then compared to the highest value Fst could have. This will be called the optimal secondary fitness, Fso. Because test suites with tests that are likely to find the most faults first are weighted, the optimal secondary fitness would be the fitness of a prioritization whose first test covers all code for that test suite. For example, in the example above, if T1 covered 100% of all code covered by T, Fso = .6(5+3+1) = 5.4. Fst is compared to the best value it could be, Fso , by dividing the two results to give the secondary coverage. Hence, the secondary fitness is

$$Fs(P) = Fst(P) \div Fso(P)$$
$$= 3.3 \div 5.4$$
$$= .61$$

Adding together the fitness values, the total fitness is obtained. Therefore, this example has a final fitness of

$$F(P) = Fp(P) + Fs(P)$$
$$= 60 + 0.61$$
$$= 60.61$$

Code coverage by different case is depicted in Fig. 6.
Example of calculation of fitness function
Coverage:

{T1}	=.32
{T1,T2}	=.40
{T1.T2,T3}	=.41
{T1.T2,T3,T4}	=.55
{T1.T2,T3,T4,T5}	=.60

Primary Fitness=.60 X100=60
Secondary Fitness
=5*.32+2*40+1*.41+3*.55+4*.60/.60*(5+2+1+3+4)
=6.86/9=.76
Total fitness=60+.76=60.76

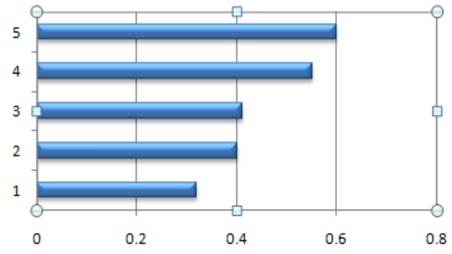


Fig 6. Code coverage by diff test case

5.2 The Initial Population

Each individual in the initial population should contain number of test case decided to take which is equal to total test case decided to take minus number of test case which are severe . First, randomly test cases, T1, T2, T3, T4,…., TS are randomly generated for individual P1 where s is total number of test case to be selected by GA and initially P1 an empty Vector. The fitness, which we have discuss above added to P1. Again another test case which is not already in Pi is chosen, and fitness is again calculated. This continues until n number of initial unique population created.

5.3 Selection

As just mentioned, the "optimal" test suite prioritization teeters on the edge of going over the designated fitness value. Thus, it is important to keep the best individuals from one generation to the next in case some slight addition or change immediately invalidates them. For this reason, two individuals were kept per generation using an elitist selection strategy. The remaining individuals needed to be used for reproduction were selected using a roulette wheel selection technique or using random selection technique.

5.4 Crossover

After two individuals are selected for reproduction, a random number R1 between 0 and 1 is chosen using random method. If R1 is less than the user-given value for *pC*, the crossover operator is applied. Otherwise, the parent individuals are unchanged and await the next step mutation. If crossover is to occur, another random number R2 from 0 to the number of genes of the smallest individual is selected as the crossover point. The subsequences before and after the crossover point are then exchanged to produce two new offspring as seen in **Fig 7. As another example, consider the individuals P1 = <T1; T3; T6; T4> and P2 = <T2; T3; T5; T9; T4>.**

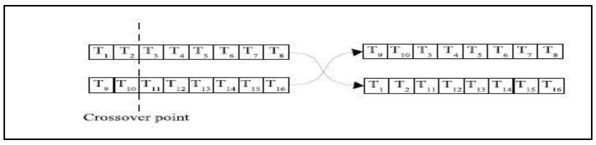


Fig 7. Example of Crossover

If 1 is picked, P1 and P2 could be crossed over after the first locus in each to produce the two offspring P1 = <T1; T3; T5; T9; T4> and P2 = <T2; T3; T6; T4>. A crossover selection is depicted in Fig. 7. The first issue arises if crossover causes two of the same test cases to be in the same individual. Although a test case may be run more than once in a test suite, there is no benefit to executing it again. Because the original test suite was independent, The first issue arises if crossover causes two of the same test cases to be in the same individual. Although a test case may be run more than once in a test suite, there is no benefit to executing it again. Because the original test suite was independent, a second execution would produce exactly the same result as the first execution and would cover no new code. Instead of including this test case, another random test not in the current individual is selected from the original pool of test cases. If the individual already includes all tests, no additions are made.

5.5 Mutation

In these experiments, individuals are subject to mutation at each locus with probability *pM*. If a random number R3 is drawn such that R3 is less than *pM*, a new test not included in the current individual is randomly chosen from the original pool of test cases as demonstrated in Fig. 8.

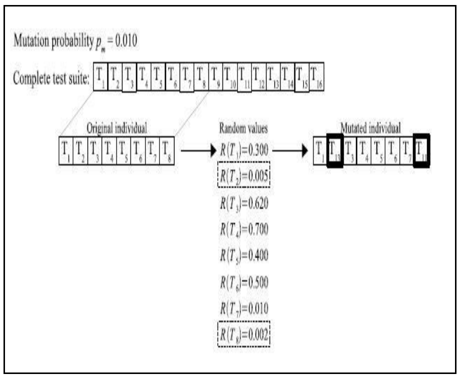


Fig. 8. Mutation was performed as displayed above. At each locus, a random number R3 is selected. If R3 is less than *pM*, the individual is mutated at that point.

5.6 Empirical Evaluation

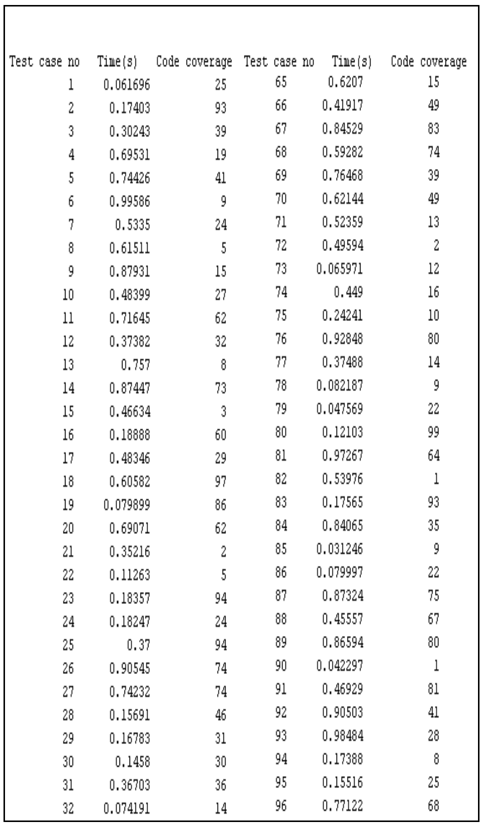
The primary goal of this paper's empirical study is to identify and evaluate the challenges that are associated with time-aware regression test suite prioritization. The goals of the experiment are as follows:

1. Identify the execution time of each test case.
2. Find out code coverage of test case to be prioritized.
3. Generate using GA the remaining test case after severe test case have been detected.

To practically implement our algorithm we have configure JUnit, Ant and Eclipse Emma to work with Eclipse editor. First the execution time of all the test case are determined by running all the test case using ant make file in JUnit environment. Eclipse Emma is used to find out code coverage of each test case. The summary of all test case is given in table1.

Once these value are determined next implementation of the algorithm using met Lab to find out remaining test case for the test suit after severe test case is added. Below is the met lab program code of implementation. In met lab implementation. Combination of test case as initial population are taken. Calculate fitness value for each population, do crossover and mutation with given mutation probability. This sequence is repeated for given number of generation. A Test case number,time of execution, code coverage derived by JUnit and Eclipes Emma taken for GA implementation is shown in table I.

Table I
Test case number,time of execution, code coverage derived by JUnit and Eclipes Emma taken for GA implementation.



Test case no	Time(s)	Code coverage	Test case no	Time(s)	Code coverage
1	0.061696	25	65	0.6207	15
2	0.17403	93	66	0.41917	49
3	0.30243	39	67	0.84529	83
4	0.69531	19	68	0.59282	74
5	0.74426	41	69	0.76468	39
6	0.99586	9	70	0.62144	49
7	0.5335	24	71	0.52359	13
8	0.61511	5	72	0.49594	2
9	0.87931	15	73	0.065971	12
10	0.48399	27	74	0.449	16
11	0.71645	62	75	0.24241	10
12	0.37382	32	76	0.92848	80
13	0.757	8	77	0.37488	14
14	0.87447	73	78	0.082187	9
15	0.46634	3	79	0.047569	22
16	0.18888	60	80	0.12103	99
17	0.48346	29	81	0.97267	64
18	0.60582	97	82	0.53976	1
19	0.079899	86	83	0.17565	93
20	0.69071	62	84	0.84065	35
21	0.35216	2	85	0.031246	9
22	0.11263	5	86	0.079997	22
23	0.18357	94	87	0.87324	75
24	0.18247	24	88	0.45557	67
25	0.37	94	89	0.86594	80
26	0.90545	74	90	0.042297	1
27	0.74232	74	91	0.46929	81
28	0.15691	46	92	0.90503	41
29	0.16783	31	93	0.98484	28
30	0.1458	30	94	0.17388	8
31	0.36703	36	95	0.15516	25
32	0.074191	14	96	0.77122	68

After running the metlab implementations with the detail given in Fig. 7 it gives the following sequence of test case as prioritized test case and the graph is given in Fig 8.

```
15-Jun-2011 22:45:02
popsize = 100 mutrate = 0.15
#generations=50best fitenss value=52.4799
best solution
10  78  77  46  25  28  36  54  46  17  50  58  15  73   2  15  38  31  21  37
each test suit is represented by 20 test case
```

Prioritize test suite generated by given algorithm

5.7 Experiments and Results

Experiments are run in order to analyze (i) the effectiveness and the efficiency of the genetic algorithm. In order to compare this test suit with

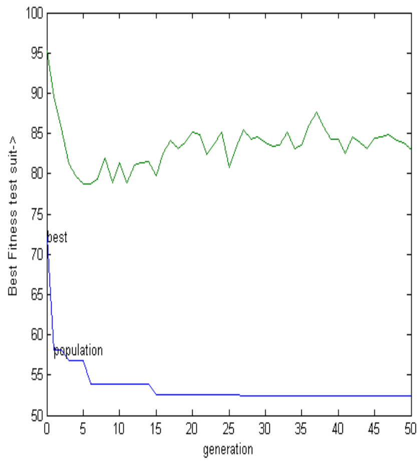


Fig. 9. Graph showing generation and best test suit

other available prioritization technique. The “random test case prioritization” were choosen .

Here is done an experiment by randomly taking test case from the test case given in Fig 10.

```
Random test case:
36  69  77  97  15  87  15  13  47  44  54  17  17  42  92  21  82  49   3  71
```

Fig. 10. Prioritize test suite generated by the given algorithm

The experiment show that test case generated by GA takes less time as compared to test case generated by random test, goal oriented and path oriented test case prioritization, which is depicted in Fig 11,Fig 12 and Fig 13.

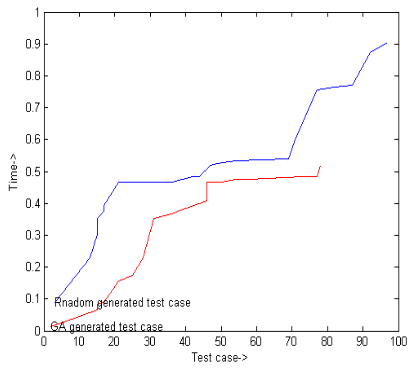


Fig. 11. Time of execution comparision between random test case prioritization and given algorithm

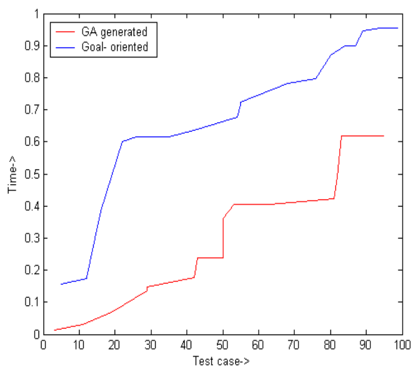


Fig. 12. Time of execution comparision between goal oriented case prioritization and given algorithm

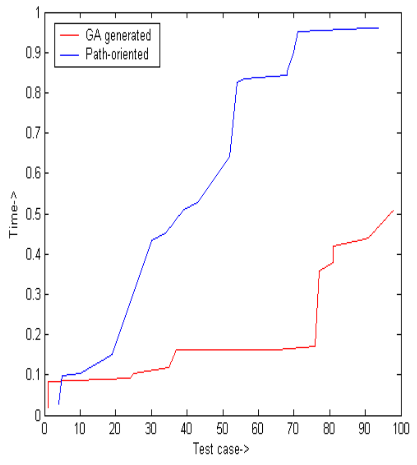


Fig. 13. Time of execution comparision between path oriented case prioritization and given algorithm

The experiment further show that the code coverage by test case generated by given algorithm is more than that of the test case generated by random test, goal oriented and path oriented test case prioritization which is depicted in Fig. 14, Fig 15, Fig 16.

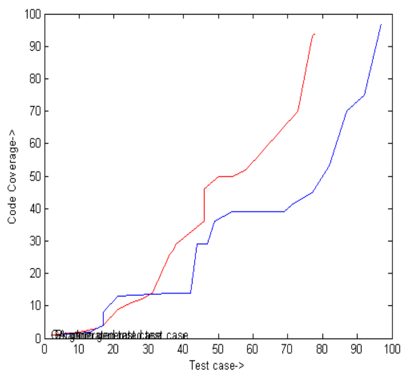


Fig. 14. Code coverage comparision between random test case prioritization and given algorithm

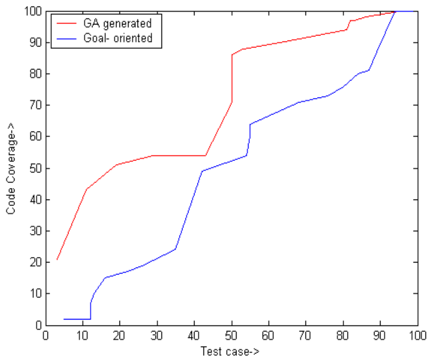


Fig. 15. Code coverage comparision between goal oriented test case prioritization and given algorithm

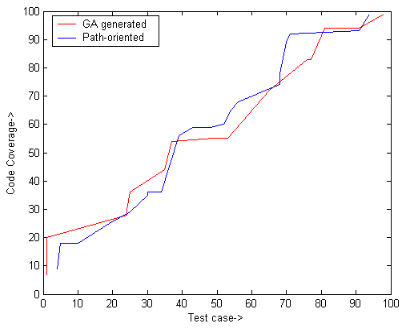


Fig. 16. Code coverage comparision between path oriented test case prioritization and given algorithm

6. CONCLUSIONS

In this paper it is described to generate test cases which are severe as per customer, takes less time and cover more code. Here it is compared test case generated by GA with test case generated by goal oriented, random and path oriented test prioritization technique and found some better result in terms time and code coverage.

7. REFERENCES

[1] R.A. DeMillo, R.J. Lipton, F.G. Sayward(1978), "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE *Computer*, Vol. 11, No. 4, pp. 34-41.

[2] R.G. Hamlet (1977), "Testing Programs with the Aid of a Compiler",IEEE Transactions on Software Engineering, Vol. 3, No. 4, pp. 279-290.

[3] David Banks, William Dashiell, Leonard Gallagher, Charles Hagwood, Raghu Kacker and Lynne Rosenthal: "*Software Testing by Statistical Methods Preliminary Success Estimates for Approaches based on Binomial Models",* Coverage Designs, Mutation Testing, and Usage Models

[4] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayword (1979), "Mutation analysis," Tech. Rep. GIT-ICS-79/08, School of Inform. and Computer. Science, Georgia Institute. of Technology, Atlanta GA

[5] William E. Howden (1982), "*Weak mutation testing and completeness of test sets*", IEEE Trans. On Software Engineering, Vol. SE-8,pp. 371-379.

[6] William E. Howden (1987), *"Functional Programming Testing and Analysis." , New York: McGraw-Hill.*

[7] W. M. Spears and V. Anand, "A study of Crossover Operators in Genetic Programming," *Proc. 6th Int. Symp. On Methodologies for Intelligent Systems*, 1991.

[8] C. C. Michael, G. McGraw, M. A. Schatz, "Generating Software TestData by Evolution," *IEEE Trans. on Software Engineering*, Vol. 27, No.12, pp. 1085-1110, 2001.

[9] M. D. Smith and D. J. Robson, \Object-oriented programming | the problems of validation," Proc. IEEE Conference on Software Maintenance | 1990. pp. 272 { 281.

[10] C.D. Turner and D.J. Robson. The state based testing of object oriented programs. In Proc. Of the Conf. on Software Maintenance, 1993, p. 302-11, Sept'1993.

[11] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM, Transactions on Software Engineering and Methodology 2(3):270–285, July 1993.

[12] J. Lee and X. He, A methodology for test selection," Journal of Systems and Software, Vol. 13, pp. 177 - 185, 1990.

[13]K.F. Fischer, F. Raji and A. Chruscicki, "A Methodology for Re-Testing Modi_ed Software", National Telecomms. Conf. Procs., pp. B6.3.1-6, Nov. 1981.

[14] K.F. Fischer, "A Test Case Selection Method for the Validation of Software Maintenace Modi_cations", IEEE COMPSAC 77 Int. Conf. Procs., pp. 421-426, No. 1977.

[15] H.K.N Leung and L. White, "Insights into regression testing", Proc. Conf. Software Maintenance, pp. 60-69, Miami, FL, Oct. 1989.

[16] Ronald E. Prather and J. Paul Myers, JR, "The Path Prefix Software Testing Strategy", IEEE Transactions On Software Engineering, Vol. SE-13, NO. 7, July 1987.

[17] F.J. Weyukar. Axiomatizing software test data accuracy. IEEE transactions on Software Engineering SE-12(12):1128-38 Dec'1986.

[18] K.F. Fischer, "A Test Case Selection Method for the Validation of Software Maintenace Modi_cations", IEEE COMPSAC 77 Int. Conf. Procs., pp. 421-426, No. 1977.

[19] P. Coad and Yourdon, E. \Object-oriented Analysis." Yourdon Press, 1990.

[20] G. Rothermel and M. J. Harrold, A Safe, Efficient Regression Test Set Selection Technique, ACM Transactions on Software Engineering and Methodology, V.6, no. 2, April 1997, pages 173-210.

[21]D.Hoffman and C.Brealey. Module test case generation. IN Proceedings of the Third Workshop on Software Testing, Analysis, and Verification, pages 97-102, December 1989..

[22]M. J. Harrold and M. L. Soa, "An incremental approach to unit testing during maintenance", Proc. Conf. Software Maintenance, pp. 362-367, Phoenix, 1988.

[23] C.D. Turner and D.J. Robson. The state based testing of object oriented programs. In Proc. Of the Conf. on Software Maintenance, 1993, p. 302-11, Sept'1993.

[24]G. Rothermel, R. Untch, C. Chu, and M.J. Harrold. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10):929–948, October 2001.

[25]Wei-Tek Tsai, Xiaoying Bai, Ray Paul, Lian Yu. Scenario-Based Functional Regression Testing, , COMPSAC 2001

[26] L. White and H.K.N. Leung, "A Firewall Concept for both Control-Flow and Data-Flow in Regression Integration Testing", Proc. Conf. Software Maintenance, pp. 262-271, 1992.

[27] Janusz Laski and Wojciech Szermer, "Identi_cation of Program Modi_cations and its Applications in Software Maintenance", Proc. Conf. Software Maintenance, pp.282-290, 1992.

[28] R. Gupta, M.J. Harrold and M.L. Soffa, "An Approach to Regression Testing Using Slicing", Proceedings of the Conference on Software Maintenance, 1992, pp. 299-308.

[29] M. Dyer, The Cleanroom Approach to Quality Software Development, Wiley, New York, New York, 1992.

[30] G. Booch, Object-Oriented Design with Applications" Redwood City, Calif.: Benjamin/Cummings, 1991.

[31] T.Y. Chen and M.F. Lau. Dividing strategies for the optimization of a test suite. Information Processing Letters, 60(3):135–141, March 1996.

[32] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In Proceedings of the Twelfth International Conference on Testing Computer Software, pages 111–123, June 1995.

[33] D. Kung, J. Gao, P Hsia, Y. Toyoshima, C. Chen, Y-S. Kim, and Y-K. Song. Developing an object oriented software testing and maintenance environment. Communications of the ACM, 38(10):75-87, Oct'1995

[34] N. Wilde and R. Huitt, \Issues in the maintenanceof object-oriented programs," University of West Florida and Bell Communications Research, 1991.

[35] G. Rothermel, M.J. Herrold, J. Dedhia. Regression Test selection for C++ software. Journal of software testing, verification and reliability, V. 10, No. 2, June 2000.

[36] G. Rothermel. Efficient, effective regression testing using safe test selection techniques, Technical Report 96-101, Clemson University, Jan' 1996.

[37] B. Beizer, Software Testing Techniques," 2nd ed., Van Hostrand Reinhold, 1990.

[38] G. Rothermel and M. J. Harrold, Analyzing Regression Test Selection Techniques, IEEE Transactions on Software Engineering, V.22, no. 8, August 1996, pages 529-551.

[49] P.A Brown and D. Hoffman. The application of module Engineering Conference Proceedings, pages 487-496, September 1989. regression testing at TRIUMF Nuclear Instruments and Methods in Pysics Research, Section A, . A293(1-2):377- 381, August 1990.

[40] S. P. Fiedler, Object-oriented unit testing," Hewlett-Packard Journal, pp. 69 - 74, April 1989.

[41] D. E. Perry and G. E. Kaiser, Adequate testing and object-oriented program-ming," Journal of Object-Oriented Programming, Vol. 2, pp. 13 - 19, January/February 1990.

[42] J. Lee and X. He, A methodology for test selection," Journal of Systems and Software, Vol. 13, pp. 177 - 185, 1990.