Using Metaheuristic Search Technique Optimizing the Test Cases for Regression Testing

Alok Kumar[1], Vaibhav Sharma[2],
[1]*Department of Computer Science & Technology, Department of Computer Science & Engineering*
[12]*Assistant Professor, Central University of Jharkhand, Ranchi, India, SRM University, Delhi-NCR, Sonepat, Haryana, India.*

Abstract- This proposed technique investigates the use of a metaheuristic approach, called genetic algorithm, for this test-suite reduction problem. Unlike other algorithms, our algorithm uses a new criteria, which is a combination of a block based coverage criteria and a test-execution cost criteria, to make decisions about reducing a test suite. Finally, Results show that our algorithm can significantly reduce the size and the cost of the test-suite for regression testing, and the test-execution cost is one of the most important features that must be taken into consideration for test-suite reduction. In Existing techniques, Greedy approach use test-suite coverage criteria, other criteria such as risk or fault-detection effectiveness, or combination of this criterion. The greedy algorithm is sub set selection problem which is NP complete. The problem occurring when use this approach are test suit reduction time is more and Increasing complexity to find the coverage.

Index Terms:-Test-suite reduction; Regression testing; Test Levels; Genetic Algorithms, testing strategies**.**

I. INTRODUCTION

Testing is a crucial part of the software life cycle, and recent trends in software engineering evidence the importance of this activity all along the development process. Testing activities have to start already at the requirements specification stage, with ahead planning of test strategies and procedures, and propagate down, with derivation and refinement of test cases, all along the various development steps since the code-level stage, at which the test cases are eventually executed, and even after deployment, with logging and analysis of operational usage data and customer's reported failures. Testing is a challenging activity that involves several high demanding tasks: at the forefront is the task of deriving an adequate suite of test cases, according to a feasible and cost effective test selection technique. However, test selection is just a starting point, and many other critical tasks face test practitioners with technical and conceptual difficulties (which are certainly under-represented in the literature): the ability to launch the selected tests (in a controlled host environment, or worse in the tight target environment of an embedded system); deciding whether the test outcome is acceptable or not (which is referred to as the test oracle problem); if not, evaluating the impact of the failure and finding its direct cause (the fault), and the indirect one (via Root Cause Analysis); judging whether testing is sufficient and can be stopped, which in turn would require having at hand measures of the effectiveness of the tests: one by one, each of these tasks presents tough challenges to testers, for which their skill and expertise always remains of topmost importance.

II. TERMINOLOGY AND BASIC CONCEPTS

Before deepening into testing techniques, we provide here some introductory notions relative to testing terminology and basic concepts.

Nature Of The Testing

There exist many types of testing and many test strategies, however all of them share a same ultimate purpose: increasing the software engineer confidence in the proper functioning of the software. Towards this general goal, a piece of software can be tested to achieve various more direct objectives, all meant in fact to increase confidence, such as exposing potential design flaws or deviations from user's requirements, measuring the operational reliability, evaluating the performance characteristics, and so on. Generally speaking, test techniques can be divided into two classes:

1. Static analysis techniques

A. Dynamic analysis techniques

Static and dynamic analyses are complementary techniques: the former yield generally valid results, but they may be weak in precision; the latter are efficient and provide more precise results, but only holding for the examined executions.

TYPES OF TESTS

The one term *testing* actually refers to a full range of test techniques, even quite different from one other, and embraces a variety of aims.

Static Techniques

A coarse distinction can be made between dynamic and static techniques, depending on whether the software is executed or not. Static techniques are based solely on the (manual or automated) examination of project documentation, of software models and code, and of other related information about requirements and design. Thus static techniques can be employed all along development, and their earlier usage is of course highly desirable. Considering a generic development

process, they can be applied.

At the requirements stage for checking language syntax, consistency and completeness as well as the adherence to established conventions;
At the design phase for evaluating the implementation of requirements, and detecting inconsistencies (for instance between the inputs and outputs used by high level modules and those adopted by sub- modules).
During the implementation phase for checking that the form adopted for the implemented products (e.g., code and related documentation).

- ***Software inspection*:** the step-by-step analysis of the documents (deliverables) produced, against a compiled checklist of common and historical defects.
- ***Software reviews***: the process by which different aspect of the work product is presented to project personnel (managers, users, customer etc) and other interested stakeholders for comment or approval.
- ***Code reading*:** the desktop analysis of the produced code for discovering typing errors that do not violate style or syntax.
- ***Algorithm analysis and tracing***: is the process in which the complexity of algorithms employed and the worst case, average-case and probabilistic analysis evaluations can be derived.

Dynamic Techniques
Dynamic techniques obtain information of interest about a program by observing some executions. Standard dynamic analyses include testing (on which we focus in the rest of the chapter) and *profiling*. Essentially a program profile records the number of times some entities of interest occur during a set of controlled executions. Profiling tools are increasingly used today to derive measures of coverage, for instance in order to dynamically identify control flow invariants, as well as measures of frequency, called *spectra*, which are diagrams providing the relative execution frequencies of the monitored entities. In particular, *path spectra* refer to the distribution of (loop-free) paths traversed during program profiling. Specific dynamic techniques also include simulation, sizing and timing analysis, and prototyping.

Objective of Testing
Software testing can be applied for different purposes, such as verifying that the functional specifications are implemented correctly, or that the system shows specific non-functional properties.

Acceptance/ qualification testing generally, it is run by or with the end-users to perform those functions and tasks the software was built for.

- **Installation testing:** the system is verified upon installation in the target environment. Installation testing can be viewed as system testing conducted once again according to hardware configuration requirements.
- **Alpha testing:** before releasing the system, it is deployed to some in- house users for exploring the functions and business tasks. Generally there is no test plan to follow, but the individual tester determines what to do.
- **Beta Testing:** the same as alpha testing but the system is deployed to external users. In this case the amount of detail, the data, and approach taken are entirely up to the individual testers. Each tester is responsible for creating their own environment, selecting their data, and determining what functions, features, or tasks to explore. Each tester is also responsible for identifying their own criteria for whether to accept the system in its current state or not.
- **Reliability achievement:** testing can also be used as a means to improve reliability; in such a case, the test cases must be randomly generated according to the operational profile, i.e., they should sample more densely the most frequently used functionalities.
- **Conformance Testing/Functional Testing:** In particular it checks whether the implemented functions are as intended and provide the required services and methods. This test can be implemented and executed against different tests targets, including units, integrated units, and systems.
- **Regression testing:** According to *regression testing* is the "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements". In practice, the objective is to show that a system which previously passed the tests still does.
- **Performance testing**: this is specifically aimed at verifying that the system meets the specified performance requirements, for instance, capacity and response time.
- **Usability testing**: this important testing activity evaluates the ease of using and learning the system and the user documentation, as well as the effectiveness of system functioning in supporting user tasks, and, finally, the ability to recover from user errors.
- **Test-driven development**: test-driven development is not a test technique per se, but promotes the use of test case specifications as a surrogate for requirements document rather than as an independent check that the software has

correctly implemented the requirements.

- Increasing the confidence that the developed product correctly implements the required capabilities;
- Collecting information useful for deciding the release of the product.

Generally system testing includes testing for performance, security, reliability, stress testing and recovery. In particular, test and data collected applying system testing can be used for defining an operational profile necessary to support a statistical analysis of system reliability.

Regression Test

Properly speaking, *regression test* is not a separate level of testing, but may refer to the retesting of a unit, a combination of components or a whole system (see Fig.1.1 below) after modification, in order to ascertain that the change has not introduced new faults.

As software produced today is constantly in evolution, driven by market forces and technology advances, regression testing takes by far the predominant portion of testing effort in industry.

III. TEST CASE REDUCTION TECHNIQUE

A. Test Suite Reduction Problem

Test case reduction technique reduces the effective test cases thereby reducing the test cost to nearly half and hence reduces the overhead during maintenance phase. It focuses on reducing test suites to obtain a subset that yields equivalent coverage with respect to some criteria.

Problem: Find a representative set of test cases from TS_j that satisfies all of the Req_i.

	Req_1	Req_2	...	Req_j	...	Req_k
tc_1	1	1	...		...	0
tc_2	0	0	...		...	1
...	...	...	...		...	...
tc_i	1	1	...		...	1
...	...	...	...		...	...
tc_n	0	1	...		...	0
...	...	...	...		...	...

Table 2.1 Test Requirement coverage information

A. Test Requirements Coverage

The logical form of Requirement coverage information can be derived, as shown in table1. The Req_i in the foregoing statement can represent various test case requirements, such as source statements, blocks, decisions, definition-use associations, or specification items.

B. Existing Test Suite Reduction Techniques

1. Greedy Algorithm

The test case reduction technique basically known as Test Filter, selects test cases based on their statement-coverage (i.e., weight). Note, weight refers to the number of occurrences of a particular test case that covers different statement of the program under test. The technique first calculates weight of all generated test cases. Next it selects test cases of higher weight and marked all of its corresponding requirements as satisfied. Again this process continues until all requirements are satisfied. In case of tie between test cases (i.e., test cases having same weight), random selection strategy is used.

2. Modified Greedy Algorithm

Usually used in test laboratory, the greedy algorithm takes into consideration the change in the coverage when choosing a test case to add to the reduced test-suite. We calculate the marginal coverage of each test case, i.e., the change in the coverage as a consequence of the change in reduced test- suite. We then compare it with the change in cost, and choose the test case that proves to be the best.

step1: Let $T = \emptyset$;

step2: For each $t_i \in TS\text{-}T$, calculate the increase in coverage and cost if it is added to T:

$$\Delta Covrg(t_i) = Covrg(T \cup t_i) - Covrg(T)$$

$$\Delta Cst(t_i) = Cst(T \cup t_i) - Cst(T)$$

step3: Find a test case t_i in TS-T for which $\Delta Covrg(t_i)/\Delta Cst(t_i)$ is minimal. If there are more, then choose the one with the lowest index. Let $T = T \cup t_i$;

step4: If $\Delta Covrg(T) \geq K$, then STOP, otherwise go to Step2.

Here, $Covrg(t_i)$ denotes the coverage information of test case t_i and $Cst(t_i)$ denotes the cost information of test case t_i.

a. Get Split Algorithm

Dynamic Domain Reduction (DDR) DDR is the technique that creates a set of values that executes a specific path. It transforms source code to a Control Flow Graph (CFG). A CFG is a directed graph that represents the control structure of the program. Each node in the graph is a basic block, a junction, or a decision node

b. Coverall Algorithm

Steps:

1) Finding all possible constraints from start to finish nodes. A constraint is a pair of algebraic expressions which dictate conditions of variables between start and finish nodes ($>, <, =, \geq, \leq, \neq$).

2) Identifying the variables with maximum and minimum values in the path, if any. Using Conditions dictated by the constraints, two variables, one with maximum value and the other with minimum value, can be identified. To reduce the test cases, the maximum variable would be set at the highest value within its range, while assigning the minimum variable at the lowest possible value of its range.

3) Finding constant values in the path, if any. When constant values can be found for any variable in the path, the values would then be assigned to the given variables at each node.

4) Using all of the above-mentioned values to create a table to present all possible test cases.

IV. RELATED WORK

A. In Existing techniques, Greedy approach use test-suite coverage criteria, other criteria such as risk or fault-detection effectiveness, or combination of this criteria. The greedy algorithm is sub set selection problem which is NP complete. The following problem occurring when use this approach

a. Test suit reduction time is more.

b. Increasing complexity to find the coverage.

This proposed technique investigates the use of a metaheuristic approach, called genetic algorithm, for this test-suite reduction problem. Unlike other algorithms, our algorithm uses a new criteria, which is a combination of a block based coverage criteria and a test-execution cost criteria, to make decisions about reducing a test suite. Finally, Results show that our algorithm can significantly reduce the size and the cost of the test-suite for regression testing, and the test-execution cost is one of the most important features that must be taken into consideration for test-suite reduction.

V. SYSTEM DESIGN

A. TSR Using Greedy Algorithm

The working procedure of this approach is as follows:

Step 1: Calculates a Weighted Set (WS) of test cases. The weighted set is a function from test cases to their weights. The weight of a test case is the number of its occurrences in the set of test suites.

Step 2: Select the first test case (tch) from the WS that has the highest weight. In case of a tie between test cases, use a random selection.

Step 3: Move tch to the Representative Set (RS), and mark all test suites from Set of Test Suites (STS), which contain tch in their domain. If all test suites of STS are marked then exit, otherwise go back to step1. Consider the following function Value takes three integers inputs X, Y, Z, and returns an integer V.

```
Value Function: Int value (x, y, z) Int
x,y,z;
        {

        Int v; V=0;
        If (x<y)

        { Z=15;
        If (x<z) V=x+20;
        else V=x;
        }

        else
        { Z=40;
        V=x+y+z;

        }
```

Consider the values for variables X, Y, Z respectively as follows,

X [ ] = {11, 2, 15};

Y [ ] = {15, 20, 9};

Z [ ] = {6, 10, 17};

The test cases are developed using black box and white box techniques for validation purposes. All possible test cases came from number of values on the each variable 3*3*3=27.Saving (%) =100- ((100*Reduced test_cases)/all possible test_cases).

B. TSR Using Genetic Algorithm (1)Genetic algorithm:

A GA is a programming technique that mimics the process of natural genetic selection according to Darwin Theory of Biological Evolution as a problem solving strategy. Genetic algorithms represent a class of adaptive search techniques, based on biological evolution, which are used to approximate solutions.

GA are optimization algorithms based on natural genetics and selection mechanisms. To apply genetic algorithms to a particular problem, it has to be decomposed into atomic units that correspond to genes. Then individuals can be built with correspondence to a finite string of genes, and a set of individuals is called a population. A criterion needs to be

defined: a fitness function F which, for every individual among a population, gives F(x), the value which is the quality of the individual regarding the problem we want to solve.
Once the problem is defined in terms of genes, and fitness function is available, a genetic algorithm is computed following the process described.

Genetic Algorithm:
1. Initialization of Population.
2. Calculation of Fitness value among individual.
3. Reproduction.
4. Crossover.
5. Mutation.
6. Several Halting Criteria i.e. a given value is reached.

Table 4.2 Action of Genetic Algorithm

As an adaptive search technique, genetic algorithms have been used to find solutions to many NP-complete problems and have been applied in many areas and can find a better solution. Genetic algorithm uses three operators: reproduction, crossover and mutation.

(2) Steps in genetic algorithm
A simple genetic algorithm is as follows:

(1) **Initialization of population:** A population is initialized randomly. Each of these strings represents one feasible solution to the search problem.

(2) **Fitness evaluation:** The fitness of each candidate is evaluated through some appropriate measure. After the fitness of the entire population has been determined, it must be determined whether or not the Termination criterion has been satisfied. If the criterion is not satisfied, then we continue with the three genetic operations of reproduction, crossover, and mutation.

(3) **Selection:** In this operation chromosomes are selected from the population to be parents to crossover and produce offspring. According to theory of survival of fittest, the best ones should survive and create new offspring.

(4) **Crossover:** Crossover operator is applied to the mating pool with a hope that it would create a better solution. The aim of this operator is to search the parameter space. In addition, search is to be made in a way that the information stored in the present solution is maximally preserved because these parent solutions are instances of good solutions selected during reproduction.

(5) **Mutation:** Mutation is simply an insurance policy against the irreversible loss of genetic material. It introduces new genetic structures in the population by randomly modifying some of its building blocks. It helps the search algorithm to escape from local minima's traps since the modification is not related t o any previous structure of the population. The mutation is also used to maintain diversity in population.

Tournament Selection: The tournament selection provides a selective pressure by holding a tournament competition among individuals. The best individual (the winner) from this group is selected as parent. This process is repeated until the mating pool for generating new offspring is filled.

Simulated Binary Crossover: It creates children solutions in proportion to the difference in parent solutions. The two properties which give SBX its search power are, The extend of children solution is in proportion to the parent solutions Near parent solutions are more likely to be chosen as children solutions than solutions distant from parents.

Polynomial Mutation: Newly generated offspring undergo polynomial mutation operation. Like in the SBX operator, the probability distribution can also be a polynomial function, instead of a normal distribution.

A. Genetic algorithm for test-suite reduction
Gene modeling for test-suite reduction: - For our problem of test-suite reduction, gene of an individual is modeled as a '0'-'1'. Although in its primitive form each test-subset cannot satisfy the test coverage bound, it is believed that after evolution of numbers of generations it can evolve to be a feasible solution to this problem. So, each test- subset is naturally to be considered as a potential solution or an ancestor of the solution to our reduction problem. That is to say if there are k test cases in the original test-suite and the program has n statements, the size of population is n, and the length of gene code is

k. So the gene code of an individual can be represented as follows:

$G_j = [g_{j1}, g_{j2}, .., g_{jn}]$, g_{ji} (0, 1), $1 \leq j \leq k$ and $1 \leq i \leq n$, here if $g_{ji} = 1$, it means test case t_i has tested statement s_j, and if $g_{ji} = 0$ means case t_i has not tested s_j according to the original test histories shown in Table1. So, the gene of each individual denotes a subset of test cases that have tested s_j and this subset is named as T_j. Our genetic algorithm operates on the genes with selection, crossover, and mutation, to find an individual that satisfies the coverage bound with minimal cost. Another aspect of the genetic algorithm which has to be decided for the particular problem is a fitness function.

Fitness function: The fitness value for an individual is the combination value of its associated coverage and its cost. The fitness function for individual t_i can be computed as follows:

$$F(t_i) = \frac{(g_j * w_j)}{\text{----------}}$$

$C(t_i)$

$C(t_i)$ is the cost of t_i when used to test the program. Consider the function $F(t_i)$, the bigger the function $F(t_i)$ value is, the more possible the case can be selected, because the bigger value of F denotes that the case has tested more statements with less cost.

Selection: Selection operator is used to choose chromosomes from a population for mating. This mechanism defines how these chromosomes will be selected, and how many off springs each will create. Selection has to be balanced: too strong selection means that best chromosomes will take over the population reducing its diversity needed for exploration; too weak selection will result in a slow evolution. Classic selection methods are Roulette-Wheel, Rank based, Tournament, Uniform, and Elitism. In this work tournament selection is used.

In tournament selection, a "tournament" is run among a few individuals chosen at random from the population and the winner (individual with the best fitness) is selected for crossover. Selection pressure can be adjusted by changing the tournament size. If the tournament size is larger, weak individuals have a smaller chance to be selected..

Crossover: Let m be the size of individuals in a population, and let's select an integer i at random between 1 and m-i, then from two individuals ind1 and ind2, we can create two new individuals ind3 and ind4; one made of the i first genes of ind1 and the m-i last genes of ind2, and the other made of the i first genes of ind2 and m-i last genes of ind1.

Mutation: Based on the gene model, the mutation operator consists in replacing syntactic node by another licit node. The rate of the mutation is defined as 1/L; n which L is the number of bits in the gene. Based on the gene model, the mutation operator consists in changing a '1' in to '0' and vice versa. The mutation operator chooses a gene at random in an individual.

Termination: The generational process is repeated until a termination condition has been met. There is no guarantee that the genetic algorithm will converge upon a single solution. Some common terminating conditions are:

- A solution is found that satisfies minimum criteria
- Fixed number of generations reached
- Allocated budget (computation time/money) reached
- Highest ranking solution fitness has reached a plateau such that successive iterations no longer produce better results.

(3) TESTING

Control Structure testing

In this testing, all the logical statements are in the implementation of both greedy and genetic algorithm have tested by using block box testing with help of WINRUNNER tool. Finally tool ensured that following properties are satisfied from source code.

1. All independent paths are exercised at least once.
2. All the logical statements are exercised for both true and false paths.
3. All the loops are executed at their boundaries and within operational bounds.
4. All the internal data structure is exercised to ensure validity.

Basic path testing

A testing mechanism proposed by McCabe. Aim is to drive a logical complexity measure of a procedural design.

Boundary value Analysis

Generally, the large no of errors tend to occur at boundaries of the input domain. BVA leads to selection of the test cases that exercise boundary values. BVA complements equivalence portioning, rather than select any element in an equivalent class, select those at the edge of the class. Finally this analysis technique analyzed the genetic algorithm because there we need some boundary value for optimization process.

VIII. CONCLUSION AND FUTURE ENHANCEMENT

This work has presented a mathematical model of our test reduction problem and transformed it into a linear integer-programming problem. The results of studies are encouraging. They show the potential for substantial reduction of test-suite size and cost, and genetic algorithm is more effective than greedy approaches. The initial studies also showed that the promotion of effectiveness in test-cost reduction could be achieved by taking the cost criteria into consideration. We conclude that, the cost reduction is an important characteristic needed to be taken into consideration in test-suite reduction.

A. FUTURE ENHANCEMENT

With help of parallel test case execution procedure to improve the cost of testing and reduce the complexity to find the coverage of code and also future work investigate test suite reduction that attempts to use addition coverage information of test cases to selectively keep some additional test cases in the reduced suites that are redundant with respect to the test criteria used for suite minimization, with the goal of improving the fault detection effectiveness redundant of the reduced suite and modifying an existing heuristics for test suite minimization.

IX. REFERENCES

[1]. Anoj Kumar, 2Shailesh Tiwari, 3 K. K. Mishra and 4A.K. Misra, Generation of Efficient Test Data using Path Selection Strategy with Elitist GA in Regression Testing,IEEE 2007,PP 43 -51

[2]. Agastya Nanda_, Senthil Mani†, Saurabh Sinha†, Ma ry Jean Harrold‡, and Alessandro Orso‡, Regression Testing in the Presence of Non-code Changes,IEEE 2011,PP 211-218.

[3]. Bing JIANG, Yongmin MU, Research of Optimization Algorithm for Path-Based Regression Testing Suit,IEEE 2011,PP 122-128.

[4]. Dennis Jeffrey and Neelam Gupta, Improving Fault Detection Capability By Selectively Retaining Test Cases during Test Suite Reduction, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 2, FEB- 2007,PP108-127.

[5]. Engin Uzuncaova, Sarfraz Khurshid, and Don Batory, Incremental Test Generation for Software Product Lines, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 3, MAY/JUNE 2010 PP 309-321.

[6]. Kaner.C, J. Falk, and H.Q. Nguyen H.Q. Testing Computer Software,2nd Edition, John Wiley & Sons, April, 1999.

[7]. Gregory M. Kapfhammer, Empirically Evaluating Regression Testing Techniques: Challenges, Solutions, and a Potential Way Forward, IEEE 2011,PP 78-84.

[8]. Hyunsook Do, Ladan Tahvildari, The Effects of Time Constraints on Test Case Prioritization, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 5, SEPTEMBER/OCTOBER 2010PP 593-614

[9]. Irman Hermadi, Chris Lokan, Genetic Algorithm Based Path Testing: Challenges and Key Parameters, 2010 Second WRI World Congress on Software Engineering PP 341- 356.

[10]. James H. Andrews, Genetic Algorithms for Randomized Unit Testing, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 37, NO. 1, JANUARY/FEBRUARY 2011,PP 80-102.

[11]. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Preliminary Guidelines for Empirical Research in Software Engineering, IEEE 2005,PP 18-24.

[12]. Mary Jean Harrold, ,Empirical Studies of a Prediction Model for Regression Test Selection, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 27, NO. 3, MARCH 2001,PP 248-260

[13]. Mark Harman, Kiran Lakhotia, Phil McMinn, A Multi–Objective Approach To Search–Based Test Data Generation, IEEE 2008,PP 98-105.

[14]. Lyu M.R, eds., Handbook of Software Reliability Engineering, McGraw-Hill, 1996.

[15]. Vaibhav Sharma, Amit Kumar, Priority Based Congestion Control Technique for Heterogeneous Applications in WSN, IJRIT 2014.