American Journal of Information Science and Computer Engineering
Vol. 1, No. 1, 2015, pp. 10-20
http://www.aiscience.org/journal/ajisce

Optimization the Test Suite of Regression Testing Using Metaherustic Searching Technique

Aakanksha Pandey[1, *], **Jayant Shekher**[2]

[1]Computer Science & Engineering, SRM University, NCR Campus, Modinagar, Ghaziabad, India
[2]Computer Science & Engineering, Subharti University, Meerut, India

Abstract

This proposed technique investigates is for the reduction of the test suit with the use of metaheuristic approach this technique is known as genetic algorithm. The result is showing like with the help of regression testing we can reduce the size n cost of the test suit significantly the very important features of the test suit that we need to take in consideration is "test suit reduction". Here we have uses the algorithm that is the combination of the test-execution cost criteria and block based coverage criteria, these new criteria with that we can make the prominent decision for reducing the test suit. Here for the test-suit coverage criteria other criteria such as risk or fault-detection effectiveness, or combination of this criterion we have used the approach is greedy algorithm that is the sub set selection problem which is NP complete.

Keywords

Metaheuristic Approach, Genetic Algorithm, Regression Testing, Test Suite

Received: April 16, 2015 / Accepted: May 3, 2015 / Published online: June 3, 2015

1. Introduction

The software testing basically depends on three factors test case generation, test execution, and test evaluation. As the prevasiness of software has increased over decades, testing has become a business – critical part of the software lifecycle. Testing is the very important and unavoidable part of the any software life cycle, testing cannot guarantee the absence of defects. The problem at hand is to select a subset of test cases for all possible test cases with a high chance of detecting defects. Standard design for the test cases allows measurements of testing performed. The main aim of the testing is to detect the failure of the software that detect may be discovered or corrected. Although testing can precisely determine the correctness of software under the assumption of some specific hypotheses.

A very basic and fundamental problem with testing is like it might be possible that testing is feasible with all the combinations of input and preconditions.

In order to "retest all" is the very expensive and time taking task so here we are using regression test selection is perform to optimize the cost . if we are talking about the classification of RTS that we can divide this into three categories Coverage techniques, Minimization techniques and Safe techniques.

One more technique we are using here for the optimization of the test suits i.e. Genetic algorithm. This technique we can apply on all kind of problems even we can use this for the NP hard also. Here we are using the metaheuristic approach to reduce the test suit in optimal (minimum) time.'Meta' means *abstract* and a 'heuristic' is a *search*, Metaheuristic have played a very important role to optimize the test suits , this is strategy to guide all the search process in order to provide sub optional solution in a perfect reasonable time . Generally the metaheuristic approach is approximate and non-

* Corresponding author
Email address: er.aakanksha24@gmail.com (A. Pandey), Jayant_shekhar@hotmail.com (J. Shekher)

deterministic.

Related work: Before going through any testing technique or discussion we will provide some introduction to relative testing terminology and some basic concept.

1.1. Category of the Testing

There are various kinds of testing that we are using to test different software's to perform tasks.

It's totally impossible to check all the parts while testing, testing does not means that system is totally free of faults.

Mainly the software testing can be divided into major parts

i Static testing technique

ii Dynamic testing technique

Here we are elaborating the main category of the testing techniques and embraces a variety of aims.

1.2. A General Definition

Testing can refer to many different activities used to check a piece of software. As said, we focus primarily on "dynamic" software testing presupposing code execution, for which we re-propose the following general definition introduced in Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior.

2. 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.

2.1. 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.

Static testing solely depends based solely on the (manual or automated) examination of project documentation, of software models and code.

Traditional static techniques include:

- Software inspection: this process includes the step-by-step analysis of the deliverables produced.
- Software reviews: this process is all about different aspect of the work product is presented to project personnel.
- Code reading: here mainly we deal with the desktop analysis of the produced code for discovering typing errors that do not violate style or syntax.
- Algorithm analysis and tracing: here average-case and probabilistic analysis evaluations can be derived and mainly this is the process in which the complexity of algorithms employed and the worst case.

2.2. 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.

Basically the question arises like why we are using testing?

Actually the testing we are using to know the correctness of the working software, verifying the system verifying that the functional specifications are implemented correctly In the field of software testing there is 'N' numbers of testing are available for the different customers specifications and their requirement, Very frequently some of the testing that we are usin.

2.3. 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 nonfunctional properties such as performance, reliability, usability. A (certainly non complete) list of relevant testing objectives includes:

- Acceptance/qualification testing: the final test action prior to deploying a software product. Its main goal is to verify that the software respects the customer's requirement. 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. Installation procedures may also be verified.

- 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: the test cases are aimed at validating that the observed behavior conforms to the specifications. 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.

3. Test Levels

During the development lifecycle of a software product, testing is performed at different levels and can involve the whole system or parts of it. Depending on the process model adopted, then, software testing activities can be articulated in different phases, each one addressing specific needs relative to different portions of a system. Whichever the process adopted, we can at least distinguish in principle between unit, integration and system test.

3.1. Unit Test

A unit is the smallest testable piece of software, which may consist of hundreds or even just a few lines of source code, and generally represents the result of the work of one programmer. The unit test’s purpose is to ensure that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure.

3.2. Integration Test

Generally speaking, integration is the process by which software pieces or components are aggregated to create a larger component. Integration testing is specifically aimed at exposing the problems that can arise at this stage. Even though the single units are individually acceptable when tested in isolation, in fact, they could still result in incorrect or inconsistent behavior when combined in order to build complex systems.

3.3. System Test

System test involves the whole system embedded in its actual hardware environment and is mainly aimed at verifying that the system behaves according to the user requirements. In particular it attempts to reveal bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects.

3.4. 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 Figure- 1 below) after modification, in order to ascertain that the change has not introduced new faults.

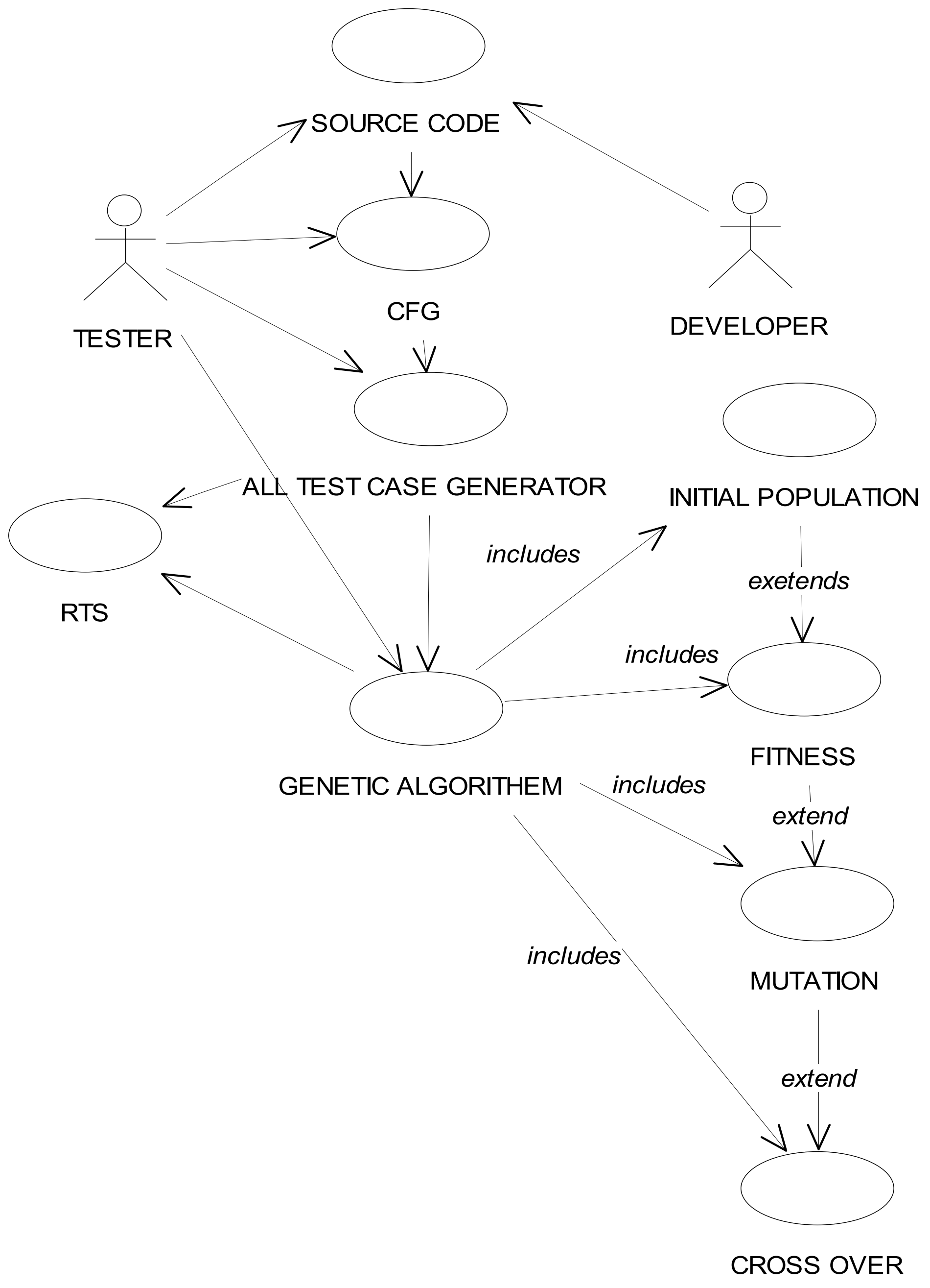


Figure 1. Genetic process use case diagram.

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.

4. Regression Testing Techniques

4.1. Definition

Regression testing is defined as “the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested code”. Let P be a program, let P′ be a modified version of P, and let T be a test suite for P. Regression testing consists of reusing T on P′, and determining where the new test cases are needed to effectively test code or functionality added to or changed in producing P′. There are various regression testing techniques (1) Retest all; (2) Regression Test Selection; (3) Test Case Prioritization; (4) Hybrid Approach. Figure1.1 shows various regression testing techniques.

4.2. Techniques

Retest All

Retest all method is one of the conventional methods for regression testing in which all the tests in the existing test suite are returned. So the retest all technique is very expensive as compared to techniques which will be discussed further as regression test suites are costly to execute in full as it require more time and budget.

5. Test Case Reduction Technique

5.1. 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.

The Optimal Test suite reduction problem may be stated as follows: Given: Test suite TS_j, a set of test requirements Req_1, Req_2, Req_n that must be satisfied to provide the desired test coverage of the program, and subsets of TS_1, TS_2, TS_3, TS_n, one associated with each of the Req_i's such that any one of the test cases tc_i belonging to TS_j can be used to test Req_i.

Table 1. Test Requirement coverage information.

	Req1	**Req2**	**…**	**Reqj**	**…**	**Reqk**
tc1	1	1	…		…	0
tc2	0	0	…		…	1
…	…	…	…		…	…
tci	1	1	…		…	1
…	…	…	…		…	…
tcn	0	1	…		…	0
…	…	…	…		…	…

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

5.2. 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.

In table 1, Req_1, Req_2, Req_j Req_k are the requirements of the program, and tc_1, tc_2,..,tc_i,...,and tc_n is the test cases that have been executed. The information in the table is all the digits of '0' or '1' which denote the requirements-coverage information; here '1' in the row i and the column j means tc_i tested requirement Req_j , and '0' means tc_i did not tested requirement Req_j. The number of '1' in rows i means how many requirements covered by tc_i; and the number of '1' in column j means how many times the requirement Req_j executed in a test.

5.3. Existing Test Suite Reduction Techniques

5.3.1. 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.

5.3.2. 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 $t_i \in TS-T$, calculate the increase in coverage and cost if it is added to T:

$$\Delta Cov(ti) = Cov(T \cup ti) - Cov(T)$$

$$\Delta Cost\,(ti) = Cost\,(T \cup ti) - Cost\,(T)$$

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

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

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

5.3.3. 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.DDR uses the Get Split algorithm to find a split point to divide the domain. The Get Split algorithm is as follows:

Algorithm

Getsplit(LeftDom, RightDom, SrchIndx)

```
Precondition
LeftDom and RightDom are initialized appropriately and SrchIndx is one more than the last time
Getsplit was called with these domains for this expression.
Postcondition
Splitvalue ≥ (LeftDom.Bot AND RightDom.Bot) and
Splitvalue ≤(LeftDom.Top AND RightDom.Top)
Input
LeftDom: Left expr's domain with Bot and Top values
RightDom: right expr's domain with Bot and Top values
Output
Split–a value the divides a domain of values into two sub domains.
BEGIN
-- Compute the current search point
-- srchPt = (1/2, 1/4, 3/4, 1/8, 3/8, … )
Choose exp such that 2exp ≤ SrchIndx ≤ 2exp +1
SrchPt = (2exp - (2 - (2exp -1) -1)) /2exp -- Try to equally split the left and right expression's domains.
IF (LeftDom.Bot≥ RightDom.Bot AND LeftDom.Top ≤ RightDom.Top)
split=(LeftDom.Top -LeftDom.Bot)*srchPt + LeftDom.Bot
ELSE IF (LeftDom.Bot≤ RightDom.Bot AND LeftDom.Top ≥ RightDom.Top)
split=(RightDom.Top -RightDom.Bot)*srchPt + RightDom.Bot
ELSE IF (LeftDom.Bot≥ RightDom.Bot AND LeftDom.Top ≥ RightDom.Top)
split=(RightDom.Top - LeftDom.Bot)*srchPt + LeftDom.Bot
ELSE -- LeftDom.Bot≤ RightDom.Bot AND LeftDom.Top ≤ RightDom.Top
split=(LeftDom.Top - RightDom.Bot)*srchPt + RightDom.Bot
END IF
RETURN split
END GetSplit
```

In the dynamic domain reduction procedure, loops are handled dynamically instead of finding all possible paths. The procedure exits the loop and continues traversing the path on the node after the loop. This eliminates the need for loop unrolling, which allows more realistic programs to be handled.

5.3.4. 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 (>, <, =, ≥, ≤, ≠).

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.

5.3.5. 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 (tc_h) from the WS that has the highest weight. In case of a tie between test cases, use a random selection.

Step 3: Move tc_h to the Representative Set (RS), and mark all test suites from Set of Test Suites (STS), which contain tc_h 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 A, B, C, and returns an integer V.

Value Function:

```
Int value (a, b, c)
Int a,b,c;
{
Int v;
V=0
If (a<b)
{
C=15;
If (a<c)
V=a+20;
Else
V=a;
}
Else
{
C=40;
V=a+b+c;
}
```

Consider the values for variables A, B, C respectively as follows,

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

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

C [ ] = {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.

6. Use Case Diagram for Genetic Process

6.1. Model of the Test Reduction Problem

Given a test suite TS = t_l, t_2, t_n consisting of the test case and the statements of a tested program S = s_l, s_2, s_k, we have a positive cost, c_j assigned to each test case measuring the amount of resources its execution needs. A positive weight, w_i is assigned to each statement, which represents the relative importance of bi with respect to the correct behavior of program or to the regression testing. For example, we can assign bigger weight to the modified statements or modification affected statements of the new version program.

Let T be an arbitrary set of the test cases, $T \subset TS$. The cost of this test set is defined as the sum of the costs of the test cases that belong to T:

$$C(T) = \sum_{t \in T} C(t)$$

Let Cov (T) denote the coverage of the test set T,

$$Cov(T) = \sum_{t \in T} w_t * Cov(t)$$

The test-suite reduction problem can be defined according to our purposes and bounds:

Minimal cost problem:

Minimal cost problem. Given a lower bound (K) for the coverage, select the set of test cases that satisfies this bound with minimal cost.

min C (T)

$$\text{subject to } Cov(T) \geq K \qquad (1)$$

$T \subset TS$

Here the lower bound (K) is the coverage of the original test-suite. In fact, the coverage of the reduced test suite is impossible to be larger than K.

In our test problem, Cov (T) measures how many statements are tested by T. Furthermore, we can calculate the coverage of the test suite T as follows:

$$Cov(T) = \sum_{t \in T} w_t * Cov(t) = \sum_{s_i \in S} w_t * stCov(s_i, T) \qquad (2)$$

Here stCov (s_i,T) measures whether the test-suite T exercised statement s_i, and if a single test case t_j . T tested s_i,s_tCov$(s_i,T)=1$, otherwise stCov$(s_i,T)=0$.

The minimal cost problem in our test selection is equivalent to the Set Covering Problem, which is known to be NP-hard. We transform the minimal cost problem to a linear integer-programming problem.

Let a_i be the characteristic vector which contains the column information in Table 1 according to the test statement for i=1.k and A be the 0-1 matrix of size k × n made up of row vectors a_i. Thus $(A)_{ij}=1$ if and only if t_j tests s_i. Let x be the characteristic vector of test set $T \subset TS$, c be the cost vector and w be the vector containing the weights of statements. Then

$$C(T)=cx \text{ and } Cov(T) = \sum^{k}_{i=1} w_i * f_i(a_i x) \text{ for } i = 1, 2, k.$$

Using these notations the minimal cost problem (1) can be written as:

min cx

$$\text{subject to } \sum ki=1 \text{ wi} * \text{fi(aix)} \geq K \qquad (3)$$

$x \in \{0, 1\}^n$

Let us define a new variable vector z = (z_1, z_2, z_k) in the following manner:

Zj= 1 if $a_i x \geq 1$ I i=1. k.

In other words, z_j = 1 if statement s_j has been tested by test set represented by x. Using this vector problem, from(3) can be transformed into as follows:

min cx

subject to $Ax \geq z$

$z \geq K$

Based on the mathematical model, we present a genetic algorithm for test-suite reduction.

6.2. TSR Using Genetic Algorithm

(1)Genetic algorithm:

A genetic algorithm is a programming technique that mimics the process of natural genetic selection according to Darwinian 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.

Genetic algorithms 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.

Table 2. Action of Genetic Algorithm.

Genetic Loop:
Choose an initial population
Calculate the fitness value for each individual.
Reproduction.
Crossover.
Mutation on one or several individual.
Several stopping criteria: X no. of generations, a given value is reached.

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 are 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.

7. Conclusion

This work has presented a mathematical model of our test reduction problem and transformed it into a linear integer-programming problem. By modifying the function Cov(),which is used to calculate the coverage of test suites, the presented reduction algorithm, can be conveniently modified to account for different coverage criteria like block and decision when reducing test suites.

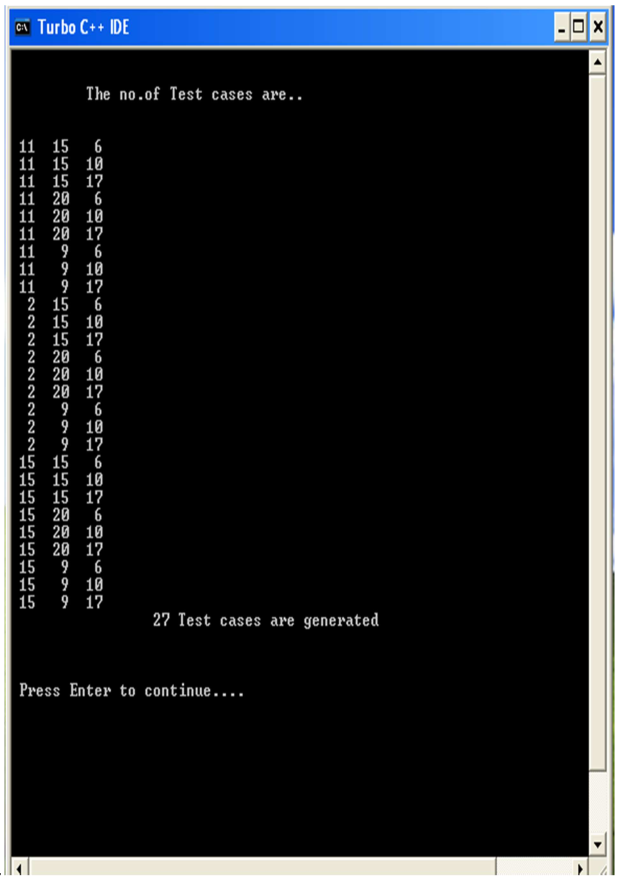
The results of studies are encouraging. They show the potential for substantial reduction of test-suite size and cost, and genetic algorithm is more effective than greedy approaches. The initial studies also showed that the promotion of effectiveness in test-cost reduction could be achieved by taking 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.

8. Future Enhancement

Experiments have to be done to further investigate the fault detection capabilities of a statement/block-based test if it is an adequate test suite for the software. These studies will help evaluate our algorithms and help provide guidelines for test-suite reduction in practice and evaluate parallel algorithm for the test case execution. 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.

8.1. Result Analysis of Greedy Approach

(1) The screen shot shows the number of test cases for Value function to be generated



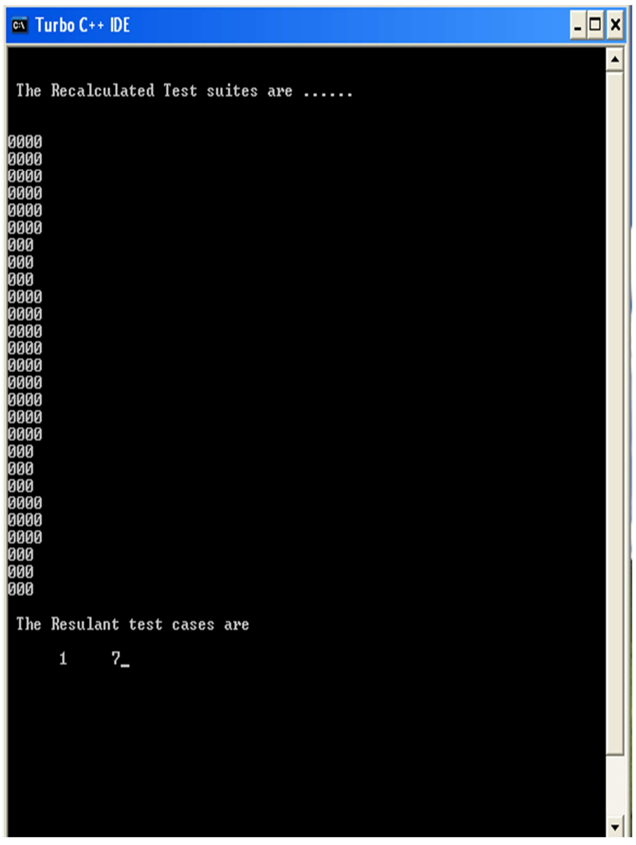
```
Turbo C++ IDE

        The no.of Test cases are..

11  15   6
11  15  10
11  15  17
11  20   6
11  20  10
11  20  17
11   9   6
11   9  10
11   9  17
 2  15   6
 2  15  10
 2  15  17
 2  20   6
 2  20  10
 2  20  17
 2   9   6
 2   9  10
 2   9  17
15  15   6
15  15  10
15  15  17
15  20   6
15  20  10
15  20  17
15   9   6
15   9  10
15   9  17
                  27 Test cases are generated


Press Enter to continue....
```

Figure 2. Test Case Generation (5)The screen shot shows the final iteration of recalculation of all the weight and the resultant test cases.



```
Turbo C++ IDE

 The Recalculated Test suites are ......

0000
0000
0000
0000
0000
0000
000
000
000
0000
0000
0000
0000
0000
0000
0000
0000
0000
000
000
000
0000
0000
0000
000
000
000

 The Resulant test cases are

      1     7_
```

Figure 3. Final iteration of recalculation of weights and Resultant test cases.

8.2. Result Analysis of Genetic Approach

For the Value function shown in by taking the same A, B, C values 27 test cases are generated.

The Req_i in the foregoing statement can represent various test case requirements, such as source statements, decisions, definition-use associations, or specification items

In the value functions identified as 7 requirements are shown in Table 3.

Table 3 shows each statement as separate testing requirement, and its associated.

Table 3. The Coverage Requirements for Value function.

Statements	Reqi	tck in Associated Set
if(A < B)	Req1	tc1 – tc27
C = 16	Req2	tc1 – tc6, tc10 – tc18, tc22 – tc24
if(A < C)	Req3	tc1 – tc6, tc10 – tc18, tc22 – tc24
V = A + 30;	Req4	tc1 – tc6, tc10 – tc18, tc22 – tc24
V = A;	Req5	
C = 30;	Req6	tc7 – tc9, tc19 – tc21, tc25 – tc27
V = C+B+A;	Req7	tc7 – tc9, tc19 – tc21, tc25 – tc27

Test cases. There are total 7 statements, so we have total 7 requirements. Then we determine which test case(s) is/are useful in validating these requirement(s). Table 4 shows the coverage information i.e., mapping of test cases to the statements.

Table 4. Coverage Information.

tci	s1 s2 s3 s4 s5 s6 s7	tci	s1 s2 s3 s4 s5 s6 s7
tc1	11 1 1 0 0 0	tc15	1 1 1 1 0 0 0
tc2	11 1 1 0 0 0	tc16	1 1 1 1 0 0 0
tc3	11 1 1 0 0 0	tc17	1 1 1 1 0 0 0
tc4	11 1 1 0 0 0	tc18	1 1 1 1 0 0 0
tc5	11 1 1 0 0 0	tc19	10 0 0 0 1 1
tc6	11 1 1 0 0 0	tc20	10 0 0 0 1 1
tc7	10 0 0 0 1 1	tc21	10 0 0 0 1 1
tc8	10 0 0 0 1 1	tc22	1 1 1 1 0 0 0
tc9	10 0 0 0 1 1	tc23	1 1 1 1 0 0 0
tc10	11 1 1 0 0 0	tc24	1 1 1 1 0 0 0
tc11	11 1 1 0 0 0	tc25	10 0 0 0 1 1
tc12	11 1 1 0 0 0	tc26	10 0 0 0 1 1
tc13	11 1 1 0 0 0	tc27	1 0 0 0 0 1 1
tc14	11 1 1 0 0 0		

Using Table 4, we can see whether a certain statement has been tested, how many statements have been covered in one test. We can also calculate the test coverage according to certain criteria and evaluate each test-case's contribution by calculating the number of '1' in the row that is associated with each test case.

The fitness function for individual t_i can be computed as follows:

$\sum (g_j * w_j)$

$F(t_i) = C (t_i)$

$C (t_i)$ is the cost of t_i when used to test the program, w_j is the weight of the requirement and g_j is the coverage information.

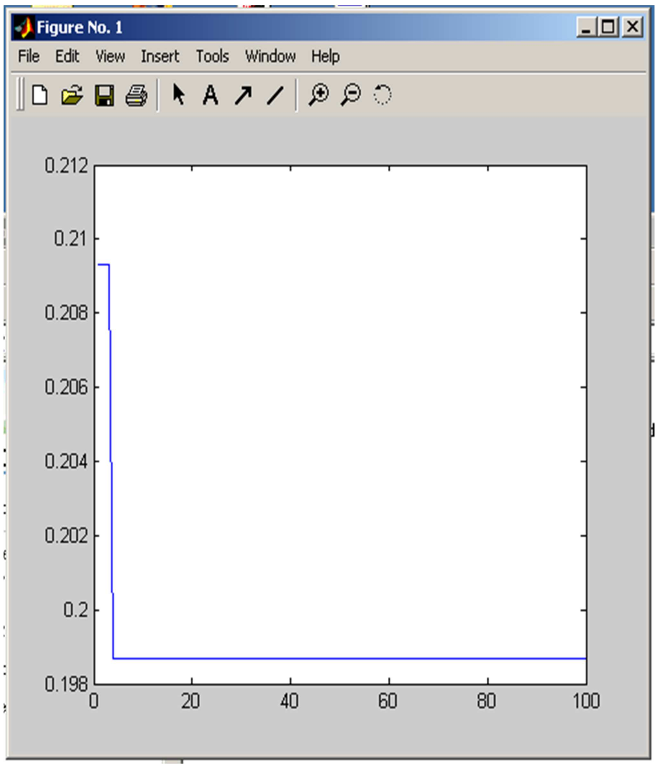


Figure 4. Convergence graph for Test Suite Reduction problem.

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†, Mary 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] Gregory M. Kapfhammer, Empirically Evaluating Regression Testing Techniques: Challenges, Solutions, and a PotentialWay Forward,IEEE 2011,PP 78-84.

[7] 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.

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

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

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

[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] Nigel Tracey John Clark Keith Mander, Automated Program Flaw Finding using Simulated Annealing,IEEE 2007,PP 201-208.

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

[16] Pavan Kumar Chittimalli and Mary Jean Harrold, Senior Member, Recomputing Coverage Information to Assist Regression Testing, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 35, NO. 4, JULY/AUGUST 2009, PP 452-472.

[17] Phil McMinn, Search-based Software Test Data Generation:A Survey,WHITE PAPER.

[18] Preeyavis Pringsulaka and Jirapun Daengdej, Coverall Algorithm for Test Case Reduction,IEEE 2005 ,PP 234-239.

[19] P fleeger.S.L, Software Engineering Theory and Practice, Prentice Hall, 2001.

[20] Stefan Wappler, Ina Schieferdecker, Improving Evolutionary Class Testing in the Presence of Non-Public Methods,IEEE 2004,PP 308-312.

[21] Simon Poulding and John A. Clark Efficient Software Verification: Statistical Testing Using Automated Search, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 6, NOVEMBER/DECEMBER 2010, PP 763-787.

[22] Shaukat Ali, C. Briand, Hadi Hemmati, A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, NO. 6, NOVEMBER/DECEMBER 2010.

[23] Ummu Salima.T.M.S,Ms. A.Askarunisha, Dr. N.Ramaraj, Enhancing The Efficiency Of Regression TestingThrough Intelligent Agents, International Conference on Computational Intelligence and Multimedia Applications 2007,PP 230-238.