Computing and Informatics, Vol. 29, 2010, 133–164

A HYBRID TEST OPTIMIZATION FRAMEWORK – COUPLING GENETIC ALGORITHM WITH LOCAL SEARCH TECHNIQUE

Dharmalingam Jeya Mala, Elizabeth Ruby, Vasudev Mohan

Thiagarajar College of Engineering
Madurai-15, Tamil Nadu, India
e-mail: djmcse@tce.edu, rubytce@gmail.com, vmohan@tce.edu

Manuscript received 23 June 2008; revised 25 March 2009
Communicated by Juraj Štefanovič

Abstract. Quality of test cases is determined by their ability to uncover as many errors as possible in the software code. In our approach, we applied Hybrid Genetic Algorithm (HGA) for improving the quality of test cases. This improvement can be achieved by analyzing both mutation score and path coverage of each test case. Our approach selects effective test cases that have higher mutation score and path coverage from a near infinite number of test cases. Hence, the final test set size is reduced which in turn reduces the total time needed in testing activity. In our proposed framework, we included two improvement heuristics, namely RemoveTop and LocalBest, to achieve near global optimal solution. Finally, we compared the efficiency of the test cases generated by our approach against the existing test case optimization approaches such as Simple Genetic Algorithm (SGA) and Bacteriologic Algorithm (BA) and concluded that our approach generates better quality test cases.

Keywords: Software under test (SUT), software test optimization, genetic algorithm (GA), hybrid genetic algorithm (HGA), bacteriologic algorithm (BA), mutation score, path coverage

Mathematics Subject Classification 2000: 68N30

1 INTRODUCTION

Software testing is one of the time and resource consuming activities in software development life cycle. Usually in testing the tester is interested in finding the quality test cases that test a piece of code. Here the quality of a test case is determined by the ability of a test case in revealing as many errors as possible in the Software Under Test (SUT). In reality, it is not easy to find the quality test cases among the infinite number of test cases generated either manually or automatically. Many studies have been conducted to improve the quality of test cases thus leadng to reduction in the size of the test set. The existing approaches have employed statistical methods, probability distribution based methods, evolutionary methods and so on [26]. In our proposed approach we applied Evolutionary Algorithms (EA) based framework to improve quality of test cases through test case optimization.

The automation of test data generation is still a research area since the automated testing tools simply generate test data with only less consideration on amount of time spent for test data generation and selection of test cases based on the test adequacy criterion [23, 24].

Evolutionary Algorithms (EA) play a vital role in most of the optimization problems [2]. Among them, Simple Genetic Algorithm (SGA) is the easiest and most flexible one. It can be applied to multi-objective optimization problems [2]. The enhanced algorithm called Bacteriologic Algorithm (BA) includes memorization without cross over operation and provides better solution than GA [7].

Basically, the initial generation of test cases is easy but improving quality requires some substantial effort. In our proposed approach, the Mutation Score and Path Coverage are used as the test adequacy measures to find out the good test case among the near infinite number of test cases in the solution space [3, 25, 34].

In this paper, we proposed a test optimization framework using Hybrid Genetic Algorithm (HGA) which has both GA's features and local search with memorization [17]. The framework includes two new heuristics for selecting the best parent for further generations.

Hybrid Genetic Algorithm (HGA) is a population based approach that combines genetic algorithm with local search technique for heuristic search in optimization problems [30]. In Hybrid Genetic Algorithm, initial heuristics are added for selecting the parent population. It has memorization function for remembering the best ancestors for back tracking. It also includes crossover operation that is not available in BA to retain the quality of both the parents. Now, the local optimum solution is moved towards the near global optimum solution. Thus, based on our experimentation results, the Hybrid Genetic Algorithm (HGA) proved to be better for the optimization of test cases when compared to the others. Even though the proposed approach consumes some additional amount of time (in fraction of milliseconds) because of the heuristics involved in decision making, it systematically improves the quality of test cases.

One of the ways to find out the quality of test cases is to artificially seed faults in the Software Under Test (SUT) and create many versions or mutants of it [3, 16, 19].

By using mutation analysis, the mutation score (the ability to detect seeded faults) of each test case is identified [19]. The mutation score is used to find the number of faults revealed out of the total number of seeded faults in the program [19]. In our approach, we applied it to various Java programs (from simple to complex) and found out the mutation score of each test case. Generally, if the initial test cases have a mutation score of 50–70 %, after improving the quality of test cases, the mutation score can reach up to 90–99 %.

The path coverage measure is used to determine the percentage of path covered by the test suite [3]. The effectiveness of the test cases was checked based on mutation score and path coverage metrics [3].

1.1 Definition

Test Case – A test case is a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE, do178b] [32].

In our proposed approach, the test case is a sequence of method calls with parameters [36] like *push*(10), *pop*(), *push*(2), *push*(−1), *pop*(), *pop*(), *push*(8), where push and pop are methods in Stack class.

Test Adequacy Criteria – To ensure the testing process, an empirical technical investigation is conducted to provide the adequacy of the test cases in testing the SUT. This may be statement coverage, branch coverage, condition coverage, mutation score etc. [3, 27].

Test Optimization – To maximize the profit of finding more bugs (mutation score) and coverage and to minimize the total number of test cases needed [9].

Mutation Testing – This is done by mutating certain statements in the source code and checking if the test case is able to find the errors [8].

Mutation operators – Following are some of the mutation operators object-oriented languages like Java, C++ etc. [3, 27]. We applied all these mutation operators in our system to generate mutants of our code.

- Changing the access modifiers, like public to private etc.
- Static modifier change.
- Argument order change.
- Super keyword deletion.
- Arithmetic and relational operator change.
- Parameter change in an expression.

Mutation Testing Tool – This tool brings a whole new level of error-detection to the software developer. By incorporating Mutation Testing into its state-of-the-art error-detection technology, Jester/MuJava are able to flush out more faults in Java programs than any other tool [12, 13].

The rest of the paper is organized as follows: Section 2 gives an introduction to optimization problem formation, Section 3 gives literature survey of the existing work, Section 4 describes the optimization using Simple Genetic Algorithm (SGA), Section 5 provides optimization using Bacteriologic Algorithm (BA) and Section 6 gives a detailed view of optimization using Hybrid Genetic Algorithm (HGA). Finally, in Section 7, the quality of the test cases in terms of their mutation score and path coverage based on Simple Genetic Algorithm (SGA), Bacteriological Algorithm (BA) and Hybrid Genetic Algorithm (HGA) are compared.

Our paper concluded that, when compared to the Simple Genetic Algorithm (SGA) and Bacteriological Algorithm (BA), the Hybrid Genetic Algorithm (HGA) produces quality test cases which can identify more seeded faults with higher path coverage.

2 OPTIMIZATION PROBLEM FORMULATIONS

2.1 Optimization

Utilizing the available resources as much as possible is called optimization. Optimization process is an incremental problem. At each and every evolution the solution leads to the target function [4].

2.2 Test Case Optimization

This means generating test cases that have the ability to reveal as many errors as possible from the Software Under Test (SUT) and to cover the Software Under Test (SUT) within less time and cost by selecting an effective set of few test cases from the universe of test cases. Here both mutation score (total number of seeded errors) and path coverage have to be maximized for each test case during test case generation. Selection of test cases is then done based on mutation score and coverage criterion [3, 27].

2.3 Issues of Interest

Improving quality of test cases through test case optimization: several research works have been conducted for the generation of effective test cases that can explore more errors in the SUT; but the issue here is not only generating test cases but improving the efficiency of test cases automatically from time to time. This indicates that the test case design process is a non-linear optimization problem and the application of evolutionary algorithms tends to solve it [2, 22]. Other non-evolutionary approaches such as statistical methods and probability based methods are quite complex and also require a lot of predefined assumptions in automated generation of efficient test data [26].

Christoph C. Michael et al. worked on dynamic test data generation using genetic algorithm [37]. Benoit Baudry et al. proposed genes and bacteria based test case optimization framework for the .NET environment [7, 29].

Our proposed approach focuses on test case optimization for object oriented applications in which each Class under Test (CUT) is a unit to be tested [1]. Our approach is applied to Java environment, in which each method of the Class under Test (CUT) is tested and the test is repeated under different execution conditions [36]. The test case that we designed here is the sequence of method calls with the parameters to be passed in them [36]. In our approach we assess the thoroughness of testing by means of test adequacy criteria, namely path coverage and mutation score.

It should be noted that, in testing classes, parameters may be necessary to invoke either the constructor, some of the methods that change the state of the object under test, or the method under test. If some of these parameters are in turn objects, they must be created and put into a proper state [9, 11].

Thus, a test case for the unit testing of a class consists of a sequence of object creations (object under test or parameters), method invocations (to bring objects to a proper state) and final invocation of the method under test [1]. For example, if we test method m of class A, a test case may be
$A\ a = new\ A();\ B\ b = new\ B();$
$a.m(45, b);\ a.m1(p1);\ a.m2(p2),$
$a.m1(p3);\ a.m2(p4).$

Consider that we had a binary search tree algorithm which is provided as a separate class. In this program we have Insert and Search procedures. Here the test cases should contain all possible combinations of accessing these two methods with different parameters. We know that this involves an exhaustive generation of test cases which is not possible in reality. Hence, we need to generate a basic set of test cases and then improve the quality of these test cases so that they can satisfy the specifications provided by the customers and at the same time exercises the entire software and reveals as many numbers of bugs as possible [31]. This is achieved by means of an automated generation and optimization of test cases using HGA.

2.3.1 Test Adequacy Criteria

The test adequacy criteria that we used are:

a) mutation score based test adequacy criterion:

   C1: A test T for program (P, P', R) is considered adequate, if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to R and can detect the mutant P' [27].

   To calculate mutation score, faults are artificially introduced into the Software Under Test (SUT) and create many versions (mutants) of it. The test cases should detect the newly introduced faults thereby exposing the similar kind

of existing faults in the software. The effective test cases can detect more faults.

$$\textit{mutation score} = (\textit{killed mutants})/(\textit{total mutants} - \textit{equivalent mutants}) \times 100$$

where *killed mutants* – Mutants detected by the test cases; *equivalent mutants* – Mutants not detected by the test cases.

b) path coverage based test adequacy criterion:

C2: A test T is considered adequate if it tests all independent paths. In case the program contains a loop, then it is adequate to traverse the loop body zero times or once [27].

$$\textit{path coverage}\,\% = (\textit{no. of paths Covered})/(\textit{total no. of independent paths}) \times 100$$

2.4 Test Case Optimization Problem Formulation

Given n test suites each consists of several test cases that must be processed on m test paths/sequences. The approach finds a set of test cases on the given set of test sequences taking into account the precedence constraints, which maximizes the mutation score and path coverage. The mutation score is identification of the number of artificially seeded errors in the SUT and path coverage is a measure to find the number of paths covered in the SUT.

Let $J = \{1, 2, \ldots, n\}$ be the set of test cases to be used and $M = \{1, 2, \ldots, m\}$ be the set of test sequences. Let MSj represent the mutation score of each test case j. The mutation score of each test case is represented by a vector $\prec MS1, MS2, MS3, \ldots, MSn \succ$.

Let $A(t)$ be the set of test cases being executed at time t and $rj, m = 1$ if test case j is suitable for test sequence m and $rj, m = 0$ otherwise. Now the conceptual model is represented as follows:

Max.

$$MSn(Fn) \tag{1}$$

$$Pcov(Fn) \tag{2}$$

Min.

$$Size(J) \tag{3}$$

Sub.to.

$$MSk > MSk - 1, k = 1, \ldots, n \tag{4}$$

$$\sum(rj, m) = 1, m > 0 \text{ belongs to } M \text{ and } j \text{ belongs to } J \tag{5}$$

$$MSj >= 0, j = 1, \ldots, n \text{ and belongs to } J \tag{6}$$

The first two objective functions maximize the mutation score and path coverage of test case n. The second objective function minimizes the test case set size. The constraint imposes the precedence relation between test cases and other constraints. It indicates one test sequence can process one test case at any point of time. Finally, the fourth constraint forces mutation score to be non-negative.

3 LITERATURE SURVEY

The earlier methods for testing are manual which consumes more cost, effort and time. These methods have no limit for testing and no proof for completion. Exhaustive testing leads to complex testing process [9].

3.1 Conventional Methods

The conventional methods for testing the software component are static testing by humans; under this the methods available [27] are desk checking, code walk through, formal inspection etc.

3.1.1 Problems in Conventional Methods

The manual testing process is slow and costly and is not effective in test case optimization. Exhaustive testing, that is testing infinitely is also not possible. And there is no proof of completion of testing and no maintenance of record for detected errors. The problems in manual testing are increased manual workload, cost and time. To reduce this, the testing process should be automated and optimized to produce selective few test cases. This can be achieved by taking minimum number of effective test cases and test the software using them [9, 27].

3.2 Evolutionary Algorithms Based Test Case Generation

McMinn investigated search based approach in test case generation [5]. The paper studies the application of search based techniques in generation and selection of test cases [5]. Also, McMinn et al. discussed state problem in applying evolutionary computation methods for test data generation [6].

In their paper Eugenia Diaz, Javier Tuya and Raquel Blanco applied Tabu search in software test data generation. The objective is to obtain branch coverage using program control graph. The Tabu search consists of two lists for memorizing the good and worst tests, respectively. The goal of the approach is to minimize the size of the test suite [28].

Roy P. Pargas et al. have applied genetic algorithm for test data generation [8]. Their paper presented a goal oriented technique for automatic test data generation. The approach uses genetic algorithm guided by the control dependencies in the program to search for test data that satisfies the test requirements. The GA conducts its search by constructing new test data from previously generated test data that are evaluated as good candidates. The method of goal oriented test case generation is done in this algorithm [8].

Mark Last et al. have introduced a new, computationally intelligent approach for the generation of effective test cases based on a novel, Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA). They identified good test cases from bad test cases based on their fault revealing capability; but the approach is application

dependent. The test configurations must be modified manually to construct the application specific fuzzy rules. Hence fuzzy rule base is not a generalized one and it is an application dependent one. The fuzzification and defuzzification processes consume a lot of time [21].

In [7], Benoit Baudry, Frank Fleurey et al. discussed the application of bacteriologic algorithm in software test data generation and selection. Mutation analysis is used as the basic concept to build confidence in test cases. The bacteriologic algorithm takes an initial set of test cases as input and a good set of test cases as output. This algorithm evolves incrementally, that is the algorithm builds final set incrementally by memorizing test cases that can improve the test set quality. Stopping criteria like after x generations, when the solution the set reaches a solution the set reaches a minimum fitness value, if the set's fitness value has not changed for a number of generations and so on [7] have been proposed. In our previous work [38], we employed Intelligent Agents for test sequence selection and optimization.

As shown in Figure 1, test case optimization can be automated using the emerging evolutionary algorithms like Simple Genetic Algorithm (SGA), Bacteriological Algorithm (BA) and Hybrid Genetic Algorithm (HGA).

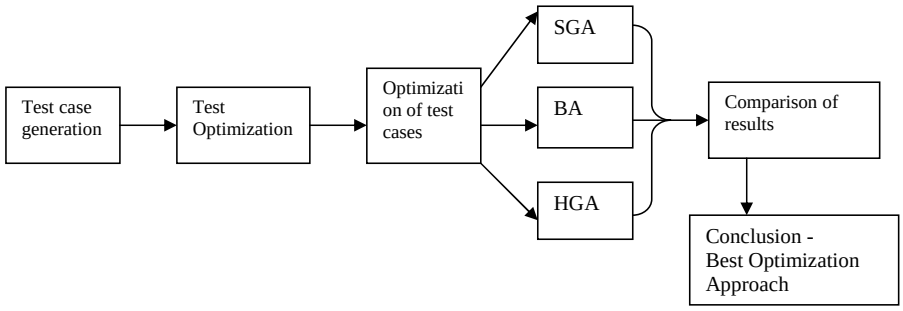

Fig. 1. Test case optimization using evolutionary algorithms

In our paper we applied SGA, BA and HGA for test case optimization and then the results were compared to find out the best optimization approach.

4 TEST CASE OPTIMIZATION USING SIMPLE GENETIC ALGORITHM (SGA)

Simple Genetic Algorithm (SGA) mimics the evolution of natural species in searching optimal solution. So, genetic algorithm can be exploited to produce test cases automatically [8, 20]. Test cases are described by chromosomes, which includes information on which object/component to create, which methods to invoke and which values to input. Genetic algorithms contain the functions crossover, mutation, selection and evaluation [10, 15].

The framework and pseudo code is shown in Figures 2 and 3. It describes the basic approach used in generating a population using Simple Genetic Algorithm (SGA).

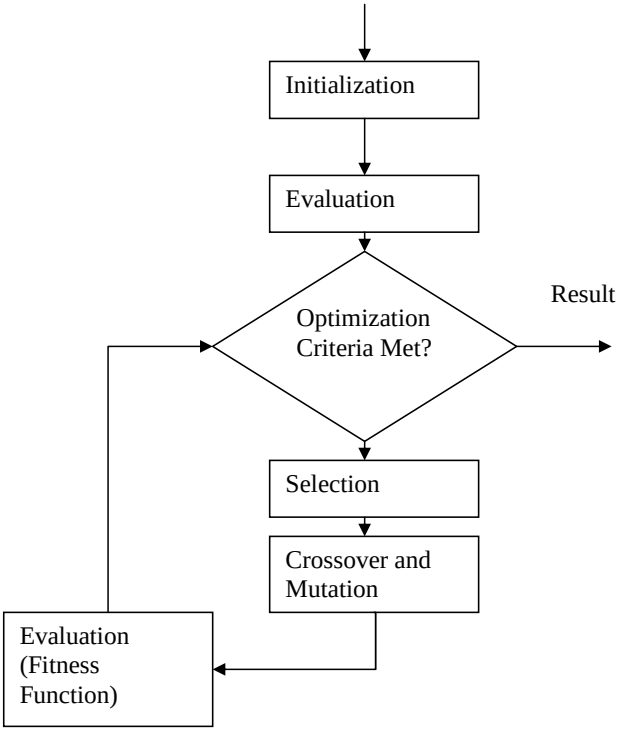

Fig. 2. Genetic algorithm framework for test case optimization

4.1 Test Case Construction – One Point Crossover and Mutation

4.1.1 One Point Crossover

This means fragmenting the selected population at some point m and recombining the $0 \ldots m-1$ portion of first member and $m \ldots n$ of the second member, as well as recombining the $0 \ldots m-1$ portion of the second member and $m \ldots n$ of the first member. It is shown in Figure 4 where $1 \preceq m \preceq k$, and $k =$ number of test cases.

For example,

Parent 1 – Test Case 1: $push(10), pop(), pop()$

Parent 2 – Test Case 2: $pop(), push(5), push(8)$.

After 1-point crossover at the second position, we get new generation of test cases as follows:

Pseudo-code of SGA algorithm

1. Choose initial population
2. Evaluate the fitness of each individual in the population
3. Repeat
   a. Select best-ranking individuals to reproduce
   b. Breed new generation through crossover and mutation (genetic operations) and give birth to offspring
   c. Evaluate the individual fitnesses of the offspring
   d. Replace parent with best ranked offspring
4. Until <terminating condition>

Fig. 3. Genetic algorithm pseudo code

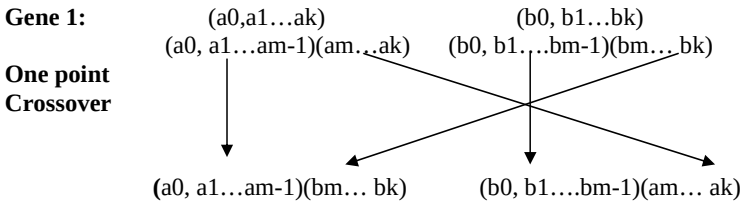

Fig. 4. One point crossover

Child 1 – Test Case 11: $push(10), push(5), pop()$

Child 2 – Test Case 21: $pop(), pop(), push(8)$.

4.1.2 Mutation

This operator is used to change the member at gene level and reproduce the remaining genes for the generation of offspring. In software testing the test case sets are considered as the population, individual test case is considered as member and method calls with values of variables are considered as the genes [10].

Parent 1 – Test Case 1: $push(10), pop(), pop()$

Parent 2 – Test Case 2: $push(7), push(5), push(8)$.

After mutation operator is applied to these test cases, the new generation of test cases is:

Child 1 – Test Case 11: $pop(), pop(), pop()$

Child 2 – Test Case 21: $push(7), pop(), push(8)$.

This new generation of test cases is then evaluated based on their effectiveness and then either selection or removal will be done.

4.1.3 Test Case Evaluation – Mutation Score Calculation and Path Coverage Criterion

The population having most favorable features can be assigned with higher fitness value for evaluation. In software testing the favorable feature of a test case is its ability to reveal as many errors as possible and higher path coverage. The test case with highest path and branch coverage criterion should be selected.

Child 1 – Test case 11's mutation score is 70 % and coverage% is 50 %

Child 2 – Test case 21's mutation score is 90 % and coverage% is 97 %

Parent 1's mutation score is 80 % and coverage% is 60 %

Parent 2's mutation score is 50 % and coverage% is 70 %.

4.1.4 Test Case Selection – Filtering Function

Selecting the best offspring as parent from the current population is called selection process. This leads to incremental solution generation. Test case that has higher fitness value when compared to the parent is selected as the parent for next generation.

After the evaluation is done, *Parent2* test case is replaced by Test Case 21.

Then during the next iteration this modified parent will be used for the generation of test cases (offspring).

4.1.5 Genetic Algorithm Implementation

Creating the initial set of test case is very easy, but improving the test case quality requires much effort. The basic set of test cases carries information that can be optimized to create better test cases by some cross-over and mutation of the test cases themselves.

At the beginning there is a population of mutant programs to be killed and a test cases pool. Those test cases are randomly combined to build an initial population of test cases. Then by applying GA operators the further generation of test cases is generated.

The fault revealing capability and path coverage measures of each of them are used to improve the quality of the test cases. Figure 5 shows the mutation score of each test case generated using GA. It indicates that the test cases have non-linear optimization.

The performance of GA in terms of path coverage is categorized into worst, average and best cases and is shown in Figures 6, 7 and 8.

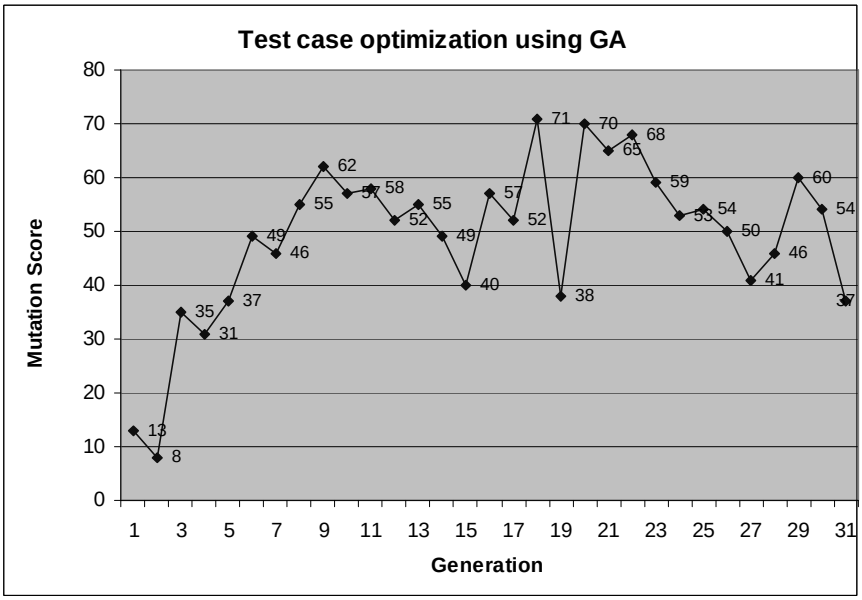

Fig. 5. Generation of test cases vs. mutation score using GA

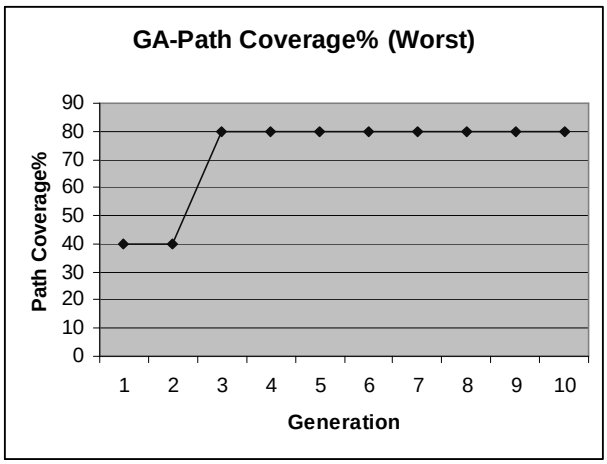

Fig. 6. GA based path coverage worst case

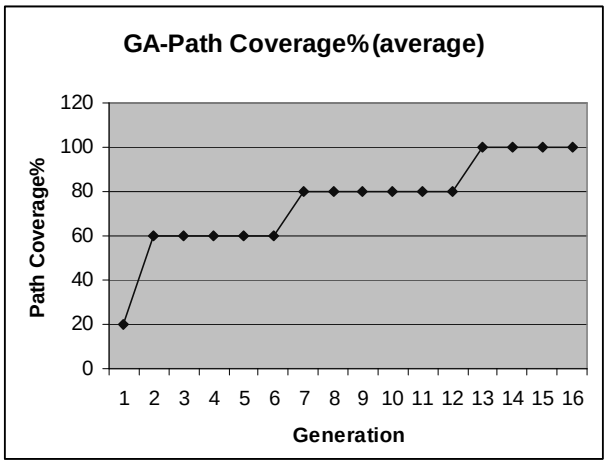

Fig. 7. GA based path coverage average case

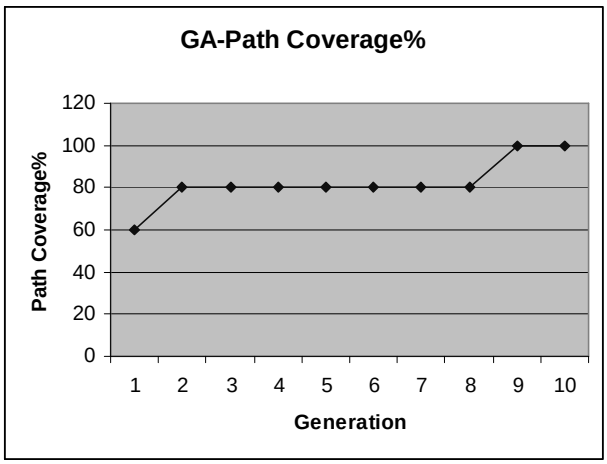

Fig. 8. GA based path coverage best case

4.1.6 Problems with SGA

The simple genetic algorithm, although simple, has some important drawbacks of no memorization, non linear optimization, risk of suboptimal solution and delayed convergence. There is also no guarantee for optimal solution even when it may be reached [7].

5 TEST CASE OPTIMIZATION USING BACTERIOLOGIC ALGORITHM

Bacteriologic algorithm is inspired by evolutionary ecology. One individual can not fit the whole environment and single perfect test case can not kill all mutants. This approach is more adaptive than Simple Genetic Algorithm (SGA). It aims at applying only the mutation operator to the initial population and the adaptation is based on small changes in the individuals. Here the individuals are called bacteria and correspond to atomic units. Bacteriologic algorithm contains mutation function, fitness function, filtering function and memorization function except crossover function [7, 29, 33].

As the Simple Genetic Algorithm (SGA), the fitness function is used to choose the best bacteria for reproduction. The selection process is an iterative one and it identifies best parents to generate a new population. The algorithm takes several bacteria which are mutated; then the best ones are selected to produce next generation. This process stops after a number of generations or when the memorized population has reached an optimum fitness value [7, 29, 33].

The bacteriologic algorithm takes an initial set of test cases as input, and outputs quality test cases. This algorithm is more stable than SGA. The test case reaches mutation score of up to 95 %; thus, BA is more adapted to test case optimization than GA [29, 33].

5.1 Test Case Construction – Mutation Function

This function generates a new test case by slightly altering an ancestor test case. Recursive application of this function may give the whole set of possible test cases (TC).

The bacteriologic algorithm takes an initial set of test cases as input and outputs a good set of test cases. This algorithm evolves incrementally. That is, the algorithm builds final set of test cases incrementally by memorizing test cases that can improve the set's quality. The stopping criteria may be number of generations, or minimum fitness value of the resultant set, or if the test set's fitness value has not changed for a number of generations and so on [29, 33].

It is used to generate test cases by applying mutation operator on all generations. This module generates the seed automatically by random permutation of initial methods. It creates required generations by mutation operation.

Test Case 1: $push(10), pop(), pop()$

Test Case 2: $pop(), push(5), push(8)$.

After Mutation:

Child 1 – Test Case 11: $pop(), pop(), pop()$

Child 2 – Test Case 21: $push(7), pop(), push(8)$.

After mutation, the *pop*() method of Test Case 2 is changed as *push*(7) and *push*(10) of Test Case 1 is changed as *pop*(). It is not due to interchange of methods, but the change of value in one gene position by another randomly chosen gene value.

5.2 Fitness Function – Mutation Score Calculation

Here mutation score is considered as the fitness function. Jester/muJava testing tool [12, 13] is used to create mutants in the application to be tested. Jester contains JUnit test suite and it provides information about the adequacy of JUnit test suite [12, 13]. Since the result produced is an XML file, an XML Parser code is written to collect the mutation score of each test case. Fitness Values:

Test Case 11's mutation score is 70 % and coverage% is 50 %

Test Case 21's mutation score is 90 % and coverage% is 97 %

Parent 1's mutation score is 80 % and coverage% is 60 %

Parent 2's mutation score is 50 % and coverage% is 70 %.

5.3 Test Case Selection – Filtering Function

This function is used to remove the test cases that are no more useful for further generations periodically from the bacteriologic medium to save the memory space during execution. The criteria to delete the test cases from the memory are size of the test cases and memorization threshold which we used here as the mutation score.

After the evaluation is done, *Parent2* test case is replaced by Test Case 21. Also, *Parent2* will be put up in back-up storage for keeping track of the best ancestor.

5.4 Memorization Function

This function is used to store all test cases for future reference. The test cases of all generations are stored in population database. The population of test cases is generated both by applying memorization and non-memorization in which it uses the current test cases as parents for the next generation. The performance of BA both with and without memorization is shown in Figure 9.

After the next iteration, if the test cases (offspring) generated were very poor and the *parent2* that has been already replaced has comparatively high score the worst child will be replaced by the best parent, i.e. *parent2*.

5.5 Drawbacks of the Bacteriologic Algorithm

Even though BA achieves 80 % to 97 % mutation score, there is a possibility of losing good parent's properties due to the absence of crossover operator. Also, the memorization operation in BA consumes lot of storage area.

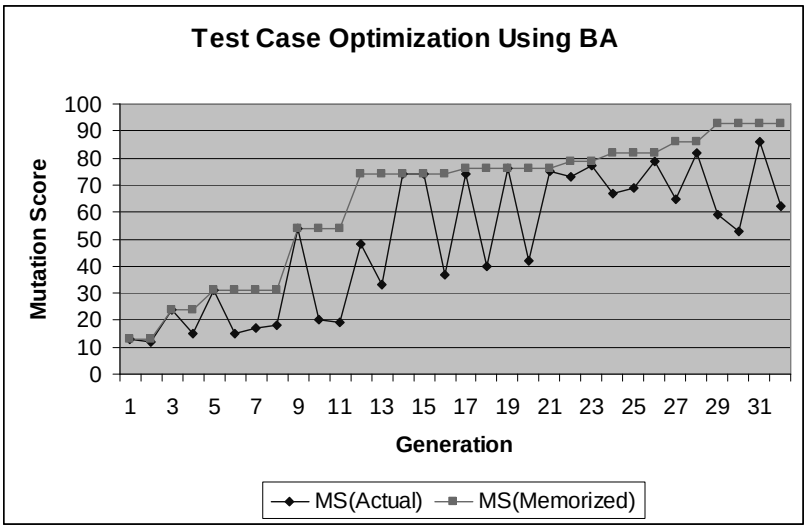

Fig. 9. Generation of test cases vs. mutation score using BA with and without memorization function

6 PROPOSED APPROACH – HGA BASED TEST CASE OPTIMIZATION FRAMEWORK

Our proposed framework uses Hybrid Genetic Algorithm (HGA) based approach for test case optimization. This involves the following steps:

1. Read the SUT.
2. Extract the test sequences/test paths in it.
3. Input them to Hybrid Tester.
4. Generate the test cases from the initial set of test cases, source code and test sequences using HGA.
5. Generate the test report based on the selected test cases.
6. Store the resultant test cases and test sequences/paths inside the test case repository/Test case Data Base (TDB).

The Hybrid tester performs the following tasks:

- Feasibility value generation for each test case.
- Mutant generation for the SUT.
- Coverage analysis based on the test path/sequence metric.
- Mutation Analysis based on the mutation score metric.

In the proposed framework as shown in Figure 10, the Software Under Test (SUT) and the optimal test sequences/paths which are generated from the Software Under Test (SUT) for ensuring the statement coverage criterion are taken as input. The test sequences are the set of independent test paths derived from the source code and are used as the basis for checking the coverage analysis of the software under test. The initial test cases are generated from source code by applying either by means of random generation or by means of manual entry. Also, it has a mutant generator which generates the mutated versions of the given SUT. Based on the coverage analysis and mutation score analysis the test case is either selected or rejected. The selected test cases are stored in the Test Database (TDB).

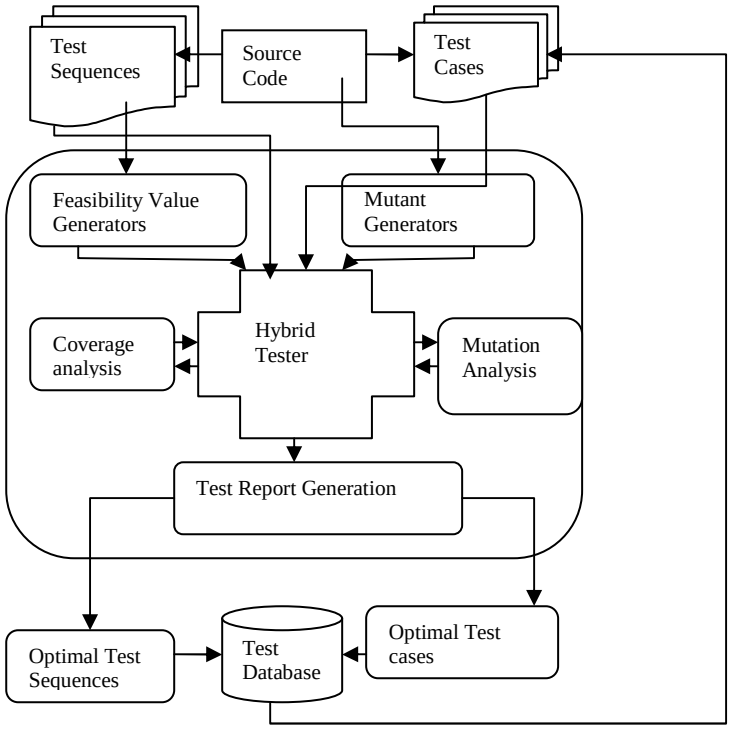

Fig. 10. HGA based test case optimization framework

6.1 Hybrid Genetic Algorithm (HGA) – Introduction

The Hybrid Genetic Algorithm (HGA) is also called Memetic algorithm [18]. It is a population based approach for heuristic search in optimization problems. They are more efficient than GA in which a local search algorithm is also included so that the final result will reach a nearly global optimum. In HGA, Genetic Algorithm is

used for attaining global optima and local search algorithms are used for attaining local optima [14].

Here the generations are called memes, not genes and they are processed and improved based on the local search algorithm employed by the persons. They are used to solve complex problems in which the shortest path to the goal is to be identified. They are also called meta-heuristic algorithms.

6.2 Generation of Test Cases Using HGA

All the genetic algorithms consist of the following main classes:

- chromosomal representation
- initial population generation
- crossover and mutation
- fitness evaluation
- selection.

Apart from these basic operations, HGA includes two new classes for local search:

- removeTop
- localBest.

The application of these classes to test case optimization is shown in Figure 11.

6.3 The Modified Algorithm Based on HGA with Improvement Heuristics

Step 1: Initialize population randomly

Step 2:

- Apply RemoveTop heuristic to all test cases in the initial population
- Apply LocalBest heuristic to all test cases in the initial population

Step 3: Select two parents based on their mutation score.

- Apply crossover and mutation operations between parents and generate offspring.
- Apply RemoveTop heuristic to each offspring.
- Apply LocalBest heuristic to each offspring.

Step 4: If (Mutation Score(offspring) $\leq$ Mutation Score (any one of the parents)) then replace the weaker parent by the offspring
Else Retain the existing parents

Step 5: Repeat steps 3 and 4 until end of specified number of iterations or the specified termination criterion is met.

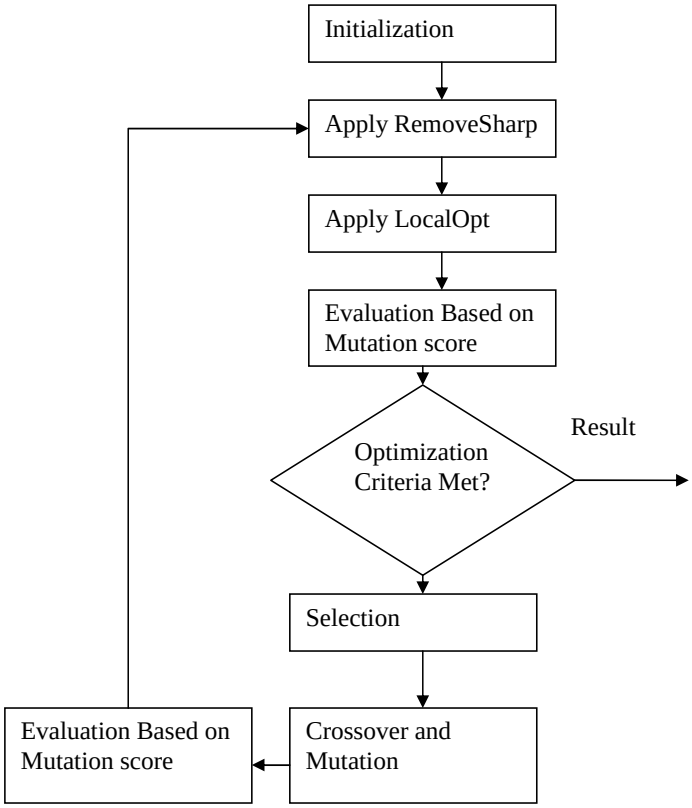

Fig. 11. Hybrid Genetic Algorithm

6.3.1 Pseudo Code of HGA

Step 1: Initial-Pop=Rand (seed). The seed is a test case given by the tester. Let $TC = \{tc1, tc2, \ldots, tcn\}$ be the set of test cases. Let $MS(tci)$ represent the mutation score of test case ti, for $i = 1$ to n.

Step 2:

a) Apply the following RemoveTop [14] algorithm to all test cases in the population:
$i \leftarrow 1$; min $\leftarrow MS(tc1)$;
repeat if $(MS(tci) \prec \text{min})$ then remove(tci);
$i \leftarrow i + 1$;
until $(i \succ$ *total number of test cases*$)$.

b) Apply the following LocalBest [14] algorithm to all test cases in the population:

$i \leftarrow 1; localopt = MS(tc1);$
repeat if $(localopt \prec MS(tci))$ then
$localopt \leftarrow tci; i \leftarrow i + 1;$
until $i \succ$ *total number of test cases*
if $(localopt \succ MS(parent1))$ then
$parent1 \leftarrow localopt;$
else if $(localopt \succ MS(parent2))$ then
$parent2 \leftarrow localopt;$
else retain the parents.

Step 3: Apply n-point crossover between parents and generate offspring. Mutate the selected test cases and generate additional offspring.

Step 4: Apply RemoveTop and LocalBest by repeating step 2 to each offspring.

Step 5: Repeat steps 3 and 4 until end of specified number of iterations or the spe cified mutation score is reached.

6.4 Chromosomal Representation

Each chromosome represents a legal solution to the problem and is composed of a string of genes. In the case of binary GA, the binary alphabet $\{0, 1\}$ is often used to represent these genes but sometimes, depending on the application, real GA in which integers or real numbers are used. For our problem of software testing we represent the chromosome as the stream of methods to be called in each class [4]. The sequences of operations are given here as the test cases.

Test cases are provided as

$$(method1(p1, p2, \ldots, pn), method2(p1, p2, \ldots, pn), methodn(p1, p2, \ldots, pn).$$

Random generation of test cases:

Test case 1: $push(10), pop(), pop()$

Test case 2: $pop(), push(5), push(8)$

Test case 3: $push(5), push(7), pop()$

Test case 4: $push(23), pop(), push(9)$

Test case 5: $pop(), pop(), pop()$

Test case 6: $push(), push(), push()$.

6.5 Selection of Parents from Initial Set of Test Cases

Hybrid Genetic Algorithm is designed to use heuristics for improvement of offspring produced by crossover. Initial population is randomly generated. The selection of parents for reproduction is done according to the probability distribution based on the individual's fitness values (*fiti*).

$$total\ fitness\ value = \sum fiti$$

where $i = 1$ to population size.

The mutation score and path coverage of each of the test cases is calculated and only the best parent is selected at this point. The generation of test cases will be done continuously and the total fitness value of each individual is evaluated.

6.6 Crossover

The crossover operator followed is n-point crossover and not one-point crossover. The total length of the parents is calculated and crossover through n points produces various offsprings. For each parent selected, a random integer number position in the range $[1 \ldots m-1]$ where m is the number of bits in a chromosome, indicates the crossing point. Now each pair of parents generates two new chromosomes called offsprings.

n-point crossover is applied to achieve high quality individuals.

Parent 1: Test case 3: $push(5), push(7), pop()$

Parent 2: Test case 4: $push(23), pop(), push(9)$

After n-point crossover is performed between the parents, the generated offsprings are:

Parent1 and *Parent2*:

Child1: $push(5), pop(), pop()$

Child2: $push(5), push(7), push(9)$

Child3: $push(5), push(9), pop()$.

Parent2 and *Parent1*:

Child1: $push(23), push(7), pop()$

Child2: $push(23), pop(), pop()$

Child3: $push(23), pop(), push(5)$.

6.7 Mutation

The crossover operator takes two individuals (parents) out of which one offspring is composed by combining two sub-portions, one from each parent. Unfortunately, these parts usually do not add up to complete members of the class to be tested. After the combination of the two sub-portions the redundant methods are deleted, and the missing methods have to be added at random positions to the gene structure to ensure that the offspring finally represents a correct genotype. Mutation operator modifies the gene structure by exchanging single method with its parameter.

Parent 1: Test case 3: $push(5), push(7), pop()$

Parent 2: Test case 4: $push(23), pop(), push(9)$.

After mutation,

Child1: $pop(), push(7), pop()$

Child2: $push(23), pop(), pop()$.

6.8 Fitness Evaluation

Fitness evaluation involves defining an objective or fitness function against which each chromosome is tested for suitability for the environment under consideration [5]. As the algorithm proceeds we would expect the individual fitness of the "best" chromosome to increase as well as the total fitness of the population as a whole. We have chosen the mutation score and path coverage criterion as the fitness values of the test cases.

Child1's mutation score is 70 % and coverage% is 50 %

Child2's mutation score is 90 % and coverage% is 97 %

Child3's mutation score is 45 % and coverage% is 37 %

...

Parent 1's mutation score is 80 % and coverage% is 60 %

Parent 2's mutation score is 50 % and coverage% is 70 %.

6.9 Assignment of Priorities to Test Cases

Mutation score and path coverage of all the offsprings produced by applying n-point crossover and mutation operators are calculated. Then the test cases are sorted based on the metrics mutation score and path coverage to find out the test cases with highest fitness values.

6.10 Local Search Procedure

The RemoveTop and LocalBest improvement heuristics are used to bring the offspring to a local optimum. If the fitness of the offspring thus obtained is greater than the fitness of any one of the parents then the parent with lower fitness is removed from the population and the offspring is added to the population. If the fitness of the offspring is lesser than that of both of its parents then it is discarded.

6.10.1 RemoveTop Heuristic

The offspring which leaves more mutants in survival will be deleted from the memory, i.e. the test cases that have very low mutation score will be removed. Test cases with higher mutation score will be saved [14].

This heuristic deletes the test case 3, which has very low mutation score (45 %) and code coverage 37 % and other test cases with score below 50 %.

6.10.2 LocalBest Heuristic

At every generation of offspring by the n-point crossover and mutation, the offspring which has the highest mutation score is selected as local optimum. This local optimal solution is compared against the parents. If any of the offsprings is better than any of the parents then the weakest parent will be replaced by the optimal offspring [14]. The selected test cases are stored in the repository for regression testing [14].

Here test case 2 is identified as the local optimum and is removed and stored. This is necessary for selecting the next best test cases from the population; otherwise the algorithm will try to select the same test case repeatedly.

6.11 Selection of Offspring

Now the best individuals which are selected as part of the above procedure are used to replace the worst parents in the next iteration of test case generation.

7 IMPLEMENTATION – TEST CASE OPTIMIZATION USING HGA BASED OPTIMIZATION

7.1 Test Case Initialization

Initial population is generated by getting the total number of methods in the SUT and the names of the methods. Then the values to be passed as parameters to these methods are generated randomly based on the user preferences. This procedure generates the initial population by randomly selecting the methods and parameters to be passed in those methods.

7.2 Test Case Evaluation – Mutation Score Calculation

Once the initial population is generated, the mutation score of each individual is calculated by using this module. We generated a finite number of mutants of the code using muJava/Jester tool [12, 13]. We used the tool Jester for finding the mutation score. The output produced by Jester tool is an XML file that has the mutation score of each of the test cases generated.

7.3 LocalBest and RemoveTop Algorithm

LocalBest: This procedure finds the local best solution from each generation and removes it from the current population. Then it stores that in the parent table as one of the best offspring. If this heuristics is not used then the algorithm will tend to choose the same offspring every time it is the optimal one.

RemoveTop: This procedure finds the offspring which leaves more number of mutants in survival will be deleted from the memory. Thus, the test cases that have very low mutation score will be removed. Test cases with higher mutation score will be saved.

7.4 Final Test Case Generation – Filter Function

This module takes the test cases from the pool and filters only the test cases that have the highest mutation score. Here we set the maximum score as 97 %. They are stored in the optimal test cases repository for further generations. Figure 12 shows the mutation score of test cases over generations. We can see that the quality of test cases is improved over generations.

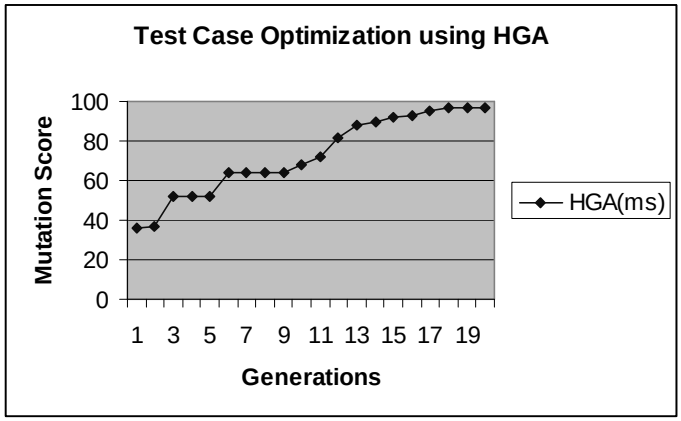

Fig. 12. Generation of test cases vs. mutation score using HGA

7.5 Experimentation Results

A comparative evaluation has been made between HGA based test optimization (the proposed technique), the GA based test optimization and the BA based test optimization. For comparison the following parameters were used: the number of test cases, mutation score of test cases in each generation, path coverage percentage of test cases and the number of generations. The experimentation settings are shown in Table 1.

Based on this setting, several benchmark problems have been evaluated based on the results obtained using the three approaches.

The mutation score of the test cases generated using Simple Genetic Algorithm, Bacteriologic Algorithm and Hybrid Genetic Algorithm were compared and shown in Tables 2 and 3. We can infer from the tables that HGA produces better results when compared to SGA, BA and BA (Memorized).

♯	Parameter	SGA	BA	HGA
1.	N	100	100	100
2.	L	4	4	4
3.	Ngen	200	200	200
4.	Pc	0.9	No crossover	0.9
5.	Pm	0.01	0.01	0.01
6.	Sm	MS	MS	MS
7.	Cm	1-point	-Not applicable-	n-point
8.	Mm	M and A	M and A	M and A
9.	MinLT	-Nil-	Memorization	Memorization
10.	Rr	0.2	0.2	0.2
11.	MaxLT	-Nil-	CH	fit

Table 1. Experimental settings; N = *population size*; L = *chromosome length*; $Ngen$ = *total number of generations*; Pc = *crossover probability adaptive*; Pm = *mutation probability*; Sm = *selection method*; MS = *mutation score based*; Cm = *crossover method*; Mm = *mutation method*; $MinLT$ = *minimum lifetime (number of generations)*; Rr = *reproduction ratio*; $MaxLT$ = *maximum lifetime (number of generations)*; M and A – *method and argument change*; CH = *complete history*; fit = *fitter individuals greater than the specified mutation score*

Generation	SGA (ms)	HGA (ms)	Generation	SGA (ms)	HGA (ms)
0	13	36	13	40	83
1	8	37	14	78	85
2	20	52	15	69	92
3	31	52	16	75	94
4	37	52	17	79	95
5	49	64	18	59	95
6	46	64	19	77	95
7	36	64	20	67	96
8	40	64	21	53	96
9	34	68	22	79	97
10	22	72	23	51	97
11	60	78	24	69	97
12	45	79	25	72	98

Table 2. Mutation Score of GA and HGA for sample test cases

Generation	BA	BA (Mem)	HGA	Generation	BA	BA (Mem)	HGA
0	13	13	36	6	17	31	64
1	12	13	37	7	18	31	64
2	24	24	52	8	54	54	64
3	15	24	52	9	20	54	68
4	31	31	52	10	19	54	72
5	15	31	64	...	...	...	...

Table 3. Mutation score of BA, BA (memorized) and HGA

7.6 Comparison Charts

The tests were conducted on various Java programs and compared the performance of each of the evolutionary algorithms. Previously in [35], a genetic algorithm based approach was applied to generate test cases for Java programs.

The path coverage details using HGA and BA in Figure 13 indicate that when compared to BA, the test cases generated using HGA show higher path coverage. Similar to that, Figure 14 indicates the path coverage of test cases generated using SGA and HGA. The chart shows that HGA based test cases shows better performance when compared to SGA based ones.

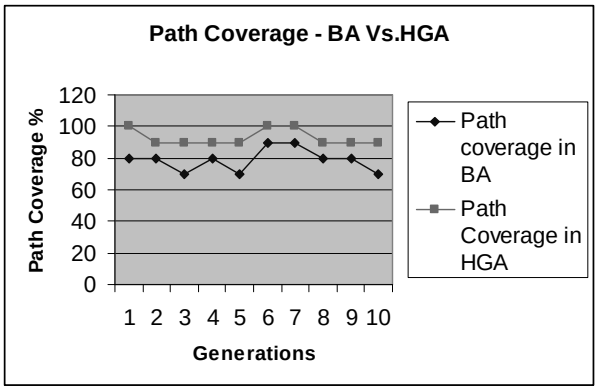

Fig. 13. Comparison of HGA and BA based on path coverage

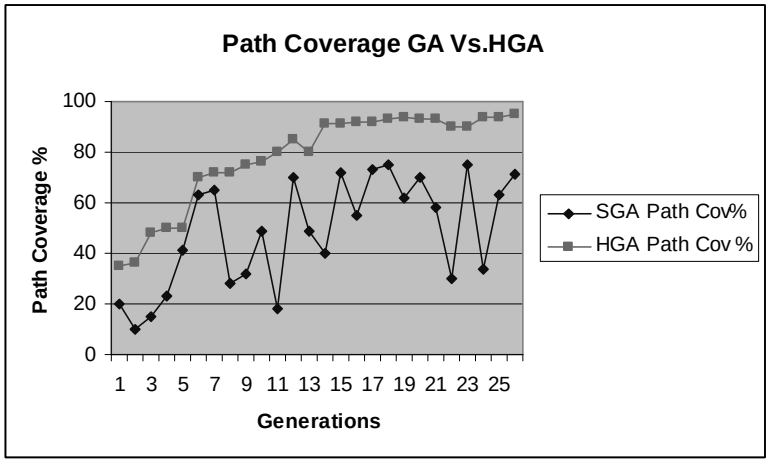

Fig. 14. Comparison of SGA/GA and HGA based on path coverage

The mutation score details using HGA and GA shows that, when compared to GA, the test cases generated using HGA were able to identify more seeded faults.

Similar to that, the comparison chart in Figures 15 and 16 shows that the test cases generated using HGA show higher mutation score than GA and BA.

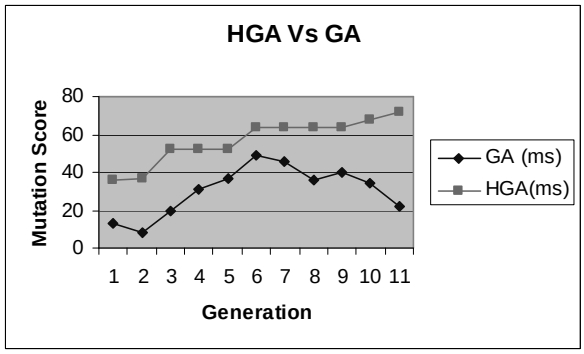

Fig. 15. Comparison of SGA/GA and HGA based on mutation score

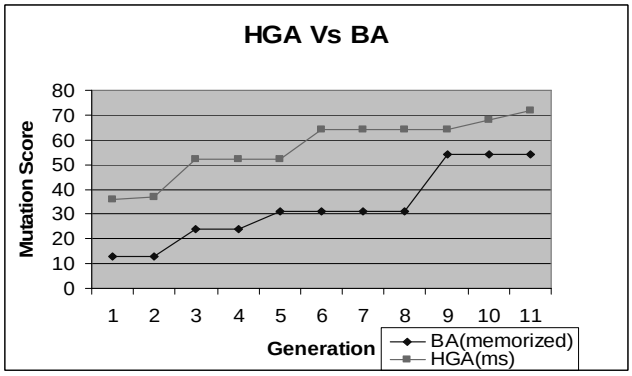

Fig. 16. Comparison of BA and HGA based on mutation score

The tested problems including academic and real-time problems were taken and evaluated. The results were shown in Tables 4 and 5.

Hence, it is evident from Tables 4 and 5 that Hybrid Genetic Algorithm performs well and generates nearly global optimum solutions. The complexity of the real-time problems is increased in terms of their Source Lines of Code (SLOC), number of methods/class and number of classes.

Areas	HGA		GA		BA	
Problem	BST	Stack	BST	Stack	BST	Stack
PTC	200	100	200	100	200	100
RTC	5	5	120	50	5	5
MS	98	99	79	70	95	96
PCTC	5	5	120	50	5	5
NGEN	50	50	200	200	50	50
TOpt	0.233432		3.699231		0.1213152	
MSPACE	Less (only the LO)		No		High	
MSC	SM		AM		SM	

Table 4. Evaluation results of simple academic problems;
BST = binary search tree; *PTC = possible test cases*;
RTC = reduced test cases; *MS = mutation score of test cases*;
PCTC = the number of test cases that covered the path;
NGEN = the number of generations; *TOpt = time taken for optimization*;
MSpace = memory space occupied due to memorization; *LO = local optimum*;
MSC = mutation score computation; *SM = survived mutants*; *AM = all mutants*

Areas	HGA		GA		BA	
Problem	CCV	FD-BT	CCV	FD-BT	CCV	FD-BT
PTC	20 000	40 000	20 000	40 000	20 000	40 000
RTC	200	250	2 500	10 000	220	380
MS	96	97	79	70	95	96
PCTC	180	330	550	775	200	340
NGEN	50	50	200	200	50	50
TOpt	0.832538		6.733234		0.7112759	
MSPACE	Less (only the LO)		No		High	
MSC	SM		AM		SM	

Table 5. Evaluation results of real time problems; *CCV = credit card validation*;
FD − BT = fraudulent detection in banking transaction;
PTC = possible test cases; *RTC = reduced test cases*;
MS = mutation score of test cases;
PCTC = the number of test cases that covered the path;
NGEN = the number of generations; *TOpt = time taken for optimization*;
MSpace = memory space occupied due to memorization; *LO = local optimum*;
MSC = mutation score computation; *SM = survived mutants*; *AM = all mutants*

8 CONCLUSIONS

The evolutionary algorithms like genetic, bacteriologic and hybrid genetic algorithms are implemented. The test cases are optimized based on path coverage, mutation score as fitness values. Seed population is randomly generated and 50 to 200 further generations are generated using GA, BA and HGA. Path coverage (in per cent) by the test cases is verified. Mutation score of each test case is calculated. It is clear from the comparison charts that GA produces non linear optimal path, suboptimal solution, BA provides linear and optimal solutions, and HGA produces nearly global optimal and linear optimal solutions with rapid convergence. Hence, comparatively Hybrid Genetic Algorithm (HGA) produces nearly optimum solution.

REFERENCES

[1] Binder, R. V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley 2000.

[2] Briand, L. C.: On the Many Ways Software Engineering Can Benefit from Knowledge Engineering. Proc. 14th SEKE, Italy, 2002, pp. 3–6.

[3] Horgan, J.—London, S.—Lyu, M.: Achieving Software Quality with Testing Coverage Measures. IEEE Computer, Vol. 27, 1994, No. 9, pp. 60–69.

[4] Dixon, L. C. W.–Szegö, G. P.: Towards Global Optimization. Elsevier, 1978, pp. 1–15.

[5] McMinn, P.: Search-Based Software Test Data Generation: A Survey. Software Testing, Verification and Reliability, Vol. 14, 2004, No. 2, pp. 105–156.

[6] McMinn, P.—Holcombe, M.: The State Problem for Evolutionary Testing. Proc. GECCO 2003, Springer Verlag, 2003, LNCS, Vol. 2724, pp. 2488–2500.

[7] Baurdy, B.—Fleurey, F.—Marc, J.—le Traon, Y.: Automatic Test Case Optimization: A Bacteriologic Algorithm. IEEE Software, 2005, pp. 76–81.

[8] Pargas, R. P.—Harrold, M. J.—Perk, R. R.: Test Data Generation Using Genetic Algorithm. Journal of Software Testing, Verification And Reliability, 1999, pp. 1–19.

[9] Desikan, S.—Ramesh, G.: Software Testing Principles and Practices. Pearson, 2002.

[10] Tonella, P.: Evolutionary Testing of Classes. ISSTA-2004, pp. 11–14.

[11] Pressman, R. S.: Software Engineering – A Practitioner's Approach. McGraw Hill, International sixth edition.

[12] `www.jester.com`.

[13] `www.mujava.com`.

[14] Jayalakshmi, G. A.—Sathiamoorthy, S.—Rajaram, R.: A Hybrid Genetic Algorithm – A New Approach to Solve Traveling Salesman Problem. International Journal of Computational Engineering Science, Vol. 2, 2001, No. 2, pp. 339–355.

[15] Seesing, A.: EvoTest: Test Case Generation using Genetic Programming and Software Analysis. Thesis, Delft University of Technology.

[16] Bradbury, J. S.: Using Mutation for the Assessment and Optimization of Tests and Properties. Doctoral Symposium in conjunction with the International Symposium on Software Testing and Analysis (ISSTA 2006), pp. 4.

[17] Kelner, V.—Capitanescu, F.—Leonard, O.—Wehenkel, L.: A Hybrid Optimization Technique coupling Evolutionary and Local Search Algorithms. Journal of Computational and Applied Mathematics, Vol. 215, 2008, No. 2, pp. 448–456.

[18] Krasnogor, N.—Smith, J.: A Tutorial on Competent Memetic Algorithms: Model, Taxonomy and Design Issues. IEEE Transactions on Evolutionary Computation, Vol. A, No. B, CCC200D, 2005.

[19] Offutt, J.—Ma, Y.-S.—Kwon, Y. R.: An Experimental Mutation System for Java. ACM SIGSOFT Software Engineering Notes, Vol. 29, 2004, No. 5, pp. 1–4.

[20] Berndt, D.—Fisher, J.—Pinglikar, J.—Watkins, A.: Breeding Test Cases with Genetic Algorithms. 36th Hawaii International Conference on System Services (HICSS '03), IEEE, 2003, p. 338.

[21] Last, M.—Eyal, S.—Kandel, A.: Effective Black-Box Testing with Genetic Algorithms. Book on Hardware and Software, Verification and Testing, Lecture Notes in Computer Science, 2006, pp. 134–148.

[22] Sofokleous, A. A.–Andreou, A. S.: Batch-Optimistic Test-Cases Generation Using Genetic Algorithms. Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence – ICTAI 2007, Vol. 1, pp. 157–164.

[23] Hangal, S.—Lam, M. S.: Tracking Down Software Bugs Using Automatic Anomaly Detection. ICSE 2002, pp. 291–301.

[24] Zhu, H.—Hall, P.—May, J.: Software Unit Test Coverage and Adequacy. ACM Computing Ssurveys, Vol. 29, 1997, pp. 366–427.

[25] Lin, J. C.–Yeh, P. L.: Using Genetic Algorithms for Test Case Generation in Path Testing. Proceedings of ATS 2000, pp. 241–246.

[26] Banks, D.—Dashiell, W. et al.: Software Testing by Statistical Methods – Preliminary Success Estimates for Approaches based on Binomial Models, Coverage Designs, Mutation Testing, and Usage Models. Technical Report, National Institute of Standards and Technology, Information Technology Laboratory, 1998.

[27] Mathur, A. P.: Foundations of Software Testing. Pearson Education, 2008.

[28] Daz, E.—Tuya, J.—Blanco, R.: A Tabu Search Algorithm for Structural Software Testing. Computers and Operations Research, Vol. 35, 2008, No. 10, pp. 3052–3072.

[29] Baudry, B.—Fleurey, F.—Le Traon, Y.—Jézéquel, J.M.: An Original Approach for Automatic Test Cases Optimization: A Bacteriologic Algorithm. IEEE Software, Vol. 22, 2005, No. 2, pp. 76–82.

[30] Land, M.: Evolutionary Algorithms with Local Search for Combinatorial Optimization. Ph. D. Thesis, University of California, San Diego, 1998.

[31] Xie, T.—Marinov, D.—Notkin, D.: Improving Generation of Object-Oriented Test Suites by Avoiding Redundant Tests. Technical Report UW-CSE-04-01-05, University of Washington, Department of Computer Science and Engineering, Seattle, WA, January 2004.

[32] Kamde, P. M.—Nandavadekar, V. D.—Pawar, R. G.: Value of Test Cases in Software Testing. IEEE International Conference on Management of Innovation and Technology, 2006, pp. 668–672.

[33] Baudry, B.—Fleurey, F.—LeTraon, Y.—Jézéquel, J. M.:. From Genetic to Bacteriological Algorithms for Mutation-Based Testing. Software Testing Verification and Reliability, Vol. 15, 2005, No. 2, pp. 73–96.

[34] McMinn, P.—Harman, M.—Binkley, D.—Tonella, P.: The Species Per Path Approach to Search Based Test Data Generation. ISSTA 2006, pp. 13–24.

[35] Khurshid, S.—Marinov, D.: TestEra: Specification-Based Testing of Java Programs Using SAT. Automated Software Engineering Journal (JASE 2004), Vol. 11, No. 4, pp. 403–434.

[36] Tonella, P.: Evolutionary Testing of Classes. ISSTA 2004, pp. 119–128.

[37] Michael, C. C.—McGraw, G.—Schatz, M.: Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, Vol. 27, 2001, No. 12, pp. 1085–1110.

[38] Jeya Mala, D.—Mohan, V.: On the Use of Intelligent Agents to Guide Test Sequence Selection and Optimization. International Journal of Computational Intelligence and Applications, Vol. 8, 2009, No. 2, pp. 155–179.

Dharmalingam Jeya Mala has received her Ph. D. degree from Anna University, Chennai, Tamil Nadu, India and completed her Masters degree in computer applications, Master of Philosophy in computer science from Madurai Kamaraj University, Madurai, Tamil Nadu, India. She has more than ten years of academic, research and industrial experiences. She published nearly twenty papers in international journals and Conferences. She is currently working as an Assistant Professor in Thiagarajar College of Engineering, Madurai, Tamil Nadu, India. She has received awards and recognitions from industries like IBM, Honeywell and Microsoft. Her research interests include software engineering, software testing, optimization, artificial intelligence, soft computing, service oriented architecture (SOA), object-oriented analysis and design and grid based optimization.

Vasudev Mohan received his Ph. D. from the Indian Institute of Technology (IIT), Bombay in 1978. He has teaching and research experience of more than three decades in Thiagarajar College of Engineering, Madurai, Tamil Nadu, India. He has published more than twenty five research papers in international and national journals. Currently he is supervising ten Ph. D. scholars. His research interests include computational methods, soft computing techniques, software engineering, software testing, optimization and database Management Systems.

Elizabeth Ruby received her Masters Degree in computer science and engineering from Thiagarajar College of Engineering, Madurai, Tamil Nadu, India. She published papers at international conferences. Her research interests include software testing, software test optimization and soft computing.