CHAPTER 5

HYBRID TESTER – HYBRID GENETIC ALGORITHM BASED TEST CASE OPTIMIZATION

5.1 PROBLEM FORMULATION

Software development organizations spend considerable portion of their budget and time in testing related activities. The effectiveness of the verification and validation process depends upon the number of errors found and rectified before releasing the software to the customer side. This in turn depends upon the quality of test cases generated. Through the years, a number of different methods have been proposed for generating test cases.

A test case is a set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement (After IEEE 610).

The objective of software test planning is to identify the test cases for testing the given software. But, for software having much input attributes, the number of combinatorial test cases is so high and it is impossible to test and execute all of them due to resource limitation. Also, having a huge number of test cases does not mean that all the errors in the SUT can be identified. Eric et al (1998) analyzed the effect of fault detection of the test set when its size is minimized. In their approach, they minimized the size of the test set by systematically improving the effectiveness of the test cases over several iterations. Finally, they concluded that, regardless of the way of how a
test set is generated, test sets that are minimized by improving their effectiveness have advantages in terms of reduction in test set size and with almost the same fault detection effectiveness over the original non-minimized test set.

Hence, the solution is to find a criterion, for choosing the most important and effective test cases and removing the redundant and unnecessary ones. The effectiveness of the test cases is depending on the application, testing approach or the availability of the testing resources.

The objective of test case optimization is to reduce the number of test cases and improve the effectiveness of test cases. Since, the software test optimization problem is a multi-objective optimization problem, it cannot be solved within combinatorial time and hence it is NP-hard. And the solution for these types of NP-hard problems cannot be obtained by a direct search; rather, the search should be done by means of heuristics guided searching techniques.

Hence, for test case optimization approach, this thesis proposed a heuristics guided population based search approach. The basic reason for applying population based search approaches is that, they generate a pool of test cases during each iteration of solution generation and a search is conducted at the end of each iteration to filter quality test cases. These selected test cases then act as parents for the next iteration. Ultimately, the quality of the test cases is steadily improving and finally, only a few efficient quality test cases are stored in the optimized test case repository.

From the literature study, it has been identified that, HGA has been applied for several hard optimization problems. Hence, its functionality has been extended in this thesis, to do test case optimization which is also a hard optimization problem.
Hybrid Genetic Algorithm (HGA), a population based search approach, combines the properties of Genetic Algorithm (GA) and Local Search (LS) procedure to achieve optimization. Since, GA is used to attain global optimal solution and LS is used to attain local optima, the proposed framework that combines these two approaches provide an optimal or near optimal solution.

Since, HGA is a population based search approach, the effectiveness of the test cases is improved during each generation in terms of the test adequacy criteria namely mutation score and path coverage.

Mutation score is a measure used to indicate the number of faults identified out of seeded faults in the SUT (Aditya P. Mathur 2008). Path coverage measure is used to find out the number of independent paths covered out of total number of independent paths in the SUT (Aditya P. Mathur 2008). Finally, test case optimization is achieved by storing only a few efficient test cases in the optimal test case repository. Further, the regression testing process is improved, using these efficient test cases available in the optimal test case repository.

5.2 REPRESENTATION OF THE SEARCH SPACE – FORMAL PROBLEM DEFINITION

The proposed approach focuses on automating the quality improvement of test cases using test case optimization. The search space of test case optimization problem includes a pool of test cases which comprises of near infinite number of test cases and the SUT. The problem here is to identify few efficient test cases from it.

In this research work, Mutation Score and Path Coverage are used as the test adequacy measures to find out the efficient test cases among the
near infinite number of test cases in the search space. Jeremy S. Bradbury wrote a technical report (2006), on applying mutation testing for the assessment and optimization of tests and properties in software. The report focuses on the empirical evaluation of synergies between testing and property-based analysis. Bruno et al (2007) used path coverage as a metric to evaluate the test cases in GEO (Generalized Extremal Optimization) based software test data generation technique. Also, several other research works have also applied these two measures as test adequacy criteria for generating test cases. Hence, in the proposed approach mutation score and path coverage are applied as the test adequacy measures to optimize the test cases in the search space.

The proposed approach applied mutation analysis to object oriented classes, specifically written in Java, to find out the mutation score of each test case. For efficient mutation analysis, various mutants (modified programs) are created using Offutt et al’s (2004) mutation operators. After the mutants are created, the proposed approach evaluates the effectiveness of the test cases based on the mutation score (number of killed mutants). The quality of the test cases is improved through the iterations based on the mutation score and path coverage based test adequacy criteria.

Basically, the initial generation of test cases is easy because of random generation of test cases but improving their quality requires some substantial effort. Generally, if the initial set of test cases has a mutation score of 50-70%, after improving the quality of test cases, the mutation score can reach up to 90-99%.

Several research works have been conducted for the generation of test cases that can exercise the SUT by revealing as much number of errors as they can. But the focus here is not only on generating test cases, but also on improving the efficiency of test cases automatically from time to time. This
indicates that, the test case generation process is a non-linear optimization problem and the application of evolutionary algorithms tends to solve it.

Evolutionary Algorithms (EA) play a vital role in solving most of the optimization problems (McMinn 2006). Among them, Simple Genetic algorithm (SGA) is the easiest and flexible one and has been extensively applied by many researchers to solve several hard optimization problems including software testing (Pargas et al 1999). In recent years, a new swarm intelligence based meta-heuristic called Bacteriologic Algorithm (BA) has been applied to software test optimization. This algorithm includes memorization without crossover operation and provides better solution than GA (Baudry et al 2005(a) and 2005(b)).

In this research work, the search has been conducted by means of an improved meta-heuristic search technique based on Hybrid Genetic Algorithm (HGA). This Meta-heuristic, combines the best properties of GA and BA, and is proposed to analyze the search space. The proposed Hybrid Genetic Algorithm has been enhanced with improvement heuristics namely ‘LocalOpt’ and ‘RemoveSharp’ to aid the search process in selecting the best parent for further generations. It has memorization function to remember the best ancestors for back tracking. Also it includes crossover operation to retain the quality of both the parents. Now, the local optimal solution is moved towards the near global optimal solution.

Based on the experimentation results, the Hybrid Genetic Algorithm (HGA) has been proved to be better for test case optimization when compared to the other approaches based on GA and BA. Even though, the proposed approach consumes some more additional amount of time (in fraction of milliseconds) because of the heuristics involved in the decision making process; the quality improvement of test cases is high due to the systematic improvement of the quality of test cases.
5.2.1 Problem Environment

Given a set of ‘n’ test cases; that must be processed on ‘m’ test paths/sequences, the proposed approach finds a subset of test cases that covers the given set of test paths/sequences taking into account the precedence constraints, which maximizes the mutation score and path coverage.

Let \( J = \{1, 2...n\} \) be the set of test cases to be used and \( M = \{1, 2...m\} \) be the set of test sequences. Let \( MS_j \) represents the mutation score of test case ‘j’. The mutation score of each test case is represented by a vector \( <MS_1, MS_2, MS_3...MS_n> \). And \( r_{j,m} = 1 \) if test case ‘j’ is suitable for test sequence ‘m’ and \( r_{j,m} = 0 \) otherwise.

5.2.2 Assumptions

The Software under Test (SUT) is written in any of the Object Oriented programming languages (C++/Java etc.), especially in Java. It has no syntax errors (Compilation Errors).

5.2.3 Objective Criterion

The objective is to generate quality test cases that have the ability to reveal as much number of errors as possible from the SUT and can cover the SUT within less time and cost. Here the mutation score and path coverage based test adequacy criteria should be maximized for each test case in the process of test case generation and the size of the final test case set should be minimized.
5.2.4 Mathematical Model

Max.

\[ MS_n(F_n) \]  \hspace{1cm} (5.1) \\
\[ PCov_n(F_n) \]  \hspace{1cm} (5.2)

Where \( F_n \) is the name of a test case set; \( MS_n \) – Mutation Score of test case ‘n’; \( PCov_n \) – Path coverage of test case n.

Min.

\[ Size (J) \]  \hspace{1cm} (5.3)

Sub. to

\[ MS_k \geq MS_{k-1} \]  \hspace{1cm} k = 1 to n \hspace{1cm} (5.4)

\[ \sum r_{j,m} \leq 1, \quad m > 0 \text{ belongs to ‘M’ and ‘j’ belongs to ‘J’} \] \hspace{1cm} (5.5)

\[ MS_j \geq 0 \]  \hspace{1cm} j = 1 to n and belongs to ‘J’ \hspace{1cm} (5.6)

\[ PCov_n \geq 0 \]  \hspace{1cm} ‘n’ is a test case in the test case set \hspace{1cm} (5.7)

\[ \text{Mutationscore} (MS) = \frac{\text{(dead mutants)}}{\text{(total mutants - equivalent mutants)}} \times 100 \] \hspace{1cm} (5.8)

\[ \text{PathCoverage} \% (PCov) = \frac{\text{(No. of paths covered)}}{\text{(Total No. of paths)}} \times 100 \] \hspace{1cm} (5.9)

where, Mutants - Changed program code,
Dead Mutants - Mutants which are killed by the test cases and
Equivalent Mutants - Mutants that cannot be identified / killed by any of the test cases.
The two objective functions (5.1) and (5.2) maximize the mutation score and path coverage of test case ‘n’. The objective function in (5.3) minimizes the size of the test case set. Constraint (5.4) focuses on the comparison of mutation score of test cases. The constraint (5.5) 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. Constraint (5.6) forces mutation score to be non-negative. The constraint (5.7) shows the value, that path coverage metric takes based on the coverage of test case n. The calculation of mutation score and path coverage are given in the formulae (5.8) and (5.9).

5.3 RELATED WORK

From the literature study, it has been identified that, some of the non-evolutionary approaches such as statistical methods and probability based methods are quite complex and also requires a lot of predefined assumptions in automated generation of optimal test data (David et al 1998, Prowell 2005, Ostrand et al 2005, Ramon and Jose 2008). Methods based on static analysis of the program’s source code, such as symbolic execution and constraint solving (Ramamoorthy et al 1976), have been limited by the dynamic nature of software and also impose the tester to select the path manually.

The chaining approach (Korel 1990) alleviates the path problem by attempting to find program nodes, on which the target is data-dependent and must be executed in the path to the target. They are explored in a series of chains. However, the drawback is that, no program transformation takes place, paths are not isolated, and thus the fitness function could potentially encounter noise from other paths. Evolutionary chaining hybrids (Ferguson and Korel 1996) have the disadvantage that an entirely new search must be attempted for each chain.
Because of the problems in non-evolutionary approaches, the application of Meta-heuristic search techniques, such as evolutionary algorithms, to search a program’s input domain for test case generation has been a topic of interest for many researchers in recent years.

Several research works applied GA for test case generation. Roper et al (1995) presented an approach, where the population of the Genetic Algorithm is the set of test data (each individual is a test data of the set) and the fitness of an individual corresponds to the coverage achieved in the program under test. The algorithm evolves the population to achieve a desired level of branch coverage.


In Jones et al’s approach (1996), the search is directed by a fitness function that is used to select the best solutions. A low fitness value is given to the candidate solution, if the branch to be executed is not reached. If the branch is reached but not executed, the fitness value measures the difference between the predicate values needed to execute the branch and that of the candidate solution.

Pargas et al (1999) applied Genetic Algorithm (GA) to search for test data that exercises program “targets” (statements or branches in the
current version). A fitness function is used to qualify each solution according to the number of executed predicates with regard to the Control-Dependence Graph (CDG) predicates. The approach assumes that the test data are “closer” to execute a target when they satisfy a higher number of predicates in the CDG path of the node. However, the drawback here is that, the fitness values do not take into account which candidate solution is “closer” in satisfying the predicates. This fact makes the search poorly directed. Since, all the candidate solutions reaching a given predicate get the same fitness value, it makes no difference on how “far” or “close” to solve the predicate.

Bottaci (2001) implemented a multi-population method where each sub population performs a test data search using a fitness function that is based on control dependency information of the original target and a set of dynamically identified sub goals. Sub goals are identified on the basis of branches that were not executed en route to the target.

Paolo Tonella (2004) proposed evolutionary testing of classes. In that work, Genetic Algorithm (GA) was exploited to automatically produce test cases for the unit testing of classes in a generic way. Test cases were described by chromosomes, which include information about the objects to be created, the methods to be invoked and values to be used as inputs. The algorithm proposed by this approach mutates them with the aim of maximizing a given coverage measure.

McMinn (2004) indicated in his work that, about 41% of the conditions in the program were usually covered by random test data generation. By contrast, genetic search covered 60% of the conditions. And genetic search achieved about 85% condition-decision coverage on an average, while the random test-data generator consistently achieved just over 55%.
Xie et al (2004) developed a framework called Rostra to identify the redundant unit tests in testing object oriented software.

Baudry et al (2005) proposed genes and bacteria based test case optimization framework for the .NET environment. This approach has all the operators of GA except the crossover operator. Also, it includes memorization for remembering all the offspring during every generation. It has the drawbacks of losing the best qualities of both the parents due to the absence of crossover operator in solution generation.

Last et al (2006), 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 is an application dependent one. The fuzzification and defuzzification processes consumes a lot of time.

Bruno et al (2007) proposed a new stochastic algorithm called Generalized Extremal Optimization (GEO) for automatic test data generation in path based testing. In their work, an initial assessment of the efficacy of GEO as a path-wise test data generator for path testing is made. A path-wise test data generator is a system that tests software using test adequacy criterion. From the SUT, a control flow graph is constructed, and the list of paths to be covered is extracted from this graph, and GEO guides the search to find test data that covers a specific path by creating new test data from previously generated ones that were evaluated as good candidates.
5.4 PROPOSED TEST CASE OPTIMIZATION FRAMEWORK

In the proposed framework, a novel meta-heuristic search algorithm namely, Hybrid Genetic Algorithm (HGA) is proposed for test case optimization process. The proposed framework, focuses on test case generation and optimization for object oriented applications in which each Class under Test (CUT) is the unit to be tested (Binder, 2000). The approach is particularly 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. The test case generated in the proposed framework is a sequence of method calls with the parameters passed to them.

It should be noted that, in testing classes as units, the following things needs to be done: (i) parameters to invoke the constructor, (ii) some of the methods that change the state of the object under test, or (iii) the method under test. If some of these parameters are objects, they must be created and put into a proper state.

Thus, a test case for 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. For example, consider a class ‘A’ with method ‘p’ and a class ‘B’ with method ‘q’, then, the possible test cases are,

\[ A \ a = \text{new} \ A(); \ B \ b = \text{new} \ B(); \ b.q(20); \ a.p(45, b); \]

Assume that, the SUT is a Binary Search Tree problem, which is provided as a separate class to be tested. The class has two methods namely, ‘Insert’ and ‘Search’. Here the test cases should contain all possible combinations of accessing these two methods with different parameters. But,
this involves an exhaustive enumeration of test cases which is not possible in reality.

Hence, the need is to generate a basic set of test cases and then improve the quality of these test cases during iterations, so that they can satisfy the specifications provided by the customers and at the same time exercises the entire software and reveals as much number of errors as possible.

In the proposed approach, for the given Binary Search Tree class, the test case is given as a sequence of method calls with parameters as, insert(10), search (5), search (2), insert (-1), insert (5), search (10), insert (8).

In this research work, a general test case optimization framework as shown in Figure 5.1 has been formed, for automation of test case optimization problem using evolutionary algorithms like Simple Genetic Algorithm (SGA), Bacteriological Algorithm (BA) and Hybrid Genetic Algorithm (HGA).

![Figure 5.1 Framework of Test Case Optimization](image)

In this research work, the performance of GA, BA and HGA are evaluated based on the fault revealing capability and path coverage of the test cases. Finally, it has been proved that HGA outperforms the other two population based search approaches.
In all these approaches, each chromosome represents a legal solution to the problem and is composed of a string of genes. For the problem of software test case optimization, each chromosome is represented as a stream of methods to be called in each class. When selecting a particular method with a value, it must produce the right result. The sequence of operations are given here as the test cases.

Test cases are provided as,

\{(\text{method}_1(p_1, p_2, \ldots, p_n), \text{method}_2(p_1, p_2, \ldots, p_n), \ldots, \text{method}_n(p_1, p_2, \ldots, p_n) \}

(Where, \(p_1, p_2, \ldots, p_n\) are parameters passed to the methods.)

Example: insert(10), insert(2), search(10), insert(5), search(-1)…

Here insert and search are method names and the parameters are the values passed to them. This sequence of method calls is considered as one test case.

5.4.1 Genetic Algorithm Based Test Case Optimization

Genetic Algorithms were invented by Holland (1975) to mimic some of the processes of natural evolution and selection. In nature, each species needs to be adapted to a complicated and changing environment in order to maximize the likelihood of its survival. Genetic algorithms contain the functions like crossover, mutation, selection and evaluation.

The knowledge that each species gains, is encoded in their chromosomes. When reproduction occurs, this undergoes transformations. Over a period of time, these changes to the chromosomes give rise to species that are more likely to survive, and so have a greater chance of passing their improved characteristics on to future generations (McMinn 2006, Pargas
1999). Hence, genetic algorithm can be exploited to improve the quality of test cases automatically.

The first step in GA is to represent a legal solution to the problem by a string of genes that can take on some value from a specified finite range or alphabet. This string of genes, which represents a solution, is known as a chromosome (Pargas 1999).

Test cases are described by chromosomes, which includes the information about, which object/component to create, which methods to invoke and which values to input. Then an initial population of legal chromosomes is constructed at random. At each generation, the fitness of each chromosome in the population is measured. The fitter chromosomes are then selected to produce offspring for the next generation, which inherit the best characteristics of both the parents. After many generations of selection for the fitter chromosomes, the result is hopefully a population that is substantially fitter than the original (Pargas, 1999).

(a) **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 fitness values of each offspring
   d. Replace worst ranked part of population with offspring
4. Until <terminating condition>
(b) **Framework for genetic algorithm based test case optimization**

The framework shown in Figure 5.2 is based on genetic algorithm. It gets as input the initial set of test cases either through the tester or by the random generation method. These test cases are then evaluated based on mutation score and path coverage. If the optimization criterion is met, then the test case is chosen as the optimal one and is stored in the repository. Otherwise, two best test cases among the initial set of test cases are chosen as parents for the next generation. Then crossover and mutation operations are performed on these parents to generate the offspring. Again the evaluation is done on these newly generated test cases till the optimization criterion is met.

![Figure 5.2 General framework of SGA](image)

(i) **Chromosomal representation:** In population based software testing approach, a test case set is considered as a population and each individual test case in it is considered as a member and the individual variables’ values and method calls are considered as the genes.
(ii) **Test case initialization:** At the beginning there is a population of mutant programs to be killed and a test case pool generated by GA are constructed. The test cases are then randomly combined to build an initial population of test cases.

(iii) **Test case generation - mutation and crossover:** Initial test cases are taken from the test case initialization procedure and are used as the parents for the next generation. Between the two parental test cases crossover and mutation operations are performed. A ‘1-point’ crossover is performed for generating new off spring. After performing these two operations, a set of newly generated test cases are generated.

(iv) **One Point Crossover:** Fragmenting the selected population at some point ‘m’ and recombine the <0..m-1> portion of first member and <m...n> portion of the second member, as well as recombine the <0..m-1> portion of second member and <m..n> portion of the first member is done in crossover operation. It is shown in Figure 5.3.

\[
\begin{align*}
\text{Gene 1:} & \quad (a_0, a_1, \ldots, a_k) \quad (b_0, b_1, \ldots, b_k) \\
& \text{Select } 1 \leq m \leq k \\
& (a_0, a_1, \ldots, a_{m-1}) \quad (a_m, \ldots, a_k) \quad (b_0, b_1, \ldots, b_{m-1}) \quad (b_m, \ldots, b_k) \\
\text{One point Crossover} \\
& (a_0, a_1, \ldots, a_{m-1}) \quad (b_m, \ldots, b_k) \quad (b_0, b_1, \ldots, b_{m-1}) \quad (a_m, \ldots, a_k)
\end{align*}
\]

*Figure 5.3 One point crossover*
(v) **Mutation**: This operation applies the mutation operator to generate offspring. It generates a new test case by slightly altering an ancestor test case. Recursive application of this operation, may give the whole set of possible test cases (TC). This module generates the seed test cases automatically by random permutation of initial methods. Then the subsequent generations are created by mutation operation.

(vi) **Test case evaluation and selection – path coverage% and mutation score**: Fitness evaluation involves defining an objective or fitness function against which each chromosome is tested for suitability in the environment under consideration. As the algorithm proceeds, one could expect the individual fitness of the "best" chromosome to increase, as well as, the total fitness of the population as a whole.

The path coverage and mutation score of each of the generated test cases is calculated to calculate the fitness value. Also, the time taken to execute each of the generated test cases is identified.

- **Path Coverage %**: Path coverage percentage is calculated by means of code instrumentation. Based on path coverage percentage and mutation score, the parent for the next generation is selected.

- **Mutation Score Calculation**: This module is used to calculate the mutation score of each test case. To find the mutation score of each test case generated by GA, the test cases are passed to a mutation testing tool. Either ‘Jester’ / ‘muJava’ is applied for calculating the mutation score of the individuals. The testing tool initially creates a collection of mutants for the SUT. Then the mutation score for each test case is identified by executing them against the mutants. The tool ‘muJava’ will directly
provides the mutation score of each individual. But, if the tester has used ‘Jester’ tool, then the result is an XML file. To handle this situation, an XML parser code is developed as part of this research work, to extract the mutation score of each test case.

(vii) Problems with genetic algorithm: In many problems with sufficient complexity, GAs may have a tendency to converge towards local optima rather than the global optimum of the problem (Baudry et al 2005). Also, operating on dynamic data sets is difficult, as genomes begin to converge early on towards solutions which may no longer be valid for later data.

GAs cannot effectively solve problems in which the only fitness measure is right/wrong, as there is no way to converge on the solution (No hill to climb.) In these cases, a random search may find a solution as quickly as a GA. The implementation and evaluation of the fitness function is an important factor in the speed and efficiency of the algorithm.

5.4.2 Bacteriologic Algorithm Based Test Case Optimization

Bacteriologic Algorithm (BA) is inspired by evolutionary ecology. One individual cannot fit the whole environment and a single perfect test case can not kill all the mutants. As per, Baudry et al (2005), the BA based approach is more adaptive than Simple Genetic Algorithm (SGA). In BA, the individuals called bacteria correspond to atomic units. The algorithm contains the functions such as mutation, fitness evaluation, filtering and memorization. The memorization operation in BA is used to remember best candidates at the end of each generation for backtracking. By using this, BA avoids striking up at local optima.
It aims at applying only the mutation operator to the initial population and the adaptation is based on small changes in the individuals. As the Simple Genetic Algorithm (SGA / GA), 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 and they are mutated to generate a list of solution candidates. Then the best ones are selected to produce next generation. This process stops after a termination criterion such as, reaching the maximum number of generations or the memorized population has attained an optimum fitness value is reached.

The Bacteriologic Algorithm takes as input an initial set of test cases, and it outputs quality test cases. This algorithm is more stable than GA due to its memorization operation. And, the test cases have reached a mutation score of up to 95%. So BA is more adapted to test case optimization than GA.

(a) **Bacteriologic algorithm - introduction**

As shown in Figure 5.4, the bacteriologic algorithm takes as input, an initial set of test cases and outputs a good set of improved test cases. This algorithm evolves incrementally. That is, the algorithm builds the final set incrementally by memorizing test cases that can improve the test case set’s quality. Stopping criteria may be maximum number of generations (provided by the tester) or when the solution set reaches an optimum fitness value or if the set’s fitness value has not changed for a number of generations and so on.
Figure 5.4 General Framework of Bacteriologic Algorithm (BA)

(i) **Chromosomal representation:** As in GA, the test case set is considered as a population and each individual test case in it is considered as a member and the individual variables’ values and method calls are considered as Bacteria.

(ii) **Test case initialization:** As in GA, at the beginning, a population of mutant programs to be killed and an initial test case pool that comprises of seed test cases along with randomly combined test cases is formulated.

(iii) **Test case generation – mutation operation:** This module applies the mutation operator to generate offspring. It generates a new test case by slightly altering an ancestor test case. Recursive application of this operation, may give the whole set of possible test cases (TC).
(iv) **Fitness function:** As in GA, the fitness function takes the mutation score and path coverage of each test case to calculate the fitness value.

(v) **Filtering function:** This function is used to remove useless test cases periodically from the bacteriologic medium to control the consumption of memory space during execution. The criteria to delete the test cases from the memory are size of the test cases, Memorization threshold and mutation score.

(vi) **Memorization function:** This function is used to store all test cases for the future reference. The best test cases of all generations stored in population table. This operation includes the memorization of Generation Number, Test cases, Fitness value (mutation score and path coverage).

After the next iteration, if the test cases (offspring) generated were very poor and the parent of any of the previous generations have higher fitness value, then the algorithm automatically replaces the offspring with the best parent.

(b) **Drawbacks of bacteriologic algorithm**

Even though BA achieves 80% to 98% of mutation score, and 70% to 95% of path coverage, there is a possibility of losing good parent’s properties due to the absence of crossover operator.

5.4.3 **Need for Hybrid Genetic Algorithm Based Approach**

From the literature study, it has been identified that, both simple optimization techniques as well as population based search heuristics have only less consideration on fault revealing capability of the test cases in test case optimization problem. Simple search heuristic such as Genetic Algorithm
(GA) has the problem of striking up at local optimal solution and lack of memorization to remember best individuals for backtracking. Even, the recently proposed Bacteriologic Algorithm (BA) has the problem of losing good parents characteristics by not having crossover operator.

It has also been observed that, the heuristic guided intelligent search techniques have provided solutions to avoid striking up at local optima by searching through the neighborhood nodes in the solution space. Hence, if the best qualities of GA, BA and local search techniques have been combined, then a better algorithm could be formed. So, in the proposed approach, the operators of GA (Crossover, Mutation, Evaluation, and Selection), memorization operation of BA and local search procedure from an intelligent searching technique are combined to form the proposed Hybrid Genetic Algorithm (HGA).

5.4.4 Proposed Hybrid Genetic Algorithm Framework

The Hybrid Genetic Algorithm (HGA) is also called as Memetic algorithm (Natalio and Jim 2005). It combines the operators of GA with Local Search (LS) to conduct a heuristic guided search in the given problem domain (Vincent 2008). They are more efficient than GAs, because of the inclusion of a local search algorithm. In HGA, Genetic Algorithm is used for attaining global optima and local search algorithms are used for reaching local optima. Hence, the final result of HGA will have near global optima (Land 1998). Here the generations are called as memes and they are processed and improved based on the local search algorithm employed by the individuals.

In the proposed approach, the HGA contains all the basic components of Genetic algorithms such as,

- Chromosomal Representation
- Initial Population
- Fitness Evaluation
- Selection
- Crossover and Mutation

Apart from these basic components, it also includes two new improvement heuristics namely ‘LocalOpt’ and ‘RemoveSharp’ in the local search process to attain a near global optimal solution. After the initialization of test cases, these two heuristics have been applied to select the best parents for the next generation. Then the evaluation of test cases is done based on mutation score and path coverage.

If the optimization criterion is met, (for example mutation score above 95%, path coverage above 92% or maximum number of generations etc.), then the solution generation is stopped and the resultant test cases are stored in the test case repository. If not, the selection of best test cases to act as parents for the next generation is done based on the evaluation criteria such as mutation score and path coverage. The offspring are generated by applying crossover and mutation operators on the selected test cases.

5.5 INTERNAL ARCHITECTURE OF HYBRID TESTER

The internal architecture of Hybrid Tester shown in Fig. 5.5 uses Hybrid Genetic Algorithm (HGA) based approach for optimal test case selection. The framework has the following activities:

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 (TCDB)

To do test case optimization, 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.

Figure 5.5 HGA based test case optimization framework

The Hybrid Tester module takes as input, the Software under Test (SUT), and the optimal test sequences/paths generated from the Software under Test (SUT) for ensuring the statement and path coverage based criteria.
The initial set of test cases is generated from the source code 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 path coverage criterion is used as a metric to verify the coverage value of each test case. And, mutation score of each test case is used as the fault revealing criterion of each test case. Based on these values, the next generation of test cases is generated by the Hybrid Tester. At the end of each generation, the Hybrid Tester produces a test report based on the coverage and mutation score analysis.

The generated optimal test cases and the test sequences which are covered by them are stored in the optimal test case and optimal test sequence repository respectively. These test cases are then used as the parents for the next generation and the process continues. Finally, if any one of the termination criterion is met, the Hybrid Tester stops the test case generation process.

5.6 ALGORITHM FOR TEST CASE OPTIMIZATION

In the proposed approach, Hybrid Genetic Algorithm (HGA) has been applied to improve the quality of test cases. The test cases are selected based on the mutation score and path coverage based test adequacy criteria.

5.6.1 Heuristics Used in Test Case Optimization Algorithm

There are two heuristics introduced in the proposed algorithm to guide the local search approach. They are: ‘RemoveSharp’ and ‘LocalOpt’ heuristics.
(a) **RemoveSharp Heuristic**

In this heuristic, Mutation Score (MS) of all the offspring produced by ‘n-point’ crossover and mutation is calculated. The offspring which leaves more number of mutants in survival are deleted from the memory. That is, the test cases with very less mutation score are removed; whereas, the test cases with higher mutation score are memorized.

(b) **LocalOpt Heuristic**

At the end of every generation, by applying the ‘n-point’ crossover and mutation, the offspring which has highest mutation score is selected as local optima. The selected test case is termed as a local optimal solution and is removed and stored in the local optimal solution list. By means of this operation, the algorithm avoids striking up at local optima.

If this solution is not removed, then the algorithm will tend to select the same test case repeatedly, since it has the highest mutation score and the algorithm will strike up at local optima. Once it is removed, and stored in the temporary local optimal solution list, a next local optimal solution is selected from the offspring and is also stored in the local optimal list.

Then, the mutation score and path coverage of the local optimal solutions are compared with that of the parents. If both the parents are weaker, then they both are replaced by the local optimal solutions. Otherwise, if one of the parents is weaker, then it is replaced by the best local optimal solution. If both the parents are better than the local optimal solutions, then they will be retained.
5.6.2 Proposed Hybrid Genetic Algorithm for Test Case Optimization

The proposed Hybrid Genetic Algorithm with improvement heuristics shown in Figure 5.6 has the following steps:

Step 1: Initialize population randomly

Step 2: Apply RemoveSharp heuristic to all test cases in the initial population

Apply LocalOpt heuristic to all test cases in the initial population

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

Apply Crossover and Mutation operations between parents and generate offspring

Apply RemoveSharp heuristic to each offspring

Apply LocalOpt heuristic to each offspring

Step 4: If (Mutation Score(offspring) > 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 specified number of iterations or the specified termination criterion is met.
Figure 5.6 Proposed HGA Algorithm

5.6.3 Pseudo code of HGA Algorithm

Step 1: Initial-Pop=Rand(seed). The seed is a test case given by the tester. Let TC={tc₁,tc₂,...,tcₙ} be the set of test cases. Let MS(tcᵢ) represents the mutation score of test case tcᵢ,(i=1 to n). The test cases are ordered as per their mutation score.

Step 2: (a) Apply the following ‘RemoveSharp’ algorithm to all test cases in the population:

i ← 1; min ← MS(tc₁);
repeat
if(MS(tc_i) < min) then remove(tc_i); 
i \leftarrow i+1; \text{ until } (i > \text{total number of test cases})

(b) Apply the following ‘LocalOpt’ algorithm to all Test cases in the population:
i \leftarrow 1; MS(localopt) = MS(tc_1);
repeat
if(MS(localopt) < MS(tc_i)) then localopt \leftarrow tc_i;
\text{i} \leftarrow i+1;
until \text{i} > \text{total-number of test cases}
if (MS(localopt) > MS (parent1)) then parent1 \leftarrow localopt;
else if(MS(localopt) > 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 ‘RemoveSharp’ and ‘LocalOpt’ by repeating Step 2 for each offspring

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

(i) Chromosomal representation

As in GA, the test case set is considered as a population and each individual test case in it is considered as a member and the individual variables’ values and method calls are considered as memes.
(ii) **Selection**

The proposed Hybrid Genetic Algorithm is designed to use heuristics for improvement of offspring produced by ‘n-point’ crossover and mutation. Initial population is randomly generated. The selection of parents for reproduction is done according to the test adequacy criteria namely mutation score and path coverage.

(iii) **Crossover**

The crossover operator followed in the algorithm, is an ‘n-point’ crossover operator. The total length of the parents is calculated and crossover through ‘n’ points produces various offspring. For each parent selected, a random integer number position in the range [1...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 offspring.

(iv) **Mutation**

The crossover operator takes two individuals as parent, out of which one offspring is composed by combining two sub portions, one from each parents. Unfortunately, these parts usually do not add up, to complete the members of the class to be tested. After the combination of the two sub portions, the redundant methods should be deleted, 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.
(v) **Local search procedure**

The improvement heuristics ‘RemoveSharp’ and ‘LocalOpt’ 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 parent then it is discarded.

- **RemoveSharp Heuristic:** Mutation Score of all the offspring produced by ‘n-point’ crossover and mutation is calculated. The offspring which leaves more number of mutants in survival will be deleted from the memory. That is, the test cases with very less mutation score are removed; whereas, the test cases with higher mutation score are memorized.

- **LocalOpt Heuristic:** At the end of every generation, by applying the ‘n-point’ crossover and mutation, the offspring which has highest mutation score is selected as local optima. The selected test case is termed as a local optimal solution and is removed and stored in the local optimal solution list. By means of this operation, the algorithm avoids striking up at local optima. Once it is removed, and stored in the temporary local optimal solution list, a next local optimal solution is selected from the offspring and is also stored in the local optimal list.

Then, the mutation score of the local optimal solutions are compared with that of the parents. If both the parents are weaker, then they both are replaced by the local optimal solutions. Otherwise, if one of the parents is weaker, then it is
replaced by the best local optimal solution. If both the parents are better than the local optimal solutions, then the parents are retained and the local optimal solutions are removed.

(vi) **Fitness evaluation**

As in GA and BA, the fitness function takes the mutation score and path coverage of each test case to calculate the fitness value.

5.7 **EXPERIMENTATION AND EVALUATION**

5.7.1 **Tested Programs**

To study the relevance and efficiency of the proposed approach, all the three approaches discussed are coded in Java and are applied to find solutions for the problems listed in Table 5.1.

The problems list includes both simple and complex problems. The number of classes available in each of these test problems indicates their level of complexity.

Among the listed problems shown in Table 5.1, some of the problems are developed by students as part of their academic curriculum and are marked as academic. The other problems indicated as industrial problems, are developed as per the requirements given by the industries.
Table 5.1 Tested Programs in C++ / Java

<table>
<thead>
<tr>
<th>Case Study #</th>
<th>Object Oriented Systems – in C++ and Java</th>
<th>Type</th>
<th>#Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Binary Search Tree using Java</td>
<td>Academic</td>
<td>2</td>
</tr>
<tr>
<td>2.</td>
<td>Coffee/COCOA/ Money Lending Machine using C++</td>
<td>Industrial</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>Stack using Java</td>
<td>Academic</td>
<td>5</td>
</tr>
<tr>
<td>4.</td>
<td>Queue using Java</td>
<td>Academic</td>
<td>5</td>
</tr>
<tr>
<td>5.</td>
<td>Library Management System using Java</td>
<td>Industrial</td>
<td>20</td>
</tr>
<tr>
<td>6.</td>
<td>Students Mark Processing System using Java</td>
<td>Academic</td>
<td>12</td>
</tr>
<tr>
<td>7.</td>
<td>Banking Transaction System using Java</td>
<td>Industrial</td>
<td>14</td>
</tr>
<tr>
<td>8.</td>
<td>Shopping Cart using Java</td>
<td>Industrial</td>
<td>12</td>
</tr>
<tr>
<td>9.</td>
<td>File System Manager using C++</td>
<td>Academic</td>
<td>7</td>
</tr>
<tr>
<td>10.</td>
<td>Network Monitor using C++</td>
<td>Industrial</td>
<td>28</td>
</tr>
<tr>
<td>11.</td>
<td>Examination Workflow system using Java</td>
<td>Industrial</td>
<td>35</td>
</tr>
<tr>
<td>12.</td>
<td>Quiz using Java</td>
<td>Academic</td>
<td>17</td>
</tr>
<tr>
<td>13.</td>
<td>Management Information System using Java</td>
<td>Industrial</td>
<td>27</td>
</tr>
<tr>
<td>14.</td>
<td>Stock Maintenance using Java</td>
<td>Academic</td>
<td>8</td>
</tr>
<tr>
<td>15.</td>
<td>Credit Card Validation using Java</td>
<td>Industrial</td>
<td>22</td>
</tr>
<tr>
<td>16.</td>
<td>Linked Lists – Singly, Doubly and Circularly using Java</td>
<td>Academic</td>
<td>14</td>
</tr>
<tr>
<td>17.</td>
<td>Fraudulent Detection in Banking Transaction using Java</td>
<td>Industrial</td>
<td>56</td>
</tr>
</tbody>
</table>
The experimental settings for GA, BA and HGA are shown in Table 5.2 (a). It has been indicated that, the tests have been conducted for a maximum of 200 generations with each population size of 100.

Table 5.2 (a) Experimental Setup – GA, BA and HGA

<table>
<thead>
<tr>
<th>#</th>
<th>Parameter</th>
<th>SGA/GA</th>
<th>BA</th>
<th>HGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( N = ) Population size (Max.)</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2.</td>
<td>( L = ) chromosome length</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3.</td>
<td>( N_{gen} = ) total number of generations (Max)</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>4.</td>
<td>( P_c = ) crossover probability</td>
<td>0.9</td>
<td>No crossover</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>Adaptive Correction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>( P_m = ) mutation probability</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>6.</td>
<td>Selection method</td>
<td>Mutation Score and Coverage Based</td>
<td>Mutation Score and Coverage Based</td>
<td>Mutation Score and Coverage Based</td>
</tr>
<tr>
<td>7.</td>
<td>Crossover method</td>
<td>1-point</td>
<td>-Not applicable-</td>
<td>‘n-point’</td>
</tr>
<tr>
<td>8.</td>
<td>Mutation method</td>
<td>Method and Argument Change</td>
<td>Method and Argument Change</td>
<td>Method and Argument Change</td>
</tr>
<tr>
<td>9.</td>
<td>( MinLT = ) Minimum lifetime (number of generations)</td>
<td>-Nil-</td>
<td>Memorization</td>
<td>Memorization</td>
</tr>
<tr>
<td>10.</td>
<td>Reproduction ratio</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>11.</td>
<td>( MaxLT = ) Maximum lifetime (number of generations)</td>
<td>-Nil-</td>
<td>Complete History</td>
<td>Only the fitter individuals greater than the specified mutation score</td>
</tr>
</tbody>
</table>

The set of mutation operators applied as per Offutt’s principle (2004), for mutation score calculation is listed in Table 5.2(b). Since, it has been proved by Offutt (2004), that the listed operators are sufficient to find
the fault revealing capability of the test cases, the proposed approach, applied those operators only to generate the mutants. This ultimately, reduces the number of mutants and thus leads to the reduction of time needed for executing the test cases against each of the generated mutants.

Table 5.2 (b) Mutation Operators Applied for Mutation Score Calculation

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td>2.</td>
<td>AOI</td>
<td>Arithmetic Operator Insertion</td>
</tr>
<tr>
<td>3.</td>
<td>AOD</td>
<td>Arithmetic Operator Deletion</td>
</tr>
<tr>
<td>4.</td>
<td>ROR</td>
<td>Relational Operator Replacement</td>
</tr>
<tr>
<td>5.</td>
<td>COR</td>
<td>Conditional Operator Replacement</td>
</tr>
<tr>
<td>6.</td>
<td>COI</td>
<td>Conditional Operator Insertion</td>
</tr>
<tr>
<td>7.</td>
<td>COD</td>
<td>Conditional Operator Deletion</td>
</tr>
<tr>
<td>8.</td>
<td>SOR</td>
<td>Shift Operator Replacement</td>
</tr>
<tr>
<td>9.</td>
<td>LOR</td>
<td>Logical Operator Replacement</td>
</tr>
<tr>
<td>10.</td>
<td>LOI</td>
<td>Logical Operator Insertion</td>
</tr>
<tr>
<td>11.</td>
<td>LOD</td>
<td>Logical Operator Deletion</td>
</tr>
<tr>
<td>12.</td>
<td>ASR</td>
<td>Assignment Operator Replacement</td>
</tr>
</tbody>
</table>

5.7.2 Case Study 1 - Performance Evaluation of HGA

For evaluation of the performance of HGA, the case study “Binary Search Tree Algorithm” is taken. It is shown as a Java code in Figure 5.7. It has two methods namely ‘search’ and ‘insert’.
(i) **Chromosomal Representation:** The chromosome of Binary tree construction program is represented as (insert (3), search (1), insert (1), search (0)), Where insert and search are methods and the numbers in the brackets represents the parameter value to be inserted or searched and is passed as arguments to the methods.

(ii) **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 preference. This procedure generates the initial population by randomly selecting the methods and parameters to be passed in those methods.

(iii) **Mutation Score Calculation:** Once the initial population is generated, the mutation score of each individual is calculated using this module. Here, the tool called Jester/ MuJava is used to find the mutation score.

(iv) **Test Case Generation - Mutation and Crossover:** As it was previously indicated, the next generation of individuals is generated by applying ‘n-point’ crossover and mutation. Then the generated population is stored for mutation score calculation.

- **Mutation**
  
  Parent1: insert(3), search(4), insert(0), search(4), search(2)
  
  Parent2: search(0), insert(4), insert(0), search(4), search(4)

- **Offspring or Children**
  1. search(3), search(4), insert(0), search(4), search(2)
  2. insert(3), search(3), insert(0), search(4), search(2)
  3. insert(0), insert(4), insert(0), search(4), search(4)
  4. search(0), insert(4), insert(3), search(4), search(4)
  5. …

- **n-point Crossover**
  
  Parent1: insert(3), search(4), insert(0), search(4), search(2)
  
  Parent2: search(0), insert(4), insert(0), search(4), search(4)
• Offspring or Children:

  Parent 1 with 2:
  1. insert(3), insert(4), insert(0), search(4), search(4)
  2. insert(3), search(4), insert(0), search(4), search(4)
  3. insert(3), search(4), search(3), search(4), search(4)
  4. insert(3), search(4), search(3), search(4), search(4)

  Parent 2 with 1:
  1. search(0), search(4), search(3), search(4), search(2)
  2. search(0), insert(4), search(3), search(4), search(2)
  3. search(0), insert(4), insert(0), search(4), search(2)
  4. search(0), insert(4), insert(0), search(4), search(2) …

(vi) **LocalOpt and RemoveSharp Algorithms:** The procedure which is meant for these two algorithms finds the local best solution from each generation and removes it from the current population. Then it stores it in the local optima store as one of the local optimal solution. The filtration occurs for test cases having mutation score (above 80%) as the local best solution in the current population.

(vii) **Final Test Case Selection:** This module takes the test cases from the pool and filters only the test cases that have the highest mutation score. Here the lower limit of mutation score is set as 97%. So, the test cases with mutation score 97% or above are filtered and stored in the optimal test cases repository for further generations.

**Result of case study 1 – test case optimization using HGA:** The case study discussed above, showed the working of HGA in generating optimal or near optimal solution. From the observations, it is understood that the efficiency of the test cases is improved during the iterations. Also, the mutation score of the best individuals are identified during each iteration by
applying ‘Localopt’ and ‘RemoveSharp’ heuristics to avoid strike up at local optima and to reach a near global optimal solution.

5.7.3 Case Study 2 – Performance Comparison of GA, BA and HGA

Here, a simple “stack” class that has methods like ‘push’ and ‘pop’ is taken for demonstration. In this case study, a comparative evaluation has been made between HGA based test case optimization (the proposed technique), the GA based test optimization and the BA based test optimization. The following areas are used to compare with the existing techniques:

1. Number of test cases
2. Mutation score of test cases in each generation
3. Path Coverage percentage of test cases and
4. Number of generations

(a) Performance of GA based test case optimization

The basic set of test cases is optimized to create better test cases in further generations by applying crossover and mutation operations. At the beginning, there is a population of mutant programs to be killed and an initial set of randomly generated test cases pool. Then, GA is applied to improve the test cases ability in terms of mutation score and path coverage. The Genetic Algorithm is coded in Java and the quality of test cases is improved using the operations in it.

(i) Test Case Construction: A test case is a sequence of method calls and arguments to the calls that exercises the classes under test. For example, the test cases are given as,
Test Case 1: Push(10), pop(), pop()
Test Case 2: pop(), push(5), push(8)

(ii) Test Case Generation - Crossover and Mutation

- 1-Point Crossover: After 1-point crossover done at the second position, the new test cases are formed as follows:
  
  Test Case 11: push(10), push(5), pop()
  Test Case 21: pop(), pop(), push(8)

  This new generation of test cases is added to the offspring list.

- Mutation: Changing a member at gene level and reproducing the remaining genes for the creation of new generation is done using mutation operator.

  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 generated as,

  Child 1 - Test Case 11: pop(), pop(), pop()
  Child 2 - Test Case 21: push(7), pop(), push(8)

  Now this new generation of test cases is also added to the offspring list.

(iv) Test Case Evaluation – Mutation Score Calculation and Code Coverage criterion: The population of test cases generated using Crossover and Mutation are evaluated for their survival. The population which has most favorable features is assigned with higher fitness value for evaluation. In software testing the favorable feature is, revealing more number of errors.
Here, Jester / muJava testing tool is used to create mutants for the SUT. Jester contains JUnit as a part of it, which provides information about the adequacy of generated JUnit test suite. Since, the result of Jester tool is an XML report that contains the mutation score of the test cases; an XML Parser code is developed as part of this research work to extract the mutation score of each test case. The test case with highest mutation score and path coverage criterion is selected.

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%

(v) **Test Case Selection – Filtering Function:** The best individual from the current population is selected and used as the parent for the next iteration. This leads to incremental solution generation. The offspring that has highest fitness value replaces any one of the weaker parent.

After the evaluation is done, Parent2 test case is replaced by Test Case 21. Then during the next iteration, these new test cases are used as parents for the generation of offspring.

(vi) **Test Report Generation:** Mutation score and Path Coverage based test reports are generated. The chart shown in Fig. 5.8 indicates the mutation score of individuals generated during the various generations using GA. From the figure, it has also been understood that, the mutation score of the individuals is not steadily improving through the generations, rather the growth is fluctuating and it usually strikes up at local optima.
Figure 5.8 Generation of Test Cases Vs. Mutation Score using GA

Figure 5.9(a) GA based path coverage - Worst case

Figure 5.9(b) GA based path coverage - Average Case
The path coverage% of test cases shown in Figures 5.9 (a), (b) and (c), indicates the worst case, average case and best case performances of GA. It has been identified that, GA strikes up at local optima and even though the ancestors have high mutation score, one cannot use them due to lack of memorization.

(b) **Performance of test case optimization using BA based optimization**

The BA algorithm is coded in Java; which comprises of five modules to generate the optimal test cases.

(i) **Test Case Construction - Mutation function**: This function generates the seed automatically by random permutation of initial methods. Then, it creates further required generations using 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)

(ii) **Fitness function – Mutation Score and Path Coverage**

**Calculation:** Here mutation score and path coverage are combined to form a fitness function. As in GA, Jester / muJava testing tool is used to create mutants for the application which has to be tested. And, an XML Parser code is written to extract 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%

(iii) **Test Case Selection - Filtering function:** This function is used to remove the test cases, which 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 and fitness values of the test cases and memorization threshold.

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

(iv) **Memorization function:** This function is used to store all the test cases for future reference. Here, two versions of BA namely BA with memorization and BA without memorization are applied to generate the population of test cases. The memorization function is used only in BA (Memorized) to store all the test cases for future reference. The test cases of
all the generations are stored in population repository. The mutation score of the test cases generated using these two approaches have been analyzed, and it is understood that, BA with memorization achieves better mutation score when compared to BA without memorization.

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

(v) Test Report Generation: The test cases of all generations are stored in population table. The path coverage of the test cases generated using BA (Memorized) is shown in Figure 5.10.

![Path Coverage - BA](image)

**Figure 5.10  Path Coverage % of test cases using BA**

The path coverage % of test cases generated using BA are steadily improving, and not showing striking up at local optima. This is due to memorization of best individuals during each iteration for backtracking.

In order to find the performance of BA, the fault revealing capability of the test cases has been investigated. The performance of both, BA with memorization and without memorization is calculated. Their performance is indicated in Figure 5.11.
Figure 5.11  Generation of Test Cases Vs. Mutation Score using BA with memorization function and without memorization function

From the Figure 5.11, it has been identified that, BA without memorization reflects the properties of GA and even it shows worse behavior than GA. This is due to the fact that, BA does not have a crossover operation; it is not possible for it to include the best features of the parents which can be used for next generations.

When, the evaluation is done on the performance of BA with memorization, it has been identified that, it shows a steady state improvement in terms of Mutation score to identify the seeded faults in the SUT due to memorization of individuals in each generation.

(c) Performance of test case optimization using HGA based optimization

(i) Test Case Initialization: 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()

(ii) **Selection:** Initial population is randomly generated. The selection of parents for reproduction is done according to the test adequacy criteria. Among the initial generation of test cases, Test Case 3 and Test Case 4 have been selected as parents based on the test adequacy criteria, for the next generation of offspring.

(iii) **Test Case Generation – Crossover and Mutation:** The crossover operator followed here is an ‘n-point’ crossover operator. ‘n-point’ crossover is applied to generate more number of 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 –

Parent1 with parent2:
Child 1: push(5), pop(), pop()
Child2 : push(5), push(7), push(9)
Child3: push(5), push(9), pop()

Parent2 with Parent1:
Child1: push(23), push(7), pop()
Child2: push(23), pop(), pop()
Child3: push(23), pop(), push(5)

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(), …..

(iv) **Fitness Evaluation:** Fitness evaluation involves the calculation of mutation score and coverage based test adequacy criteria. Then, these values are used as fitness values to evaluate each 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%

(v) **Local Search Procedure:** The improvement heuristics ‘RemoveSharp’ and ‘LocalOpt’ 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 parent then it is discarded.

- **RemoveSharp Heuristic:** For the given case study, this heuristic deletes the test case 3, which has very low mutation score of 45% and code coverage of 37% and other test cases that have below 50% of score.
- **LocalOpt Heuristic:** Here the test case 2 is identified as the local optima and is removed and stored. This operation is necessary, to select the next best test cases from the population, otherwise the algorithm will try to select the same test case repeatedly and strike up at local optima.

**(vi) Selection:** 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.

**(vii) Test Report Generation:** The performance of HGA is evaluated. It indicates an improvement in path coverage and mutation score through generations. In test report generation, the fitness of the test cases is indicated in terms of Mutation Score (MS) and Path Coverage. The test cases and their corresponding mutation score and path coverage % are shown in Figures 5.12 and 5.13 respectively. These two figures a steady state improvement in the quality of test cases in terms of coverage and mutation score based test adequacy criteria.

![Mutation Score - HGA](image.png)

**Figure 5.12** Mutation Score of test cases generated using HGA
Result of case study 2: The second case study indicates the effectiveness of test cases generated using three different evolutionary computation approaches GA, BA and HGA. The aim of this study is to identify the effective algorithm in test case optimization. The results indicated that, when compared to GA and BA, the HGA based test case optimization provides better results in generating effective test cases that have high fault revealing capability.

5.8 PERFORMANCE ANALYSIS

5.8.1 Performance of GA, BA and HGA

A comparative analysis is performed using the results of case studies. The comparisons based on Mutation Score and Path Coverage; show that HGA outperforms GA and BA in generating optimal test cases.

From the results, it has been proved that the population generated by HGA is able to detect more number of seeded faults (mutants) present in the SUT. Table 5.3, shows the mutation score (ms) of test cases based on Simple Genetic Algorithm (SGA) and Hybrid Genetic Algorithm (HGA).
Table 5.3 Mutation Score of SGA and HGA

<table>
<thead>
<tr>
<th>Generation</th>
<th>SGA (ms)</th>
<th>HGA (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>37</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>37</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>49</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>46</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>36</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>68</td>
</tr>
<tr>
<td>10</td>
<td>22</td>
<td>72</td>
</tr>
<tr>
<td>11</td>
<td>60</td>
<td>78</td>
</tr>
<tr>
<td>12</td>
<td>45</td>
<td>79</td>
</tr>
<tr>
<td>13</td>
<td>40</td>
<td>83</td>
</tr>
<tr>
<td>14</td>
<td>78</td>
<td>85</td>
</tr>
<tr>
<td>15</td>
<td>69</td>
<td>97</td>
</tr>
<tr>
<td>16</td>
<td>75</td>
<td>97</td>
</tr>
<tr>
<td>17</td>
<td>79</td>
<td>95</td>
</tr>
<tr>
<td>18</td>
<td>59</td>
<td>99</td>
</tr>
<tr>
<td>19</td>
<td>77</td>
<td>92</td>
</tr>
<tr>
<td>20</td>
<td>67</td>
<td>94</td>
</tr>
<tr>
<td>21</td>
<td>53</td>
<td>95</td>
</tr>
<tr>
<td>22</td>
<td>79</td>
<td>92</td>
</tr>
<tr>
<td>23</td>
<td>51</td>
<td>95</td>
</tr>
<tr>
<td>24</td>
<td>69</td>
<td>94</td>
</tr>
<tr>
<td>25</td>
<td>72</td>
<td>95</td>
</tr>
<tr>
<td>......</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The results that are inferred from the evaluations of BA, BA – Memorized and HGA are provided in Table 5.4.

**Table 5.4  Mutation Score of BA, BA (Memorized) and HGA**

<table>
<thead>
<tr>
<th>Generation</th>
<th>BA (ms)</th>
<th>BA (memorized)</th>
<th>HGA (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>13</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>13</td>
<td>37</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>24</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>24</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>31</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>31</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>31</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>31</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>54</td>
<td>54</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>54</td>
<td>68</td>
</tr>
<tr>
<td>10</td>
<td>19</td>
<td>54</td>
<td>72</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Apart from the discussed case study problems, the other test problems listed in the Table 5.1 are also evaluated using all the three different approaches.

The results of two of the listed academic problems and two of the industrial software are shown in Tables 5.5 and 5.6. The number of test cases is rounded to the nearest hundreds, thousands and ten thousands for evaluation purpose.
### Table 5.5 Evaluation Results of simple academic problems

<table>
<thead>
<tr>
<th>Areas</th>
<th>HGA based Test Case Optimization</th>
<th>GA based Test Case Optimization</th>
<th>BA based Test Case Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Taken</td>
<td><strong>Binary Search Tree</strong></td>
<td><strong>Stack</strong></td>
<td><strong>Binary Search Tree</strong></td>
</tr>
<tr>
<td>Possible Test Cases</td>
<td>200</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>Reduced Test Cases</td>
<td>5</td>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>Mutation Score of Test Cases</td>
<td>98%</td>
<td>99%</td>
<td>79%</td>
</tr>
<tr>
<td>No. of test cases that covered the path</td>
<td>5</td>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>Number of generations</td>
<td>50</td>
<td>50</td>
<td>200</td>
</tr>
<tr>
<td>Time taken for Optimization</td>
<td>0.233432</td>
<td>3.699231</td>
<td>0.1213152</td>
</tr>
<tr>
<td>Memory Space Occupied due to Memorization</td>
<td>Less (only the local optimum)</td>
<td>No</td>
<td>High</td>
</tr>
<tr>
<td>Mutation Score Computation</td>
<td>Mutation score is computed only on mutants that have not been killed in previous generations.</td>
<td>Done for each individual on all the mutants in each generation – Due to lack of memorization</td>
<td>Mutation score is computed only on mutants that have not been killed in previous generations.</td>
</tr>
</tbody>
</table>
### Table 5.6 Evaluation Results of Industrial Strength software

<table>
<thead>
<tr>
<th>Areas</th>
<th>HGA based Test Case Optimization</th>
<th>GA based Test Case Optimization</th>
<th>BA based Test Case Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Credit Card Validation</td>
<td>Fraudulent Detection in Banking Transaction</td>
<td>Credit Card Validation</td>
</tr>
<tr>
<td>Problem Taken</td>
<td>20000</td>
<td>40000</td>
<td>20000</td>
</tr>
<tr>
<td></td>
<td>40000</td>
<td>40000</td>
<td>40000</td>
</tr>
<tr>
<td>Possible Test Cases</td>
<td>200</td>
<td>250</td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td>40000</td>
<td>10000</td>
<td>220</td>
</tr>
<tr>
<td>Reduced Test Cases</td>
<td>200</td>
<td>250</td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td>40000</td>
<td>10000</td>
<td>380</td>
</tr>
<tr>
<td>Mutation Score (MS) of Test Cases – Quality of Test Cases</td>
<td>96%</td>
<td>97%</td>
<td>79%</td>
</tr>
<tr>
<td></td>
<td>97%</td>
<td>70%</td>
<td>95%</td>
</tr>
<tr>
<td>No. of test cases that covered the path</td>
<td>180</td>
<td>330</td>
<td>550</td>
</tr>
<tr>
<td></td>
<td>375</td>
<td>775</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>340</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of generations</td>
<td>50</td>
<td>50</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>200</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Time taken for Optimization (MS)</td>
<td>0.833432</td>
<td>6.699231</td>
<td>0.7213152</td>
</tr>
<tr>
<td>Memory Space Occupied due to Memorization</td>
<td>Less (only for memorizing the local optimum)</td>
<td>No</td>
<td>High (Memorization of set of best bacterium in previous generations)</td>
</tr>
<tr>
<td>Mutation Score Computation</td>
<td>Mutation score is computed only on mutants that have not been killed in previous generations.</td>
<td>Done for each individual on all the mutants in each generation – Due to lack of memorization</td>
<td>Mutation score is computed only on mutants that have not been killed in previous generations.</td>
</tr>
</tbody>
</table>
The evaluation results of academic problems and industrial strength problems indicate that, the performance of HGA is high when compared to GA and BA. Also, the time taken by HGA is lesser than GA but it is slightly higher than BA because of the local search procedure included in it. According to no-free lunch theorem for optimization, (Wolpert and Macready 1997), the quality of the test cases achieved in HGA compromises this excess time consumption. Even though the optimization process takes time, the test cases generated by HGA consume only less time and resources for performing testing activity since they are few, efficient and quality test cases when compared to other approaches.

5.8.2 Comparison Charts – GA Vs. BA Vs. HGA

The comparison chart shown in Figure 5.14 indicates the ability of error detection by the test cases generated using both Simple Genetic Algorithm (SGA) and Hybrid Genetic Algorithm (HGA). From this chart, it has been inferred that, when compared to GA, the proposed HGA generated quality test cases having high fault revealing capability.

From the chart in Figure 5.15, it has been identified that, the test cases generated using Hybrid Genetic Algorithm (HGA) have highest mutation score when compared to the Bacteriologic algorithm (BA).

![Figure 5.14 Comparison of GA and HGA based on Mutation Score](image)
Figure 5.15 Comparison of BA and HGA based on Mutation Score

Figure 5.16 Comparison of BA and HGA based on Path Coverage

Figure 5.17 Comparison of GA and HGA based on Path Coverage
The charts shown in Figures 5.16 and 5.17 indicate that, the HGA based test cases showed high path coverage when compared to SGA and BA based test cases. It has also been understood that, the test cases generated using GA suffers at local optima and it fails to produce a global or near global optimal solution.

Even though BA performs better than GA, it has not produced the highest possible mutation score and coverage due to the lack of crossover operator. For performance comparison, BA (Memorized) algorithm is applied.

5.9 SUMMARY

In this proposed approach, a mathematical model for Test Case Optimization problem has been formulated. Hybrid Genetic Algorithm (HGA), a meta-heuristic search approach has been proposed for test case optimization, which produces near global optimal and linear optimal solutions with rapid convergence. The quality of the test cases is improved from generation to generation using mutation score and path coverage based test adequacy criteria. The proposed approach applies two improvement heuristics namely ‘RemoveSharp’ and ‘LocalOpt’ for guiding the local search procedure. All the three algorithms GA, BA and HGA have been coded in Java and developed as a tool “Hybrid Tester”, which is packed as part of the major tool “IntelligenTester”. The screenshots and sample source code of the tool are given in Appendix 2.

The proposed HGA based test case optimization algorithm has been evaluated for its solution quality by comparing it with other meta-heuristic search approaches such as GA and BA. The experiments are conducted on various test beds ranging from simple to complex. Seed population is randomly generated and further generations are generated using GA, BA and HGA. Path coverage (in percent) of the test cases is verified by means of code
instrumentation. And Mutation score of each test case is calculated using Jester / muJava tool.

From the evaluation results, it has been inferred that, GA produces non linear suboptimal solution and usually strikes up at local optima, BA provides linear and optimal solutions, and HGA produces near global optimal and linear optimal solutions with rapid convergence. Even though there is an overhead involved in the local search procedure, this can be compensated when the quality of the solution is taken into consideration. The number of test cases needed to exercise the SUT is reduced to an amount of up to 80.6% (approx.) based on coverage and mutation score when compared to the other approaches. Hence, comparatively Hybrid Genetic Algorithm produces near optimal solutions than GA and BA. Finally, this work concluded that, for test case optimization, Hybrid Genetic algorithm is better than Genetic and Bacteriologic algorithms.