Chapter 6

Regression Test Suite Prioritization using Genetic Algorithms for System Testing

6.1 Introduction

At present most of the test case prioritization techniques increase the rate of fault detection. If all the test cases are executed within a short time then the increase in rate of fault detection caused by the prioritization may not be worth the effort. However, if a test suite takes days or weeks to run, meeting testing goals earlier can proved to be beneficial. Hence, in this chapter a new time constrained prioritization scheme for regression testing has been proposed with another soft computing technique, Genetic Algorithms (GA). In this proposed prioritization scheme, maximum time allotted for execution and coverage information are considered for all the test cases. It can be observed that the GA have been effectively used in other software engineering and programming language problems such as test generation [Par99], program transformation [Fat04], and software maintenance resource allocation [Ant05], and hence we intend to use GA in designing the proposed time constrained test case prioritization. Experiments have been conducted to study the effectiveness of the proposed prioritization scheme on two case studies and on two industrial projects. An Average Percentage of Faults Detected (APFD) metric and ATEI metric are used to determine the effectiveness of the new test case orderings.

6.2 Related work

Empirical study on test case prioritization techniques reported in [Rot99, Won97] sorts test cases such that the test cases with highest priority, according to
some criterion, are executed first. Test case prioritization can address a wide variety of objectives. For example, in [Rot01] Rothermel et al. prioritized the test cases based on coverage alone. Testers might wish to schedule, test cases in order to achieve code coverage at the fastest rate possible during initial phase of regression testing so as to reach 100% coverage at the earliest or to ensure that the maximum possible coverage is achieved by some pre-determined cut-off point. In the Microsoft Developer Network (MSDN) library, the achievement of adequate coverage without wasting time, is a primary consideration when conducting regression tests [Web01]. Furthermore, several testing standards require adequate branch coverage, making the speedy achievement of coverage, an important aspect of the regression testing process.

In literature, many techniques for regression test case prioritization for general applications and test suite prioritization strategies for web applications have been described. Most of these techniques are code–based, relying on information relating test cases to coverage of code elements. In [Elb02], Elbaum et al. investigated several prioritizing techniques such as total statement (or branch) coverage prioritization and additional statement (or branch) coverage prioritization that can improve the rate of fault detection. Recently, to improve the rate of fault detection, Bo et al. [Bo07] have designed a test case prioritization technique for black box regression testing and Sreedevi et al. [Sre08] have designed a technique for prioritizing the user-session-based test cases for web applications testing. In [Won97], Wong et al. prioritized test cases according to the criterion of increasing cost per additional coverage’. Greedy Algorithms are also used and are implemented in the tool, ATAC [Vla05]. In [Sri02], Srivastava and Thiagarajan studied a prioritization technique that is based on the changes that have been made to the program and focused on the objective function of “impacted block coverage”. Other non-coverage based techniques found in the literature include Fault–Exposing–Potential (FEP)
prioritization [Rot01], history–based test prioritization [Kim02], and the incorporation of varying test costs and fault severities into test case prioritization [Elb01, Elb02]. Paolo et al. [Pao06] have proposed a test case prioritization technique that takes advantage of user knowledge through a machine learning algorithm, Case-Based Ranking (CBR). CBR elicits just relative priority information from the user, in the form of pairwise test case comparisons. Siavash Mirarab and Ladan Tahvildari [Sia08] have prioritized the test cases using Bayesian Network approach.

In [Zhe07] Zheng Li, Mark Harman, and Robert M. Hierons studied five search techniques: two meta–heuristic search techniques (Hill Climbing and Genetic Algorithms), together with three greedy algorithms (Basic Greedy, Additional Greedy and 2–Optimal Greedy) and proved that Genetic Algorithms performed well in test case prioritization. Hyunsook Do, Gregg Rothermel and Alex Kinneer [Hyu03] have designed and performed a controlled experiment examining whether test case prioritization can be effective on Java programs tested under JUnit, and compared the results to those achieved in earlier studies. Their analyses [Hyu04] show that test case prioritization can significantly improve the rate of fault detection of JUnit test suites. Saff and Ernst [Saf03, Saf04a, Saf04b] considered test case prioritization for Java in the context of continuous testing, which used spare CPU resources to continuously run regression tests in the background as the programmer codes. They combined the concepts of test frequency and test case prioritization, and reported that, continuous prioritized testing can reduce the time wasted during development. Garey and Johnson reduced the time constrained test case prioritization problem to the NP-complete zero/one knapsack problem [Gar79], which can often be efficiently approximated with a genetic algorithm (GA) heuristic search technique [Chu98].
Test case prioritization has also been done based on the relevant slices. Recently, Dennis Jeffry and Neelam Gupta [Den06] proposed a prioritization technique based on the coverage requirements present in the relevant slices of the outputs of test cases. These prioritization techniques are based on different sources of information, such as history of recent or frequent errors, test cost and code coverage information but have not considered test suite time. Hence in the proposed prioritization technique, Genetic Algorithm and test suite execution time along with coverage information are considered. By means of genetic algorithm, a test suite can be better prioritized using structural test adequacy criteria than by using a random prioritization for maximization of the rate of fault detection within a time constrained test suite execution environment. The challenges, in time constrained test suite execution is presented in the following section.

6.3 Challenges in time based prioritization

Test prioritization schemes typically create a single reordering of the test suite that can be executed after many subsequent changes to the program under test [Rot04, Rot01]. Test case prioritization techniques reorder the execution of a test suite in an attempt to ensure that defects are revealed earlier in the test execution phase. In case of early termination of testing, a reordered test suite can be more effective at finding faults than one that is not prioritized [Rot01].

For example, suppose that regression test suite $T$ contains six test cases with the initial ordering $\{T_1, T_2, T_3, T_4, T_5, T_6\}$ as described in Figure 6.1(a). A prior knowledge of the faults detected by $T$ in the program $P$ is assumed in this example. The number of faults identified, the execution time and the average faults detected per minute for the test cases $T_1$ to $T_6$ are tabulated in Figure 6.1(b). From the tabulation it can be inferred that the test case $T_1$ can find seven
faults, \( \{f_1, f_2, f_4, f_5, f_6, f_7, f_8\} \) in nine minutes, \( T_2 \) finds one fault, \( f_1 \), in one minute, and \( T_3 \) isolates two faults, \( \{f_1, f_5\} \) in three minutes. Test cases \( T_4, T_5, \) and \( T_6 \) each find three faults in four minutes, \( \{f_2, f_3, f_7\}, \{f_4, f_6, f_8\}, \) and \( \{f_2, f_4, f_6\} \), respectively.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>( f_1 )</th>
<th>( f_2 )</th>
<th>( f_3 )</th>
<th>( f_4 )</th>
<th>( f_5 )</th>
<th>( f_6 )</th>
<th>( f_7 )</th>
<th>( f_8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_3 )</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>( T_4 )</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>( T_5 )</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_6 )</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>No. of Faults</th>
<th>Execution Time (mins)</th>
<th>Avg faults /min</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( T_4 )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( T_5 )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( T_6 )</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

(b)

Figure 6.1 Sample Test cases, Faults identified and their Execution time

Suppose that the time budget for regression testing is twelve minutes. Since it is needed to find as many faults as possible at the earliest, it would be
intuitive to order the test cases by simply considering the number of faults that they can detect. For the test case order $T_1, T_4, T_5, T_6, T_3, T_2$, under a twelve minute time constraint, only the test case $T_1$ would have time to run and can find only a total of seven faults. Since time is the primary concern, it may also seem intuitive to order the test cases with regard to their execution time. The test tuples $TC_1 = \{T_1\}, TC_2 = \{T_2, T_3, T_4, T_5\}, TC_3 = \{T_2, T_1\}$ and $TC_4 = \{T_5, T_4, T_3\}$ are considered for execution and the comparison of prioritizations are presented in Figure 6.2.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Time</th>
<th>APFD</th>
<th>Intelligent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_2$</td>
<td>$T_3$</td>
</tr>
<tr>
<td></td>
<td>$T_3$</td>
<td>$T_4$</td>
<td>$T_4$</td>
</tr>
<tr>
<td></td>
<td>$T_4$</td>
<td>$T_5$</td>
<td>$T_5$</td>
</tr>
<tr>
<td></td>
<td>$T_5$</td>
<td>$T_1$</td>
<td>$T_3$</td>
</tr>
<tr>
<td>Total Faults</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Total Time</td>
<td>9</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 6.2 Comparison of Prioritizations

In the time constrained environment, a time-based prioritization test tuple $TC_2$ can be executed to find eight defects. Another option would be, to consider the time budget and fault information together. To do this, we could order the test cases according to the average percent of faults that can be detected per minute. Under the time constraint, the test tuple $TC_3$ can be executed to find a total of seven faults. If the time budget and the fault information are both considered intelligently, that is, in a way that accounts for overlapping fault detection, the test cases can be better prioritized and thus increase the overall number of faults found in the desired time period. In this example, the test cases would be
intelligently reordered so that the test tuple TC_4 would run, revealing eight errors in 11 minutes which is less when compared to the time taken by TC_2 (12 minutes). Also, it is clear that TC_4 can reveal more defects than TC_1 and TC_3 in the specified testing time. Finally, it is important to note that the first two test cases of TC_2, that is T_2 and T_3, find a total of two faults in four minutes whereas the first test case in TC_4, T_5, detects three defects in the same time period. Therefore, the “intelligent” prioritization, TC_4, is favored over TC_2 because it is able to detect more faults earlier in the execution of the tests. Hence it is proposed, to consider the execution time of test cases for prioritization.

6.4 Overview of genetic algorithm

A genetic algorithm (GA) is a form of evolutionary algorithm often used as a search heuristic for combinatorial optimization problems [Bri02]. Invented by John Holland in the 1960s at the University of Michigan, the original goal of genetic algorithms was to formally study the phenomenon of evolution and adaptation as it occurs in nature [Joh92]. A typical structure for a genetic algorithm is shown in Figure 6.3.

![Genetic loop]

- Choose an initial population
- Calculate the fitness value for each individual
- Reproduction
- Crossover
- Mutation on one or several individuals
- Several stopping criteria: X number of generation, a given fitness value reached

Figure 6.3 The Process of genetic algorithm
A population is formed from a set of “chromosomes” that typically take the form of a string. Usually, the string is a bit string. Each chromosome consists of “genes,” such as bits, and each gene is an instance of a particular “allele” (e.g., 0 or 1), where an allele is a biological term denoting the different possible settings of a trait [Mel98]. The GA processes the population of chromosomes by successively replacing one population with another. With each generation, a fitness based on a fitness function is assigned to each chromosome. If an individual is strong, its genes are more likely to be selected for inclusion in the new population. Selection of individuals for reproduction is implemented by eliminating the low-fitness individuals, and inheritance is implemented by making multiple copies of high-fitness individuals. After pairs are selected, for example $c_k$, $c_l$, the chromosomes are either retained as it is, combined using a crossover operation based on the crossover probability or are mutated by flipping individual elements of the gene in a probabilistic manner based on the mutation probability. Reproduction continues until the number of chromosomes in the original population is reached for the new population [Mel98]. Then the old population is replaced with the new, and the process is repeated. The overall fitness of the population generally improves with each generation, and evolution continues until the individuals converge to an answer, a pre-determined maximum number of generations have been completed, or some other stopping condition such as stagnation is reached [Ste04].

The initial population of individuals is usually generated randomly, although, if prior knowledge of the solution is available, it can be directly encoded into the first population as was done by Hillis [Hil90]. Any environment may be used, examples including a computer simulation, interactions with other individuals, a physical world, or a human’s subjective judgment. Within the determined environment, each individual is tested empirically and assigned a
fitness function, which is generally a single number with a higher number representing higher fitness [Ste04].

Determining an appropriate fitness function, is one of the most difficult tasks in designing a genetic algorithm. It determines how each chromosome will be interpreted, and it should quantify the optimality of a solution in a genetic algorithm in order to allow that particular chromosome to be ranked against all the other chromosomes.

Optimal chromosomes, or at least chromosomes which are more optimal, are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will be even better. Another way of looking at fitness functions is in terms of a fitness landscape, which is a representation of the space of all possible chromosomes along with their fitnesses. An ideal fitness function correlates closely with the algorithm's goal, and yet may be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many times in order to produce a usable result for a non-trivial problem. The chief bottleneck in a genetic algorithm is generally in the fitness function calculation. Once the fitnesses have been calculated for all individuals, the next step is to generate a second generation of chromosomes, which is done using any or all of the genetic operators: selection, crossover, mutation and terminating conditions. The genetic operators are explained in the following subsections.

6.4.1 Selection methods

First, a pair of organisms is selected for breeding. Selection is biased towards elements of the initial generation which have better fitness, though it is usually not so biased that poorer elements have no chance to participate. This
prevents the solution set from converging too early to a local solution in hopes that it will eventually lead to points in the search space with an ultimately higher payoff [Joh99]. Stephanie Forrest states in [Ste04] that while there are many well-defined organism selection methods, roulette wheel selection and tournament selection are the most typical. Both of these methods may also be combined with an elite selection strategy which allows some of the better organisms from the first generation to carry over to the second unaltered. In tournament selection, a specified group of two or more individuals is chosen at random from the population. The single individual in the chosen group with the highest fitness is then selected as the winner of the tournament and goes on to reproduce. This type of selection is especially useful when only a partial order exists among the individuals in the population [Dav02]. In roulette wheel selection, individuals are selected with a probability proportional to their fitness. After fitness values are assigned, they are normalized in relation to the rest of the population by multiplying each by a fixed number. Next, the population is sorted by descending fitness values, and accumulated normalized fitness values are calculated. A random number $r \in [0, 1]$ is then chosen, and the first individual whose accumulated normalized value is greater than or equal to $r$ is selected. While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. Thus, roulette wheel selection is often combined with the elitist strategy in order to guarantee that the best individual in a population is kept.

### 6.4.2 Crossover and mutation methods

Following organism selection, the crossover operation is performed upon the selected chromosomes. Most genetic algorithms will have a single probability of crossover $p_c$, typically between 0.6 and 1.0, which encode the probability that two selected organisms will actually breed [Joh99]. In the general case, a random
number between 0 and 1 is generated, and if it falls under the crossover threshold, the chromosomes are mutated; otherwise, they are propagated into the next generation unchanged. Crossover results in two new child chromosomes by mixing the chromosomes of the parents in some way, typically by simply swapping a portion of the underlying data structure at a crossover point. The children are then placed in the second generation pool. This process is repeated with different parent organisms until there are an appropriate number of candidate solutions in the second generation pool. It is important to note that there are many details left unspecified by the general description of a genetic algorithm. Just as there are many ways to do selection, as pointed out in Section 6.4.1., the crossover operator also has many design options. For example, the crossover operator can pass on both offspring to the new generation, or it can arbitrarily choose only one offspring. The number of crossover points can be restricted to one per pair, or there may be up to $N$ points per pair, where $N$ is less than the number of genes in the smallest individual.

The next step is to mutate the newly created offspring. Typical genetic algorithms have a fixed, very small probability of mutation. Theoretical work on the mutation rate setting gives strong evidence that for many problems, mutation probability of $(1/N)$ is an appropriate choice [Sam94]. Generally, a mutation point along the string is randomly chosen, and the single character at that point is changed. More than one mutation may be applied to a particular individual, and the mutated individuals are then copied into the next generation of the population. The goal of mutation is to restore genetic diversity to the population. A simple example of crossover and mutation can be observed in Figure 6.4.
6.4.3 Terminating conditions

The selection, crossover and mutation processes ultimately result in a second generation pool of chromosomes that is different from the initial generation. Generally the average degree of fitness is increased by this procedure for the second generation pool, since mainly the best organisms from the first generation are selected for breeding. The algorithm is repeated until an organism is produced which gives a solution that is “good enough”. The terminating condition usually consists of either satisfying a problem-specific success predicate or completing a specified number of generations. A success predicate depends on the goal of the algorithm or on the nature of the problem. For example, success may consist of reaching a certain threshold level, or for certain problems, it may be possible to recognize a 100% correct solution to the problem. Although almost all components of a genetic algorithm are unspecified by the general description of a GA, there are three main components of all genetic algorithms, as described by Forrest [Ste04]:

1. Independent sampling is provided by large populations that are initialized randomly.
2. High-fitness individuals are preserved through selection, which biases the sampling process toward regions of high fitness.
3. Crossover combines partial solutions from different strings onto the same string, thus exploiting the parallelism provided by the population of candidate solutions.

While the computational setting of a genetic algorithm is much simplified when compared with the natural world, genetic algorithms are capable of evolving complex and interesting structures. Individuals within a population can be used to represent solutions to problems, visual images, or strategies for computer games; genetic algorithms are appropriate for use in studying and modeling evolution in biological, social, and cognitive systems.

Having described the general description of general GA, we present the proposed new prioritization technique utilizing the GA, in the next section.

### 6.5 Proposed prioritization technique

The proposed prioritization technique is based on both testing time and potential fault detection information to intelligently reorder a test suite with Genetic Algorithm. Our prioritization algorithm reorders the tests in any sequence that maximizes the suite's ability to isolate defects. First the execution time of each test case is recorded. Because time constraint could be very short, test case execution times must be exactly known in order to ensure proper prioritization. The recorded time include only the execution time of the test case and not loading time.

Timing information additionally includes any initialization and shutdown time required by a test. Inclusion of initial time and shutdown time is necessary because these operations can greatly increase the execution time required by the test case. The proposed GA based prioritization algorithm is presented in Figure 6.5. The program $P$ and each test case in a test suite are input to the proposed algorithm, along with the user specified parameters.
Algorithm

Input:
- Program $P$
- Test suite $T$
- Number of tuples to be created per iteration $s$
- Maximum iterations $IT_{max}$
- Percent of total test suite time $PT$
- Crossover Probability $CrossProb$
- Mutation Probability $MutProb$
- Addition Probability $AddProb$
- Deletion Probability $DelProb$
- Test Adequacy Criteria $TAC$
- Program Coverage Information $Covg$

Output:
- Test suite with greatest Fitness

Begin
1. Compute $PT$
2. Obtain maximum execution time of a tuple, $time_{tuple}$ from $PT$
3. Create $s$ test tuples executed in $time_{tuple}$
4. Obtain coverage information of all tuples
5. Determine goodness (Fitness) of all tuples using coverage information.
6. For $IT_{max}$ iterations repeat steps 7 to 14
7. Select two best tuples to be the element next generation
8. If the selected tuples are not fit for next generation then until all $s$ test tuples are selected repeat steps 9 to 14
9. Select a pair of parent tuples using Roulette wheel selection based on probability proportional to $|Fitness|$
10. Merge the pair based on $CrossProb$ to create potentially new pairs
11. If the created potentially new pair are fit for next generation then
12. Mutate each new tuple based on $MutProb$
13. Add mutated tuples to $T$ based on $AddProb$, if they fit for next generation else
14. Delete the mutated tuples based on $DelProb$
15. In the final set, tuple with greatest fitness is determined

End

Figure 6.5 Proposed Prioritization Algorithm
The proposed algorithm performs test case prioritization on test suite $T$, based on a given time constraint $PT$. This algorithm computes $PT$ percent of the total time and stores the value in $time_{tuple}$, the maximum execution time for a test suite. It creates a set of $s$ random test tuples $TS$ that can be executed in $time_{tuple}$ time. Once a set of test tuples is created, coverage information of each test tuple is determined, and it is used to determine the Fitness value of each test tuple. Having computed the Fitness value of each test tuple, two best tuples are selected to be the elements of next generation. The two best test suites are chosen in order to guarantee that the next generation has at least one “good” pair. It is important to carry these highly fit test tuples into next generation because they are most likely very close to exceeding $time_{tuple}$. Any slight change to these test tuples could cause them to require too much execution time, thus invalidating them.

Since GA is trying to identify one particular test tuple, this exclusive selection technique ensures that the best test tuple in current generation survives on to next generation. If the selected tuples are not fit for next generation then pairs of test tuples are identified from current generation through a roulette wheel selection technique based on a probability proportional to $|Fitness|$. The Fitness values are normalized in relation to the rest of the test suite set by multiplying each of the test tuple’s Fitness by a fixed number, so that the sum of all Fitness values equals one. The test tuples are then sorted by descending Fitness values, and accumulated normal Fitness values are calculated. A random number $r \in [0, 1]$ is next generated, and the first individual whose accumulated normalized value is greater than or equal to $r$ is selected. This selection method is repeated until enough test tuples are selected to fill the set for next generation. Candidate test tuples with higher Fitnesses are therefore less likely to be eliminated, but a few with lower fitness have a chance to be used in the test tuple set as well.
Suppose if the selected tuples are not fit for *next generation* then the pairs of test tuples are merged to create two potentially new test suites based on *CrossProb*, a user specified crossover probability. Each test tuple in the pair may then be mutated based on *MutProb*, a user provided mutation probability. A new test case may be added to the test suite based on *AddProb* or deleted based on *DelProb*. The cross over operator exchanges subsequences of the test tuples, and mutation operator only mutates single elements. Test case addition and deletion are needed because no other operator allows for a change in the number of test cases in a test suite. After each of these modifications has been made to the original pair, both test tuples are entered into *next generation*. The infrastructure of the proposed GA based prioritization algorithm is presented in Figure 6.6.

Figure 6.6 Proposed GA Prioritization Infrastructures

The same transformations are applied to all selected pairs of test tuples until *next generation* contains *s* test tuples. In total *IT*<sub>max</sub> sets of *s* test tuples are iteratively created in this fashion. When the final set of *IT*<sub>max</sub> has been created,
the test tuple with the greatest Fitness is determined. This test tuple is guaranteed to be the test suite with the highest Fitness out of all sets of size, s. Computation of coverage information, fitness value, crossover, mutation, addition and deletion of test cases are presented in the following subsections.

6.5.1 Test coverage

Since it is very rare for a tester to know the location of all faults in $P$ prior to testing, the prioritization technique must estimate how likely a test is to find defects, which factors into the function fit. The function fit yields the fitness of the test tuple $T_{S_j}$ based on its potential for fault detection and its time consumption. As it is impossible to reveal a fault without executing the faulty code, the percentage of code covered by a test suite is used to determine the tuple's potential. We consider, two forms of test adequacy criteria TAC: (i) method coverage and (ii) block coverage [Rot04, Vla05]. A method is covered when it has been entered. A basic block, a sequence of instructions without any jumps or jump targets, is covered when it is entered for the first time. Because several high level language source statements can be in the same basic block, it is sensible to keep track of basic blocks rather than individual statements at the time of execution. The proposed genetic algorithm accepts coverage information based on the code covered in an application by an entire test suite. As noted by Kessis et al., this is the form that many tools such as Clover [Kes05], Misurda [Mis05], and Emma [Vla05] produce.

6.5.2 Fitness function

The fitness function, represented by fit, assigns each test tuple, a fitness based on (i) the percentage of code covered in $P$ by that test tuple and (ii) the time at which each test covers its associated code in $P$. It is then appropriate to split fit into two parts such that $fit = F_{primary} + F_{secondary}$. The primary fitness
$F_{primary}$ is calculated by measuring the code coverage ($cc$) of the entire test tuple ($TS$). Because the overall coverage of the test tuple is more important than the order in which the coverage is attained, $F_{primary}$ is weighted by multiplying the percent of code covered by the program coverage information, $Covg$. The selection of $Covg$'s value should be sufficiently large so that when $F_{primary}$ and $F_{secondary} \in [0, 1]$ are added together, $F_{primary}$ dominates the result. For a test tuple the $F_{primary}$ is computed as follows:

$$F_{primary} = cc \times Covg$$  \hspace{1cm} (6.1)$$

The second component $F_{secondary}$ considers the incremental code coverage of the test tuple, giving precedence to test tuples whose earlier tests have greater coverage. $F_{secondary}$ is also calculated in two parts. First part, $F_{st}$ is computed by summing the products of the execution time of each test case in a test suite ($time(T_i)$) and the code coverage $cc$ of the subtuple. For some $TS$, $F_{st}$ is computed as follows:

$$F_{st} = \sum_{i=1}^{[TS]} time(T_i) \times cc(\text{subtuple})$$  \hspace{1cm} (6.2)$$

Second part $F_{so}$ represents the maximum value that $F_{st}$ could take (i.e., the value of $F_{st}$ if $T_1$ covered 100% of the code covered by $T$). For a $TS$, $F_{so}$ is computed as follows:

$$F_{so} = cc \times \sum_{i=1}^{[TS]} time(T_i)$$  \hspace{1cm} (6.3)$$

Finally, $F_{st}$ and $F_{so}$ are used to calculate the secondary fitness $F_{secondary}$ as follows:

$$F_{secondary} = \frac{F_{st}}{F_{so}}$$  \hspace{1cm} (6.4)$$

As an example of a fitness calculation, let the program coverage information $Covg = 100$, $P$ be a program, and $TAC$ be a test adequacy criterion. Suppose $TS=$
(T₁, T₂, T₃) and the execution times are \(\text{time}(T₁) = 5\), \(\text{time}(T₂) = 3\), and \(\text{time}(T₃) = 1\) with the test suite code coverage \(cc = 0.20\). Then,

\[ F_{\text{primary}} = 0.2 \times 100 = 20 \]

\(F_{\text{secondary}}\) next gives preference to test suites that have more code covered early in execution. To calculate \(F_{\text{secondary}}\), the code coverages of the subtuples \(TS₁ = (T₁)\), \(TS₂ = (T₁, T₂)\), and \(TS₃ = (T₁, T₂, T₃)\) must each be measured. For this example that \(cc(TS₁) = 0.05\), \(cc(TS₂) = 0.19\), and, as already known, \(cc(TS₃) = cc(TS) = 0.20\). \(F_{\text{secondary}}\) is computed as follows,

\[
F_{s₁} = (5 \times 0.05) + (3 \times 0.19) + (1 \times 0.20) = 1.02
\]

\[
F_{s₀} = 0.2(5 + 3 + 1) = 1.8
\]

\[
F_{\text{secondary}} = \frac{1.02}{1.8} = 0.567
\]

Adding \(F_{\text{primary}}\) and \(F_{\text{secondary}}\) gives the total \textit{Fitness} value of \(TS\). Therefore, in this example,

\[
\text{Fit} = F_{\text{primary}} + F_{\text{secondary}}
\]

\[
= 20 + 0.567 = 20.567
\]

If a test suite execution time, \(\text{time}(TS)\) is greater than the time budget \(\text{time}_{\text{tuple}}\), \(F_j\) is automatically set to -1. Because, such a test suite violates the execution time constraint, it cannot be a solution and thus receives the worst fitness possible. While a test suite \(TS\) with \textit{Fitness} = -1 could simply not be added to the next generation, populations with individuals that have a fitness of -1 can actually be favorable. Since the “optimal” test suite prioritization likely move unsteadily on the edge of exceeding the designated time budget, any slight change to a \(TS\) with \textit{Fitness} = -1 could create a new valid test suite. Therefore, some \(TS\)'s with \textit{Fitness} = -1 are maintained in the next generation. If the test suite execution time \(\text{time}(TS) \leq \text{time}_{\text{tuple}}\), equations (6.1) to (6.4) are used.
6.5.3 Crossover

Crossover is used to vary test tuples from one test tuple set to the next through recombination. It is unlikely that the new test tuple after recombination may be identical to a particular parent test suite. This pairs of test tuples are selected out of current generation. Two potentially new hybrid test tuples are created by crossover. First, a random number \( r_1 \in [0, 1] \) is generated. If \( r_1 \) is less than the user provided value for CrossProb, the crossover operator is applied. Otherwise, the parent individuals are unchanged and wait for the next step, mutation.

If crossover is to occur, another random number \( r_2 \) from 0 to the number test cases of the smallest individual is selected as the crossover point. The subsequences before and after the crossover point are then exchanged to produce two new offspring, as presented in Figure 6.7. If crossover causes two of the same test cases to be in the same test suite, another random test not in the current tuple is selected from T instead of including the duplicated test case. Although a test case may be run more than once in a test suite with all independent test cases, there is rarely a benefit to executing it again. If the new test suite already includes all tests, no additions are made.
### 6.5.4 Mutation

Mutation provides a way to add variation to a new test tuple. The new test suite is identical to the prior parent tuple except that one or more changes may be made to the new test tuple’s test cases. All test tuples that are selected first, are considered for crossover. Then they are subject to mutation at each test case position with a small user specified mutation probability $MutProb$. If a random number $r_3 \in [0, 1]$ is generated such that $r_3$ is less than $MutProb$ for test case $T_i$, a new test not included in the current test tuple is randomly selected from $T$ to replace $T_i$, as demonstrated for $T_2$ in Figure 6.8. If there are no unused tests in $T$ when $T_8$ is chosen for mutation, the test suite is still mutated. Instead of replacing the test with a random test, the test to be mutated is swapped with the test case that succeeds it and this swapping is presented in Figure 6.9.

#### Figure 6.8  Selected random number is less than the mutation probability

$(0.005 < 0.10)$ so the individual is mutated ($T_2$ by $T_{13}$)

<table>
<thead>
<tr>
<th>Original individual</th>
<th>Mutated individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_{13}$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_3$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$T_4$</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$T_5$</td>
</tr>
<tr>
<td>$T_5$</td>
<td>$T_6$</td>
</tr>
<tr>
<td>$T_6$</td>
<td>$T_7$</td>
</tr>
<tr>
<td>$T_7$</td>
<td>$T_8$</td>
</tr>
<tr>
<td>$T_8$</td>
<td>$T_9$</td>
</tr>
<tr>
<td>$T_9$</td>
<td>$T_{10}$</td>
</tr>
<tr>
<td>$T_{10}$</td>
<td>$T_{11}$</td>
</tr>
<tr>
<td>$T_{11}$</td>
<td>$T_{12}$</td>
</tr>
<tr>
<td>$T_{12}$</td>
<td>$T_{13}$</td>
</tr>
<tr>
<td>$T_{13}$</td>
<td>$T_{14}$</td>
</tr>
<tr>
<td>$T_{14}$</td>
<td>$T_{15}$</td>
</tr>
<tr>
<td>$T_{15}$</td>
<td>$T_{16}$</td>
</tr>
</tbody>
</table>

Random values:

- $R(T_1) = 0.300$
- $R(T_2) = 0.005$
- $R(T_3) = 0.620$
- $R(T_4) = 0.700$
- $R(T_5) = 0.400$
- $R(T_6) = 0.500$
- $R(T_7) = 0.010$
- $R(T_8) = 0.002$

Mutation probability $pM = 0.010$

Complete test suite:
Figure 6.9 In the special case that the test subtuple already contains all tests from the original tests suite, mutation is achieved by swapping the test at the designated point with the test that succeeds it.

6.5.5 Addition and deletion

Test cases can be added to or deleted from the test tuples. The sets of test tuples in *current generation* must be allowed to grow beyond the initial set. Addition and deletion ability permits such growth. Although addition and deletion operations are necessary, they should be performed occasionally so as to not violate the principle of the genetic algorithm. If a random number $r_4 \in [0, 1]$ is generated such that $r_4 < AddProp$, a random test case is removed from the individual. If another random number $r_5 \in [0, 1]$ is generated and $r_5 < DelProb$, a random test case not yet executed in the individual is added to the end of the test sequence.
6.6 Experiments and results

Experiments have been conducted with two different categories of projects to measure the effectiveness of the proposed GA based prioritization technique. In Category-I, case study with two projects are conducted and validated with APFD metric. In Category-II, 12 faulty programs are created by inducing one fault in each, for two industrial projects and Average percentage of the number of test cases executed to identify the faults are computed using the metric ATEI.

6.6.1 Category-I

The primary goal of this empirical study is to identify and evaluate the challenges that are associated with time based regression test suite prioritization. The goals of the experiment are as follows:
1. Analyze trends in the average percentage of faults detected by prioritizations generated using different GA parameter values.
2. Determine if the GA-produced prioritizations, on an average, outperformed a selected set of other prioritizations according to the average percentage of faults detected.
3. Identify the trade-offs between the configuration of the genetic algorithm and the time and space overheads associated with the creation of the prioritized test suite.

6.6.1.1 Experiment design

The proposed GA base prioritization algorithm, is implemented in the Java programming language, and it prioritizes JUnit test suites. An overview of the test prioritization implementation with edges between interacting components is presented in Figure 6.10. The test suite is first transformed into a set of test cases and test case execution times. To begin the proposed GA execution, the test
cases and program information are given as input into the genetic algorithm along with the other nine parameters for the GA, as depicted in Figure 6.10.

As the genetic algorithm executes, coverage information is gathered at most \(|TS|\) times whenever the *Fitness* of test suite TS\(_j\) is calculated. *Fitness* is calculated before any test tuple is added to the next test tuple set in *current generation*. For the fitness function calculations, the program coverage information Cov\(_g\), is set to 100 for all experiments, because this would ensure that \(fit \in [0, 100]\). Emma, an open-source toolkit for reporting Java code coverage, is used to calculate test adequacy. Emma can instrument classes for, method and block coverage. Coverage statistics are aggregated at method, class, package, and all class levels for the application under test, and Emma, like most tools, only reports coverage for the entire test suite.
Coverage calculation is expensive due to the number of times it must be gathered. In order to prevent redundant coverage calculations, memorization is performed. This is especially useful in the calculation of the secondary fitness function $F_{secondary}$, which requires the code coverage information for up to $|TS|$ subtuples of test cases for each $TS_j \in current\ generation$. Coverage information is used in the fitness function to calculate a Fitness value for every $TS_j \in current\ generation$. Based on this value, the GA creates $IT_{max}$ sets of $s$ test tuples. From the last generated test tuple set, the test tuple with the Maximum Fitness is returned. This Maximum Fitness is then used in the new test suite, $T'$.  

### 6.6.1.2 Case study applications

Two industrial case studies such as Progress Report and JDepend are considered for evaluation. In Figure 6.11, reviews of the two case study applications are presented. Progress Report provides functions to perform typical progress evaluation tasks including adding student homework grades, adding lab grades, and calculating curves. JDepend is used to traverse directories of Java class files and generate design quality metrics for Java packages. It allows the user to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively. The test cases of Progress Report differ from those in JDepend in that they are I/O-bound by their frequent interactions with a database. On an average, Progress Report's test cases take longer to run, while JDepend's test cases have very short execution time.

<table>
<thead>
<tr>
<th></th>
<th>Gradebook</th>
<th>JDepend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>Functions</td>
<td>73</td>
<td>305</td>
</tr>
<tr>
<td>Test Cases</td>
<td>28</td>
<td>53</td>
</tr>
<tr>
<td>Test Exec. Time</td>
<td>7.008 sec</td>
<td>5.468 sec</td>
</tr>
</tbody>
</table>

Figure 6.11 Case Study Applications
6.6.1.3 Evaluation metrics

In order to evaluate the effectiveness of the given test case, prior knowledge of the faults within the program under test is assumed. A regression test suite prioritization can be empirically evaluated based on the weighted average of the percentage of faults detected over the life of the test suite, or the APFD. Preference is given to prioritization schemes that produce test suite with high APFD values. APFD is defined as

\[
\text{APFD}(T, P) = 1 - \frac{\sum_{i=1}^{g} \frac{\text{reveal}(i, T)}{ng}}{2n} + \frac{1}{2n}
\]

where, \( T = \) test suite, \( g = \) number of faults in program under test, \( n = \) number of test cases, \( \text{reveal}(i, T) = \) position of the first test in \( T \) that exposes fault \( i \).

For example, if it is assumed that the test suite \( T \) consist of seven test cases \( \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\} \) and then it is assured that the tests detect the five faults \( f_1, f_2, f_3, f_4, f_5 \) in \( P \) as shown in Table 6.1.

Table 6.1 Fault detected by \( T = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\} \)

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Faults</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( T_1 )</td>
</tr>
<tr>
<td>( F_1 )</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>( F_2 )</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>( F_3 )</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>( F_4 )</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>( F_5 )</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Consider the two prioritized test suites \( TS_1 = \{T_3, T_2, T_1, T_6, T_4\} \) and \( TS_2 = \{T_1, T_5, T_2, T_4\} \).

\[
\text{APFD}(TS_1, P) = 1 - \frac{3+1+2+5+2}{5 \times 5} + \frac{1}{2 \times 5} = 0.58 \quad \text{and}
\]

\[
\text{APFD}(TS_2, P) = 1 - \frac{1+5+3+4+3}{4 \times 5} + \frac{1}{2 \times 4} = 0.325
\]
Here \( TS_2 \) is penalized because it fails to find \( f_2 \). According to the APFD metric, \( TS_1 \) with APFD\(_{TS_1, P}\) = 0.58 has a better percentage of fault detection than \( TS_2 \) with APFD\(_{TS_2, P}\) = 0.325 and is therefore more desirable.

### 6.6.1.4 Results

Experiments are run in order to analyze the effectiveness and the efficiency of the parameterized genetic algorithm. For all experiments, the resulting test suites are run on programs that are seeded with faults. Each source file in JDepend and Progress Report is seeded with faults multiple times as determined by a mutation configuration file, which contains value substitutions.

For example, `+` is replaced by `-`, `<` is replaced by `>`, and so on. Rule for all mutations used in this experiment is listed in Table 6.2. 40 errors that could be found by at least one Ti \( \in T \) are randomly selected for each application. 25\%, 50\%, and 75\% of the 40 possible mutations are seeded into each program \( P \), where the larger mutation sets are supersets of the smaller mutation sets.

**Table 6.2 Rules used for mutations**

<table>
<thead>
<tr>
<th>Mutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace</td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>if()</td>
</tr>
<tr>
<td>if()</td>
</tr>
<tr>
<td>if()</td>
</tr>
<tr>
<td>if ()</td>
</tr>
<tr>
<td>==</td>
</tr>
<tr>
<td>!=</td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td>--</td>
</tr>
</tbody>
</table>

The overview of the analysis system is shown in Figure 6.12. To analyze the experiment results, the APFD metric is applied to each prioritized test suite. The test suites are run on programs seeded with faults.
The first experiment compares the proposed GA execution results and overheads (when different GA parameter configurations are used). The proposed GA is run with the parameters described in Table 6.3.

Table 6.3 GA Parameters

<table>
<thead>
<tr>
<th>Genetic Algorithm Parameters</th>
<th>Progress Report, JDepend (25, 60), (50, 30), (75, 15) Progress Report, JDepend (25, 60), (50, 30), (75, 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>0.25, 0.50, 0.75</td>
</tr>
<tr>
<td>Maximum number of generations and population size</td>
<td>0.7</td>
</tr>
<tr>
<td>Percent of total test suite time</td>
<td>0.1</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.02</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>method, block</td>
</tr>
<tr>
<td>Addition probability</td>
<td>100</td>
</tr>
</tbody>
</table>
In order to run all possible configurations, 36 experiments are completed: 18 using Progress Report and 18 using JDepend. Thirty-six identical computers are used, each running one trial with one unique configuration. For example, one computer ran a genetic algorithm on the test suite $T$ of the Progress Report application calculating $IT_{max} = 25$ generations of test suite sets, each of which contained $s = 60$ test suites. In this configuration, the prioritization is created to be run with $PT = 0.25$, requiring solution test suites to execute within 25\% of the total execution time of $T$, and fitness is measured using method coverage. An average APFD value of block coverage and method coverage are presented in Table 6.4.

Table 6.4 APFD values

<table>
<thead>
<tr>
<th>Program</th>
<th>Block</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress Report</td>
<td>0.64943</td>
<td>0.58325</td>
</tr>
<tr>
<td>JDepend</td>
<td>0.65832</td>
<td>0.68437</td>
</tr>
</tbody>
</table>

From Table 6.4 we can observe that, on an average, the prioritizations created with Fitnesses based on block coverage, outperformed those developed with Fitnesses based on method coverage. In Progress Report, uses of block coverage produced APFD values 11.35\% greater than the use of method coverage, and in JDepend, block coverage APFD values increased by 12.65\% over method coverage due to block coverage's finer level of granularity. As the time budget is increased, the APFD values increase as well for both Progress Report and JDepend, although the amount of increase for a JDepend prioritization is less than that of the Progress Report prioritizations. This trend, which is due to the nature of the applications' test cases, can be observed in Figure 6.13(a). The Progress Report test cases that find the most faults take a comparatively longer time to execute than the test cases of JDepend. A few other short tests exist in Progress Report's test suite, but these are not the most crucial test cases in terms of defect detection. Thus, fewer influential Progress Report
test cases can be executed within a shorter time budget of 25%. When $PT$ is increased to 50%, the majority of the test cases that find the most faults are able to be run within the time budget, which greatly increases test suite APFD values. An increase to $PT = 0.75$ allows for the inclusion of the shorter, less useful test cases. Since these new test cases are unlikely to find many new faults, changes to the overall APFD of the test suites are minor. JDepend's test cases all have very short execution times, and many of them cover about the same amount of code. As in Progress Report, the longer running JDepend test cases generally detect more faults than the shorter tests. However, because the execution time difference between JDepend test cases is much smaller than that of Progress Report test cases, we observe a less drastic APFD increase in JDepend's prioritizations as $pT$ grows. This is evident from Figure 6.13(a), especially between $PT = .25$ and $PT = .50$.

![Code coverage: Block vs Method](a)
![User Time Overhead](b)

Figure 6.13 GA APFDs and Time Results

Modification of the number of faults seeded and of $(IT_{max}, s)$ led to APFD values that are nearly constant in terms of block and method coverage in the test prioritizations for Progress Report and JDepend. This provides confidence in the
results generated by the genetic algorithm because, about the same percentage of
defects can be found by any of the prioritizations inspite of how many faults
there are or how the GA created the prioritizations. Just as in Table 6.4 and
Figure 6.13(a), prioritizations based on block coverage slightly outperformed
those using method coverage.

Space costs are insignificant, with the peak memory use of all experiments
being less than 9344 KB. Most experiments ran with peak memory use of
approximately 1344 KB. As is seen in Figure 6.13(b), the number of generations
and the number of test suites per generation greatly influence the time overhead.
For example, using block coverage, the genetic algorithm's prioritization of
Progress Report's test suite executed for 13.8 hours of user time on average when
creating 25 generations of 60 test suites. On the other hand, if 75 generations
with 15 test suites are created, the GA only required 8.3 hours of user time to
execute. Due to memorization, many of the fitness values of test subtest suites
created in later GA iterations are already recorded from earlier iterations. Thus,
the fitness of the subtest suites are need less to be calculated again. Under the
experiments that created 25 generations of 60 test suites, there is likely to be
more genetic diversity. Thus, there are more subsequences that are likely to be
found than when prioritization is performed with 75 generations of 15 test suites.
In this circumstance, Emma must be run many more times, and this increases the
prioritization time overhead. The same trend observed in Figure 6.13(b) occurs
when the system time values for Progress Report and JDepend are compared. For
example, a GA executing Progress Report's test suite with 25 generations of 60
test suites using block coverage requires 13.8 hours of user time and 0.78 hours
of system time. However, a GA running Progress Report's test suite with 75
generations of 15 individuals using block coverage required only 8.3 hours of
user time and 0.44 hours of system time, a vast improvement over the (25, 60)
configuration. Time aware prioritization of JDepend test suites consumed 18.0
hours of user time and 2.1 hours of system time when using the (25, 60) configuration but only 12.5 hours of user time and 1.38 hours of system time using (75, 15). A GA prioritizing the test suite of JDepend requires a longer execution time than a GA prioritizing the Progress Report test suite due to JDepend's larger test suite. Since there are more possible subtest suites that can be generated, on average, the fitness function had to be calculated more times.

As the percentage of total test suite execution time is increased for both Progress Report and JDepend, the number of fitness function calculations also increased due to the fact that more test cases could be included in the prioritizations. Since the fitness function demonstrates itself to be the main bottleneck of the technique, when the genetic algorithm needs to run the fitness function calculator less frequently, less time is required overall to reach a result. This confirms the trend seen in Figure 6.13(b) as well. It is also noted that no significant difference is observed between the time overheads of test suite prioritization using block versus method coverage.

Results indicate that the APFD values for Progress Report are similar irrespective of the value that is used for \((IT_{max}, s)\). However, Figure 6.13(b) reveals that a change in \((IT_{max}, s)\) has a significant impact on the time overhead of time-aware test suite prioritization. It is also clear from Figures 6.13(a) and (b) that on average block coverage outperformed method coverage in relation to APFD while not increasing the time overhead of test suite prioritization. Based on our empirical data, a configuration of the proposed GA Prioritize that uses \((IT_{max}, s) = (75, 15)\) and \(tC = block\) would yield the best results in the shortest amount of time. Even though the time required to perform test suite prioritization is greater than the execution time of the test suite itself, a given prioritization can be reused each time a software application is changed. Furthermore, the experiment results clearly indicate that the prioritized tests will detect more faults
earlier than a non-prioritized test suite that is executed over the same extended time period.

### 6.6.2 Category II

The effectiveness of the proposed prioritization technique is validated based on the number of test cases executed. For validation purpose the same set of projects (VB project-Project-1 and PHP project-Project-2) each with 20 requirements and 50 test cases, used in section 2.4.2 of chapter 2 is considered. The programs are thoroughly tested and 12 faulty programs are created (6 faulty programs from each project) by seeding one fault in each, invariant of the severity. On the entire faulty programs, the test cases, prioritized based on the proposed GA technique are run and the total numbers of test cases run to find the faults are noted. Then the test cases are executed in 20 different random orders and the total numbers of test cases run to find faults are detected. The mean value of these 20 different results is computed for each of the 12 faulty programs. The results of fault detection in both the cases are compared to strengthen the effectiveness of the proposed prioritization technique. Test cases for the six faulty programs of Project-1 and six faulty programs of project-2 are executed both in proposed GA technique and in random order.

During execution of test cases of all faulty programs of Project-1 in the proposed prioritized order, for the first faulty program, the fault is detected after running 11 test cases. For second, third, fourth, fifth and sixth faulty programs 9, 8, 7, 6 and 3 test cases are executed respectively to detect the faults. TTEI and ATEI are computed with Equations (2.8) and (2.9):

\[
\text{TTEI}_{\text{Prioritized}} = 11 + 9 + 8 + 7 + 6 + 3 = 44
\]

\[
\text{ATEI}_{\text{Prioritized}} = \frac{44}{300} = 0.147
\]
During execution of test cases of all faulty programs of *Project-1* in Random order, for the first faulty program, the fault is detected after running 31 test cases. For second, third, fourth, fifth and sixth faulty programs 32, 35, 36, 25 and 29 test cases are executed respectively to detect the faults. TTEI and ATEI are computed:

\[
\text{TTEI}_{\text{Random}} = 31 + 32 + 35 + 36 + 25 + 29 = 188
\]
\[
\text{ATEI}_{\text{Random}} = \frac{188}{300} = 0.627
\]

Similarly, for the faulty programs of *Project-2*, ATEI_{Prioritized} and ATEI_{Random} are computed. The values obtained for both the projects are presented in Table 6.5.

Table 6.5 ATEI of *Project-1* and *Project-2* for TCP based on genetic algorithm

<table>
<thead>
<tr>
<th>Project No.</th>
<th>ATEI_{Prioritized}</th>
<th>ATEI_{Random}</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Project-1</em></td>
<td>0.147</td>
<td>0.627</td>
</tr>
<tr>
<td><em>Project-2</em></td>
<td>0.163</td>
<td>0.734</td>
</tr>
</tbody>
</table>

On comparing ATEI_{Prioritized} and ATEI_{Random} of *Project-1*, to detect all the induced faults, 14.7% of test cases are run in ATEI_{Prioritized} and 62.7% of test cases in ATEI_{Random}. Similarly to detect all the induced faults in *Project-2*, 16.3% of test cases are run in ATEI_{Prioritized} and 73.4% of the test cases are run in ATEI_{Random}. So the number of test cases executed to find all the faults is less in case of the proposed prioritized technique. Also lower the value of ATEI, the better is the prioritization technique.

Comparisons between the proposed techniques based on ATEI values are presented in the following subsection.
6.6.2.1 Comparison of proposed prioritization techniques

The comparison of the ATEI\textsubscript{Prioritized} values obtained by the five different prioritization techniques developed in this thesis is presented in Table 6.6. It is evident from Table 6.6 that, the ATEI\textsubscript{Prioritized} values of the proposed prioritization techniques are less than ATEI\textsubscript{Random} and are in reducing order.

Table 6.6 Comparison of ATEI for the proposed techniques

<table>
<thead>
<tr>
<th>Proposed prioritization techniques</th>
<th>Projects under test</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Projects -1</td>
<td>Project -2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ATEI\textsubscript{Prioritized}</td>
<td>ATEI\textsubscript{Random}</td>
<td>ATEI\textsubscript{Prioritized}</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>23%</td>
<td>54%</td>
<td>24%</td>
</tr>
<tr>
<td>With factor values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 3</td>
<td>18%</td>
<td>59.2%</td>
<td>19%</td>
</tr>
<tr>
<td>With factor weight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 4</td>
<td>16%</td>
<td>61.2%</td>
<td>17.1%</td>
</tr>
<tr>
<td>With test cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 5</td>
<td>15%</td>
<td>62.3%</td>
<td>16.2%</td>
</tr>
<tr>
<td>Fuzzy Based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 6</td>
<td>14.7%</td>
<td>62.7%</td>
<td>16.3%</td>
</tr>
<tr>
<td>GA Based</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following observations are made when comparing the ATEI\textsubscript{Prioritized} results of the Project-1 presented in Table 6.6:

- The ATEI\textsubscript{Prioritized} value is 23\% for the factor value based technique proposed in chapter 2, and it is only 18\% for the factor weight based technique proposed in chapter 3. So, the factor weight based technique (chapter 3) reduces the ATEI\textsubscript{Prioritized} by 5\% when compared to the factor value based technique (chapter 2)
• The test cost based technique proposed in chapter 4 reduces the ATEI \textsuperscript{Prioritized} by 2% when compared with the factor weight based technique proposed in chapter 3 and by 7% when compared to the factor value based technique proposed in chapter 2.

• The fuzzy based technique proposed in chapter 5 reduces the ATEI \textsuperscript{Prioritized} by 1% when compared to the test cost based technique proposed in chapter 4, by 3% when compared to the factor weight based technique proposed in chapter 3 and by 8% when compared to the factor value based technique proposed in chapter 2.

• The GA based technique proposed in chapter 6 reduces the ATEI \textsuperscript{Prioritized} by less than 1% when compared to the fuzzy based technique proposed in chapter 5, by 1% when compared to the test cost based technique proposed in chapter 4, by 3% when compared with the factor weight based technique proposed in chapter 3 and by 8% when compared to the factor value based proposed in chapter 2.

The ATEI \textsuperscript{Prioritized} and ATEI \textsuperscript{Random} values of Project-1 for all the five proposed techniques are graphically presented in Figure 6.14. The following observation is made from Figure 6.14:

• The ATEI value for GA based technique is less when compared to all other techniques. Since GA based technique iteratively analyzes the Fitness values of test cases for their ordering, it is able to sort out the test cases in an effective manner.

• The ATEI \textsuperscript{Prioritized} and ATEI \textsuperscript{Random} values of Project-1 and Project-2 for all the five proposed techniques are graphically displayed in Figure 6.15.

• The following observations are made from Figure 6.15:
The ATEI values for all the proposed prioritization techniques are better than the Random technique.

For Project-2, ATEI value of Fuzzy based technique is better when compared to other techniques.

From the observations made from Figure 6.14 and Figure 6.15 it is evident that the prioritizations based on soft computing techniques are better than the other techniques.

Figure 6.14 Comparison of ATEI of Project-1 for the proposed techniques

Figure 6.15 Comparison of ATEI of Project-1 and Project-2 for the proposed techniques
In general, all the techniques proposed in this thesis are, good to better based, on their goal than the techniques reported in the literature and hence can be adopted for practical implementation of efficient software testing.

6.7 Conclusion

In this chapter, a time constraint coverage based prioritization scheme with genetic algorithm for regression testing is proposed. Experiments have been conducted to measure the effectiveness of the proposed scheme on two case studies. From the experiments we can conclude that this method of prioritization has the potential to provide an effective ordering of test cases for applications. Increased granularity in the fitness function led to more effective test case prioritizations as well. An APFD metric and ATEI metric are used to determine the effectiveness of the proposed prioritization scheme. Also in this chapter the challenges associated with time-aware prioritization are identified and evaluated.

In the following chapter, the discussion of the various prioritization techniques, proposed in this research work and suggestions for further research are presented.