Chapter 3

TEST CASE PRIORITIZATION
Chapter 3

TEST CASE PRIORITIZATION

To improve the effectiveness of certain performance goals, test case prioritization techniques are used. These techniques schedule the test cases in a particular order for execution so as to increase the efficacy in meeting the performance goals. For every change in the program, it is considered inefficient to re-execute each and every test case. The limited resources force to choose an effective prioritization technique, which makes an ordering of the test cases, so that the most suitable test case will be executed first. Test case prioritization techniques arrange the test cases within a test suite in such a way that the most important test case is executed first. This process enhances the effectiveness of testing. Executing the regression test case for a fixed time is all about time aware test case prioritization. In this chapter, we proposed three different test case prioritization technique using metaheuristic techniques. Our first algorithm is based on genetic algorithm, which is ordered according to severity, time of execution and code coverage. This algorithm, during time constraint execution, has been shown to have detected maximum number faults while including the severe test cases. Regression test case prioritization using chemical reaction optimization (CRO) for the object-oriented program is proposed secondly. The effectiveness of the test case ordering is measured using APFD (Average percentage of Faults Detected). Experiments on three object oriented subject programs involving three different techniques are performed to judge the said approach. The empirical results indicate that the algorithm implemented using CRO gives higher APFD value than the other two techniques. Regression test case prioritization using Ant colony optimization for the object-oriented program is presented lastly. The empirical results indicate that the algorithm implemented using ant colony optimization gives higher APFD value than the random techniques. These techniques may be used by the quality assurance team for
prioritizing test case as its space and time complexity is less as compared to random ordering.

3.1 Introduction

High-quality software system cannot be completed without rigorously developing testing methodologies [117, 118]. With the rise in size and complexity of recent software system product, the importance of regression testing is growing faster. In the field of software system development, considering the character of software system quality and growing stress on software system products, test suit prioritization is of much importance and relevant to business.

The whole testing method consumes 30-50% of the development cost. In the given resource constraints, the re execution of a set of test cases is impractical. It’s tough for the tester to create the product 100 percent bugs free in limited given time [119]. Additionally, as three hundred and sixty-five days functionality of a product is commonly used, the tester is required to target creating such functionalities bug-free [120, 121]. To undertake this, in given resource constraint the application of all sets of test cases is impractical. So the partial test cases are selected which may find the most number of faults within the given version of the software system [122]. Such choice, which may amount to minimization of test cases but might not always be effective in finishing the method of finding the faults. It so needs that the test cases ought to be prioritized by following code coverage and execution time. Test case prioritization [123, 22, 12] is one in all these approaches that orders test cases so the test cases with the highest priority, consistent with some fitness metric, are executed first.

Rothermel et al. [22] define the test case prioritization problem and describe several issues relevant to its solution. The test case prioritization problem is defined (by Rothermel et al.) as follows:
The Test Case Prioritization (TCP) Problem

*Given:* $T$, a test suite; $PT$, the set of permutations of $T$; $f$, a function from $PT$ to the real numbers.

*Problem:* Find $T' \in PT$ such that $(\exists T' \in PT \ (T'' \neq T')) \ (f(T')) \geq (f(T'))$

*Here,* $PT$ represents the set of all possible prioritization (orderings) of $T$ and $f$ is a function that, applied to any such ordering, yields an award value for that ordering.

Meta-heuristic search techniques [24] are high-level frameworks that utilize the automatic discovery of heuristics so as to search out solutions to combinatorial problems at an inexpensive procedure price. Evolutionary algorithms, of which Genetic Algorithms are a subclass, are a form of meta-heuristic search that employs a Darwinian Evolutionary metaphor to guide the search by a process of “survival of the fittest”. In the case of scheduling problems, of which regression test case prioritization is an example, the application of Genetic Algorithms has been shown to be effective [125]. As such, an empirical study of the effectiveness of those and connected meta-heuristic techniques is timely. As a by-product of such a study, it’s doable to realize an insight into the character of the search space denoted by test case prioritization and to review the fitness metrics used to guide the search.

This chapter focuses on test case prioritization techniques for code coverage as well as execution time. With totally different objective functions, techniques can have totally different complexity and search-space characteristics. Given a function $f$ that assesses the rate of accomplishment of code coverage, an economical answer to the test case prioritization problem would provide an economical answer to the knapsack problem that is understood to be NP-hard [18]. Thus, prioritization techniques for code coverage are essentially heuristic [22].

The CRO includes a smart looking ability that shows glorious operation in 2 necessary options of improvement meta-heuristics: intensification and diversification. It conjointly enjoys the benefits of GA by using crossover operator and mutation that are typically utilized in GA [125]. The CRO, by outperforming several existing evolutionary algorithms with success resolved several issues in recent years. CRO has been applied successfully to the quadratic assignment problem [126], resource-constrained project
programming problem [127], the stock portfolio choice problem [128], artificial neural network training [129], and network coding optimization [130] to several alternative problems.

### 3.2 Related Work

In [22,123,131] Rothermel et al. formally outlined the test suite prioritization drawback and by experiments investigated six prioritization techniques. Four of the techniques were supported the coverage of either statements or branches for a program and 2 of the techniques were supported the calculable ability to reveal faults. In [132], Srivastava and Thiagarajan studied a prioritization technique supported the changes that are created by the program. Their technique orders the given test cases to maximally cover the affected elements of the program in order that defects are likely to be found quickly and inexpensively.

Panigrahi C, Mall R [133, 134] proposed S-RTP and H-RTP which determine the affected nodes in ESDG model based on an analysis of control and data dependencies as well as dependencies arising from object relations and then prioritizes regression test cases based on the number of affected nodes exercised by a test case.

The use of greedy algorithms (total and additional) for regression, TCP has been widely studied in the literature [22]. However, results obtained from empirical studies carried out by Rothermel et al.[13] indicates that the greedy strategies may not always produce the optimal ordering of test cases. To prioritize regression test cases, Li et al. [94] further proposed other greedy strategies such as the 2-optimal strategy and two meta-heuristic search strategies (Hill-climbing and Genetic algorithm).

Jeffrey and Gupta [135] proposed an approach to prioritize regression test cases based on coverage of a relevant slice of the output of a test case. They defined a relevant slice as the set of statements that influence or can influence the output of a program when running on a regression test case [135]. The main aim of their prioritization technique was to achieve higher rates of fault detection.
Mohapatra, S.K., Prasad, S [136] proposed a genetic algorithm based TCP where code coverage and severity of the test case is taken into consideration. Smith A, Geiger J, Kapfhammer GM, Soffa ML [34] use call tree for TCP.

In the literature on test case prioritization, the researchers are largely found to be emphasizing on prioritization technique either based on code coverage [89, 102, 123, 137] or on fault exposing the potential of the test case [138, 139]. Rothermel et al. [123, 125] experimentally investigate six test case prioritization techniques, four of which are based on code coverage and other two are based on fault exposing potential. Their results show that prioritize test suit detect more faults. Li et al. [89] experimentally comparing five algorithms, conclude that in the detection of a fault in the program it is the search space, not the size of the test suite, which is important. The use of the genetic algorithm and code coverage information as fitness criteria Walcott et al. [102], proves that where time is a major constraint, the genetic algorithm gives the promising result. In literature, one also finds certain other prioritization technique where in used either ant colony optimization (ACO) [37] or Particle swarm optimization (PSO) [139, 140]. Mirarab et al. [141] use Bayesian network (BN) approach in prioritization technique. The experiment that is set up using object oriented program leads to the conclusion that soft computing approach yields better results within specified time. According to Sanchez [141], test case prioritization is useful to reduce the quality assurance cost while minimizing the fault detection effort.

### 3.3 Test Case Prioritization Using Genetic Algorithm

In the proposed technique we use the Genetic algorithm. This technique is multi-objective optimization techniques which use execution time of the test suit, code coverage and the severity of the test case as the ordering criteria. Using these above criteria it reorders the test cases intelligently. The process of ordering is represented in Fig. 3.1. The objective of the algorithm is to generate test cases sequences which maximize the test suite’s ability in terms of code coverage and execution time.

According to customer requirement the test case which tests important functionalities of the software are identified as severe. The severe test case must have to be taken in the test suit for retesting. Once the severe test case is identified and included in the prioritized test suite, the remaining test cases should be generated by our GA-based
algorithm. Let’s take an illustrative example to understand it better. We have a pool of 120 test cases and test cases < T3, T19, T62, T67, T88> are identified as severe. The rest 115 test cases in the test suite are optimally reordered by our GA-based algorithm. It is represented in figure 3.1.

![Diagram of test cases](image)

**Figure 3.1: The TCP process using GA**

As timely delivery of the SUT is impartation, it makes the execution time the major criteria for ordering. Our algorithm first considers execution time of the test cases and secondly it considers code coverage of the test case. The inputs to the algorithm are

- Test suite – T
- Number of severe test case - S
- Stopping criteria – \( S_{\text{max}} \)
- Crossover probability - \( C_p \)
- Mutation probability - \( M_p \)
- Initial Population P
Algorithm 1 \textit{GA-pri}

\textit{Input:} Program with initial population $P$

\textit{Output:} Prioritized test cases

1. $a \leftarrow 1$
2. Loop
3. $P_a \leftarrow \text{null}$;
4. Loop
5. $P_a \leftarrow P_a \cup t$ \hspace{1em} where $t$ is test case selected from $T$ randomly
6. while $|P_a| = n$ \hspace{1em} $\{n=T-S\}$
7. $a \leftarrow a+1$
8. while $a = n$
9. $e \leftarrow 1$
10. Loop
11. $b \leftarrow 1$
12. Loop
13. $F_b \leftarrow \text{CalculateFitness}(P_i)$
14. $b \leftarrow b+1$
15. while $b \leq n$
16. $b \leftarrow 1$
17. Loop
18. $P_b \leftarrow \text{ChooseParent}(P_{\text{random}})$
19. $P_{b+1} \leftarrow \text{ChooseParent}(P_{\text{random}})$
20. $C_1, C_2 \leftarrow \text{Crossover}(C_p, P_b, P_{b+1})$
21. $C_1 \leftarrow \text{Mutation}(M_p, C_1)$
22. $C_2 \leftarrow \text{Mutation}(M_p, C_2)$
23. while $b \leq n$
24. $e \leftarrow e+1$
25. while $e \leq S_{\text{max}}$
26. $\delta_{\text{min}} \leftarrow \text{ChooseTuplewithminFitness}(P,F)$
27. return \{Set of severe test case\} $\cup \delta_{\text{min}}$
3.3.1 Fitness Function

The fitness is calculated by taking the code coverage of test case and their time of execution. Two type of fitness constitutes the cover all fitness of the chromosome. Our first fitness component which we call primary fitness (F<sub>p</sub>), is based on measurement of the test adequacy of the final test suite. Primary fitness is computed by taking into account the code coverage of the entire chromosome.

\[ F_p = \text{Code Coverage} \left( T_{Si} \right) \times W \quad i \leq n \]  

(3.1)

Where \( T_{Si} \) is the code coverage of test suite, \( n \) is no of the initial population. Wait \( w \) is taken sufficiently large to make primary fitness dominant in the fitness value. The second component, which we call secondary fitness, is calculated by first considering individual test case execution time and their code coverage. For a single population, the \( F_{cost} \) will be

\[ F_{cost} = \sum_{i=1}^{n} \text{Time(TSi)} \times \text{Code Coverage(TSi)} \quad n \leq k \]  

(3.2)

Where \( k \)=Test suit size-No of identified severe test case, \( n \)=no of the initial population.

After calculating \( F_{cost} \) for all the test cases, we calculate the possible maximum cost \( F_{max} \) by considering the highest coverage of one of the test case in test suit and individual test case execution time.

\[ F_{max} = \text{Max(Code Coverage(TSi))} \times \sum_{i=1}^{k} \text{Time(TSi)} \quad i \leq k \]  

(3.3)

Then the secondary fitness

\[ F_S = \frac{F_{cost}}{F_{max}} \quad j \leq n \]  

(3.4)

It is possible that any two test cases may cover same or part of the same statements. Which make it impossible, to sum up, the coverage of individual test case. That is why secondary fitness is introduced. The secondary fitness considers incremental code coverage of test cases. To understand incremental coverage better let’s take an example, if a chromosome of size 3 is considered with test case <T9, T21, T33> and their execution time are <5, 3, 1> seconds. Let first <T9> selected, in the second iteration T21 is added to it <T9, T21>, and similarly in next iteration T33 is added <T9, T21, T33>. There combine coverage is 0.2%, 0.4%, and 0.5% coverage of the program. Using equation 3.1 the primary
fitness $F_p = 0.5 \times 100 = 50$. TCP is all about early fault detection, the incremental coverage is considered. The secondary fitness is calculated by calculation $F_{cost}$ and $F_{max}$.

$$F_{cost} = (5 \times 0.2) + (3 \times 0.4) + (1 \times 0.5) = 2.7$$

$F_{cost}$ is subject to comparison with its possible highest value possible, which is called the optimal secondary fitness; $F_s$. Optimal secondary fitness would be the fitness of a prioritization whose first test covers all code for that test suite. In our illustration for example T9 covered 100% of all statements covered by T then $F_{max} = 0.5 \times (5+3+1) = 4.5$. Now secondary fitness using equation 3.4

Fs = $F_{cost} \div F_{max}$

= $2.7 \div 4.5$

= 0.60

The fitness of a chromosome is a summation of primary fitness and secondary fitness. For our example the fitness is

$$F = F_p + F_s$$

= $50 + 0.60$

= 50.60

Incremental code coverage after addition of new test case is presented in the following example.

*Example*

Coverage:

| {T1} | =.32 |
| {T1,T2} | =.40 |
| {T1,T2,T3} | =.41 |
| {T1,T2,T3,T4} | =.55 |
| {T1,T2,T3,T4,T5} | =.60 |

Primary Fitness = 60 * 100 = 60

Secondary Fitness = $5 \times .32 + 2 \times .40 + 1 \times .41 + 3 \times .55 + 4 \times .60 / 60 * (5+2+1+3+4)$

= $6.86 / 9 = .76$

Total fitness = 60 + .76 = 60.76
3.3.2 The Initial Population

Each individual population size will be $k$ where

$k =$ Test suit size – No of identified severe test case. The total no of the population we have taken is 100 for our experiment but it can be taken more or less. Always a large population doesn't give the optimal result as it is for a small population.

3.3.3 Selection

We use rank selection to select the chromosome to go to the next epoch. Elitism is used as test show that best population are selected.

3.3.4 Crossover

After the chromosome is selected we applied single point crossover with crossover probability of 0.6 to generate new child from the selected parent.

![Figure 3.2. Example of Crossover](image)

In figure 3.2 the one point crossover is represented. Two chromosome (parents) with test cases <T1,T2,T3,T4,T5,T6,T7,T8> and <T9,T10,T11,T12,T13,T14,T15,T16> are considered for crossover. The crossover point is two which produce two new chromosome (Childs) as < T9,T10,T3, T4,T5,T6,T7,T8> and < T1,T2,T11,T12,T13,T14,T15,T16>.

3.3.5 Mutation

With a given mutation probability, the selected chromosome goes for mutation. In epoch, it is possible that duplicate value may present in the chromosome. Our algorithm replaces these duplicate values (test case) with those which are still not considered for inclusion into the chromosome.
3.4 CRO for Test Case Prioritization

To adapt to test case prioritization problem, we design one molecule which includes these characteristics: MolSize, minHits, numHits, PE, KE, buffer, KElossRate, InitialKE, MoleColl. The potential energy PE is defined as the objective function value of the corresponding solution represented by $\omega$ the molecule. If $f$ denotes the objective function, then PE is defined as $\text{PE}_\omega = f(\omega)$. The potential energy for test case prioritization problem is defined in the following section.

3.4.1 Potential Energy (PE)

The potential energy represented by equation 3.5, assigns each test molecule a potential energy based on two major factors

- Code coverage percentage of the molecule
- Time at which each test covers its associated code in the program.

The potential energy is divided into two parts, the first component $\text{PE}_{pri}$ is used to calculate the code coverage of the entire test molecule $\omega$. It ensures the overall coverage of the molecule which is more important for test case prioritization than the ordering. $\text{PE}_{pri}$ is weighing by multiplying the percent of code covered by the program coverage weight, W. The selection of W's value should be sufficiently large so that when $\text{PE}_{pri}$ and $\text{PE}_{sec}$ are added together, $\text{PE}_{pri}$ dominates the result. The primary potential energy $\text{PE}_{pri}$ for a molecule $\omega_i$ is given by equation 3.5.

$$\text{PE}_{pri} = CC(P, \omega_i) * W \quad (3.5)$$

The second component $\text{PE}_{sec}$ considers the individual coverage of the test case. It uses incremental code coverage of the molecule, giving precedence to test molecule whose earlier tests have greater coverage. $\text{PE}_{sec}$ is also calculated in two parts. First, $\text{PE}_{sec-actual}$ is computed by adding the multiplication of the execution time ($<T_j>$) and the code coverage of the sub-molecule $\omega_k(1,j)=<T_1,T_2,...,T_j>$ for each test case $T_j \supseteq \omega_k$. Formally, for some molecule $\omega_k$ contains random test tuples which are power set of T (perms(2T)),

$$\text{PE}_{sec-actual}(P,wi) = \sum_{j=1}^{w_i} \text{time}(<T_j>) \times \text{code coverage}(p,wi(1,j)) \quad (3.6)$$

41
PE\text{sec-max} represents the possible maximum value that PE\text{sec-actual} could take (i.e., the value of PE\text{sec-actual} if $T_1$ covered 100% of the code covered by $T$.) For some molecule $\omega_i$ contains random test tuples which are power set of $T$ (perms(2$T$)),

$$PE_{\text{sec-max}}(P,wi) = \text{code coverage}(P,wi) \times \sum_{j=1}^{\|\|wi\|\|} \text{time}(<T_j>)$$  \hspace{1cm} (3.7)

Finally, PE\text{sec-actual} and PE\text{sec-max} are used to calculate the PE\text{sec}. Specifically, for some molecule $\omega_i$ contains random test tuples which are power set of $T$ (perms(2$T$)),

$$PE_{\text{sec}} = \frac{PE_{\text{sec-actual}}(P,wi)}{PE_{\text{sec-max}}(P,wi)}$$  \hspace{1cm} (3.8)

The potential energy is given by

$$PE = PE_{pri} + PE_{sec}$$  \hspace{1cm} (3.9)

3.4.2 Initial Solution Generator

This operator is used to generate molecules for CRO. It creates the molecules that of the size of MolSize. Initially, $\omega_0$ is empty, it adds test case from the set of available test case i.e $T=<T_1,T_2,\ldots,T_n>$ where $\omega_i$ contains random test cases which are the power set of $T$ (perms(2$T$)).

\begin{center}
\textbf{Initial SolnGen(MolSize)}
\end{center}

\begin{enumerate}
\item \textit{Begin}
\item $\omega I \leftarrow \phi$
\item \textit{for} $i = 1 \text{ to } |\omega I| = \text{MolSize}$
\item $\omega I \leftarrow \omega I \cup \text{Randomly generate from set(2$T$)}$
\item \textit{end for}
\item \textit{return} $\omega$
\item \textit{End}
\end{enumerate}
3.4.3 On-Wall Ineffective Collision

In CRO on wall ineffective collision is used to find a neighbor of solution \( \omega \) in search space. For our problem, mutation operator of the genetic algorithm is found suitable so it is used as on wall ineffective collision. In the process of on wall ineffective collision, one position \( i \) in solution \( \omega \) will be chosen and the value of \( \omega_i \) is replaced with that test case which is not present in the molecule.

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( T_3 )</th>
<th>( T_4 )</th>
<th>( T_5 )</th>
<th>( T_6 )</th>
<th>( T_7 )</th>
<th>( T_8 )</th>
</tr>
</thead>
</table>

**On-wall ineffective collision(\( \omega \))**

1. Begin
2. Randomly generate \( i \) from set \{1…..\( n \)\}
3. Replace \( i^{th} \) test case in \( \omega \)
4. with \( T_i \in \{ T \} - \{ \omega \} \)
5. return \( \omega' \)
6. End

\[
\begin{array}{cccccccc}
W & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & T_8 \\
W' & T_1 & T_2 & T_9 & T_4 & T_5 & T_6 & T_7 & T_8
\end{array}
\]

3.4.4 Decomposition

The diversification of CRO is achieved by decomposition. It produces two solutions from one original solution. This is designed according to the “half-total-exchange” operator that is used to solve the channel assignment problem in [124]. The operator creates two solutions \( \omega_1 \) and \( \omega_2 \) from solution \( \omega \). Firstly, \( \omega \) is duplicated to generate \( \omega_1 \) and \( \omega_2 \). After that, exchange for \( n/2 \) positions in solutions \( \omega_1 \) and \( \omega_2 \) are made randomly.
Decomposition(ω)

1. Input: a solution ω
2. Copy ω to produce ω1 and ω2
3. for change = 1 to n/2
4. do
5. Get i and j randomly in the set {1, ... , n}
6. Add random exchange to ω1(i) and ω2(j)
7. end for
8. Output ω1 and ω2

3.4.5 Synthesis Operator

The synthesis operator used in [22] is used in this paper. The operator produces a molecule ω by combining two molecules ω1 and ω2. Each molecule ω(i) is randomly selected either from ω1(i) or ω2(i).

synthesis(ω1, ω2)

1. Input: solutions ω1 and ω2
2. for i←1 to n do
3. Get t randomly in [0, 1]
4. if (t > 0.5) then
5. ω(i)←ω1(i)
6. else
7. ω(i)←ω2(i)
8. end if
9. end for
10. Output: ω
3.4.6 Inter-Molecular Ineffective Collision

Intemolecular ineffective collision produces two new molecules $\omega_1$ and $\omega_2$ from two old molecules $\omega_1$ and $\omega_2$. GA’s popular crossover is used for it. Two-point crossover is best suited so it is used in our paper. In the two-point crossover, two-point are randomly selected. These points separate a molecule in three parts. The solution $\omega_1$ is created from the even parts of $\omega_1$ combined with the odd parts of $\omega_2$. The solution $\omega_2$ is created from the even parts of $\omega_2$ combined with the odd parts of $\omega_1$.

---

**Inter_molecular_ineffective_collision($\omega_1, \omega_2$)**

1. **Input**: two solutions $\omega_1$ and $\omega_2$.
2. **Get two points** $k_1 < k_2$ randomly in $\{1, \ldots, n\}$
3. **Begin**
4. **if** ($i < k_1$ or $i > k_2$) **then**
5. $\omega_1(i) \leftarrow \omega_1(i)$
6. $\omega_2(i) \leftarrow \omega_2(i)$
7. **end if**
8. **if** ($k_1 \leq i \leq k_2$) **then**
9. $\omega_1(i) \leftarrow \omega_2(i)$
10. $\omega_2(i) \leftarrow \omega_1(i)$
11. **end if**
12. **end**
13. **Output**: two solutions $\omega_1'$ and $\omega_2'$
3.4.7 CRO Algorithm

**Input:** Problem-specific information (the objective function $f$, constraints, and the dimensions of the problem)

Assign parameter values to PopSize, KELossRate, MoleColl and InitialKE

Let $Pop$ be the set of molecules $\{1, 2, \ldots, \text{PopSize}\}$

For each of the molecules do

Assign a random solution to the molecular structure

Calculate the $PE$ by $f(\omega)$

Assign the $KE$ with $\text{InitialKE}$

end for

Let the central energy buffer be $\text{buffer}$ and assign $\text{buffer} = 0$

while the stopping criteria not met do

Get $t$ randomly in interval $[0, 1]$

if $t > \text{MoleColl}$ then

Select a molecule $M$ from $Pop$ randomly

if decomposition criterion met then

$(M_1, M_2, \text{Success})=\text{decompose}(M, \text{buffer})$

if $\text{Success}$ then

Remove $M$ from $Pop$ Add $M_1$ and $M_2$ to $Pop$

end if

end if
else
ineff-coll-on-wall(M, buffer)
end if
else
Select molecules M1 and M2 from Pop randomly
if synthesis criterion met then
(M, Success)=synthesis(M1, M2)
if Success then
Remove M1 and M2 from Pop
Add M to Pop
end if
else
inter-ineff-coll(M1, M2)
end if
end if
Check for any new minimum solution
end while
Output: The overall minimum solution and its function value

The above is a generalized CRO algorithm[127] which can be modified for a specific problem.
Figure 3.3: CRO's flow chart [127]
3.4.8 Test Case Prioritization Model

Figure 3.4 describes the procedure of execution of CRO algorithm. Before applying the TCP techniques, we collected the test case-requirement matrices from the previous execution of the test case T over program P. In the case of regression testing the test case T is prioritized using CRO and give prioritize T’. These test cases are run on the modified program P’ in the maintenance stage.

![Diagram of test case prioritization model](image)

**Figure 3.4: Model for execution of CRO prioritization procedure**

3.5 Ant-Based Prioritization Technique

The set of test case i.e. T={t1,t2,t3,……,tn} and the set of fault present in the object oriented program is F={f1,f2,f3,……,fn}. The relation between T and F is that each test case ti detects some set of fault. Let f where f ∈ F. An undirected graph G is constructed in which all vertices represented the test cases. The graph is a complete graph where |V|=|T|. Initially, all ants start from their corresponding vertices. Let the set of ant A={A1,A2,….An} and |A|≥|T|.

The solution will start by positioning the ant in a corresponding vertex. Individually each ant searches the best path by adding new edges to its path. For doing this the Choose_adjacent function is used. This function will select the edge with the highest
pheromone deposit. If all the connected edges have same pheromone value then a random
edge is selected. The selection of edges $E_{ij}$ when the current position of the ant is vertex $i$
and next vertex selected is $j$ from all the adjacent vertex of $i$. this selection is done using the
equation $3.10$.

$$E_{ij} = \begin{cases} \text{Random Edge} & \text{If pheromone is same} \\ \text{Edge With highest pheromone} & \text{Otherwise} \end{cases} \quad (3.10)$$

In a single iteration, each ant selects a set of test case which represented by a vertex
of the graph. Let the test cases selected by ant $A_i$ is $T_i$, then $T_i \in T$. Each ant will stop
exploring new path once his $T_i$ covers 100% fault. Once each ant selects test case (As a the
test case represents as a vertex in the graph) which satisfy all requirements, the next job of
the algorithm is to select the path with minimum cost i.e. the path whose execution time is
minimum. The pheromone on that path is updated and evaporation is done using function
Update_pheromone. This function adds 1 to all the edges as pheromone deposit and
evaporates 10% of the current pheromone on the best path. This process continues till no
update in the path is possible. That means when in successive iteration the best-selected
path is not changed, the algorithm stops further execution. The algorithm and its notations
are given below.

<table>
<thead>
<tr>
<th>Notations used in the algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i$ :Selected path of ant $A_i$</td>
</tr>
<tr>
<td>$T_i$ :Test case represented by a vertex of the graph.</td>
</tr>
<tr>
<td>$F_c$ : Fault covered.</td>
</tr>
<tr>
<td>$V$ : temporarily selected vertex.</td>
</tr>
<tr>
<td>$A_i$ :Ant $i$ present in started his movement from vertex from vertex represented by test case $T_i$</td>
</tr>
<tr>
<td>$Time_i$ : Time of execution of ant $i$</td>
</tr>
<tr>
<td>$Ev$ : Adjacent edges of vertex $V$</td>
</tr>
</tbody>
</table>
Algorithm 2 ACO-Pri

Input: Test Case Represented by the complete graph.

Output: A set of the prioritized test case.

1. \( ACO\text{-Pri} (\text{Test Case Represented by Complete Graph}) \)
2. \{
3. While No updation in the best path do
4. For \( i = 1 \) to \( n \)
5. \( P_i = P_i \cup \{T_i\} \)
6. \( V = T_i \)
7. \( Time_i = Time\text{ to execute } (T_i) \)
8. While \( fc = |F| \) do
9. \( t = \text{Choose_Adjacent}(A_i, V) \)
10. \( P_i = P_i \cup \{t\} \)
11. \( Time_i = Time_i + Time\text{ to execute } (t) \)
12. \( V = T_i \)
13. End
14. End
15. \( T_{\min} = \text{Min}\{P_1,P_2,P_3,\ldots,P_n\text{ according to } \{Time_1, Time_2, Time_3, \ldots, Time_n\}\} \)
16. Update_pheromone(\( T_{\min} \))
17. For \( i = 1 \) to \( n \)
18. \( P_i = \emptyset \)
19. End
20. Add the rest of test case in any order
21. \}
22. Choose_Adjacent (A_i, V)
23. \{
24. \( Ev = \{\text{Set of all edges connected to adjacent of } V\} \)
25. If \( (\text{Equal Pheromone at } Ev) \)
26. Select Random edge
27. Else
28. Select edge with max pheromone
29. \}
30. Update_pheromone(\( T_{\min} \))
31. \{
32. Update pheromone in the all edge associated with \( T_{\min} \)
33. Evaporate 10\% of pheromone
34. \}

3.5.1 Theoretical Example

Table 3.1 contains some test case and corresponding fault detected by the test case. This information has collected a priory in the first implementation of the object-oriented
programs. From the table the complete graph, let $G$ is created which is display in figure 3.5 in the graph $|V|=|T|$. From each vertex of the graph, a single or more ants will start exploring the optimal path. In this example, the no of ant will be equal to a number of vertex i.e. $|A|=|V|$.

**Table 3.1. Test case and the fault detected by them with execution time**

<table>
<thead>
<tr>
<th>Faults</th>
<th>Test case</th>
<th>Time</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>$T_1$</td>
<td>6</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_2$</td>
<td>4</td>
<td>•</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td>$T_3$</td>
<td>8</td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_4$</td>
<td>$T_4$</td>
<td>9</td>
<td>•</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_5$</td>
<td>$T_5$</td>
<td>3</td>
<td></td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_6$</td>
<td>$T_6$</td>
<td>2</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: The complete graph generated from table 3.1
Here six ant start from the six vertexes of the complete graph. After the first iteration, each ant selects an optimal path represented in table 3.2. Out of the entire six paths selected, the path selected by ant A4 is optimal. The time of execution of all the test case selected is 22 which is minimum among all the six ants.

Table 3.2: Detail of ant position after 1st iteration

<table>
<thead>
<tr>
<th>A</th>
<th>Test case selected by ant after 1st iteration</th>
<th>% Fault Covered</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>T1 T3 T6 T2 T5</td>
<td>100</td>
<td>23</td>
</tr>
<tr>
<td>A2</td>
<td>T2 T5 T4 T6 T3</td>
<td>100</td>
<td>26</td>
</tr>
<tr>
<td>A3</td>
<td>T3 T1 T5 T6 T4</td>
<td>100</td>
<td>28</td>
</tr>
<tr>
<td>A4</td>
<td>T4 T6 T3 T5</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td>A5</td>
<td>T5 T6 T1 T3 T4</td>
<td>100</td>
<td>28</td>
</tr>
<tr>
<td>A6</td>
<td>T6 T4 T1 T5 T3</td>
<td>100</td>
<td>27</td>
</tr>
</tbody>
</table>

The pheromone along the path T4-T6-T3-T5 is updated. Initially, the cost of all the edge is “0”, so 1 is added to the best path and 10% of the pheromone will be lost to evaporation. The final graph after initial iteration is denoted in figure 3.6.
In the next iteration again all the ants search an optimal path in the graph with available pheromone deposit. Out of the entire selected path, the path selected by ant A3 is optimal. It selects T3-T5-T4 vertex whose execution time will be 20. The pheromone along the path is updated. When the ant start execution from vertex T3 the pheromone at adjacent edges are e36=0.9, e35=0.9, e34=0, e32=0, e31=0. From all this adjacent edges ant will select edge with the highest pheromone deposit. The edge e36=0.9 and e35=0.9 are with the same pheromone so randomly it selects e35=0.9.

When the pheromone on this selected path is updated for edge e35, the update value is e35 =0.9 + 1=1.9. The evaporation will be 10% so at last, pheromone deposit is 1.71. Accordingly, rest of the edges are updated. The graph and table are represented in table 3.3 and figure 3.7.

Figure 3.6: The complete graph after 1st iteration with pheromone update
Table 3.3: Detail of ant position after 2nd iteration

<table>
<thead>
<tr>
<th>AN T</th>
<th>Test case selected by ant after 1st iteration</th>
<th>%Fault Covered</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>T1 T3 T6 T4 T5</td>
<td>100</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>6 8 2 9 3</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>T2 T1 T6 T5 T3</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>4 6 2 3 8</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>T3 T5 T4</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>8 3 9</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>T4 T6 T3 T5</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>9 2 8 3</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>T5 T2 T3 T6 T4</td>
<td>100</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>3 4 8 2 9</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>T6 T4 T2 T3 T5</td>
<td>100</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>2 9 4 8 2</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.7: The complete graph after 2nd iteration with pheromone update.
The 3rd iteration of the algorithm is not able to produce a new optimal path as given in table 3.3. The path selected in this phase is T3-T5-T4 which is same as the previous iteration, so the algorithm stops here and the optimal path produced is T3-T5-T4 with execution time 20.

Table 3.4: Detail of ant position after 3rd iteration

<table>
<thead>
<tr>
<th>AN</th>
<th>Test case selected by ant after 1st iteration</th>
<th>% Fault Covered</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>T1 T2 T5 T3</td>
<td>100</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 4 3 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>T2 T4 T6 T3 T5</td>
<td>100</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 9 2 8 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>T3 T5 T4</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 3 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>T4 T6 T5 T3</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9 2 3 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>T5 T3 T6 T4</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 8 2 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>T6 T3 T5 T4</td>
<td>100</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 8 3 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the sequence is generated, the rest of the test cases which are not part of the above sequence can be added in any order. In the above example, the priorities sequence is {T3-T5-T4} followed by any order of {T1-T2-T6}.

3.6 Understanding Average Percentage of Faults Detected (APFD)

Rothermel et al. [30] planned APFD to measure the average rate of fault detection of a regression test suite. The APFD is used by many researchers [31, 32] to find out the effectiveness of a test prioritization scheme. For a test suite, APFD is calculated by taking the weighted average of the quantity of fault detected throughout the execution of the program with the test suite. APFD metric values range from 0-100, where high APFD value decided the faster rate of fault detection.
Let \( n \) number of the test cases is present in test suite \( T \), and the set of fault revealed by \( T \) will be \( F \). The total no of fault present will be \( m \). In an ordering, let \( TF_i \) be the primary test case that reveals a fault \( i \). Then the average percentage of faults detected in the said ordering is obtained using the equation:

\[
APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_n}{n \times m} + \frac{1}{2n}
\]  

(3.11)

Table 3.5: Test case and fault detected by it

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Fault</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_6 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To understand APFD matrix better let us select a program which has 6 test cases. The test case are labeled as \( T_1, T_2, T_3, T_4, T_5, T_6 \) and are prioritized in two ordering sequence \( O_0 < T_1, T_2, T_3, T_4, T_5, T_6 \) and \( O_1 < T_3, T_4, T_5, T_1, T_2, T_6 \). The figure represents detection of a fault to a fraction of test cases. The area under the graph represented the average percentage of fault detected. In figure 3.8(a) it is shown that 0.17 percent of test cases detect 10% of fault whereas in figure 3.8(b) it is the same 0.17 where the test cases detect 50% of the fault.
3.7 Experiments and Results

The experimental section is represented in four sections. The first section gives details of experiments done for TCP using GA, the second section gives details of TCP using CRO, the third section contains results of TCP using ant colony optimization. Lastly the three proposed techniques are compared in the fourth section.
The TCP technique using GA, which is proposed in this chapter, was implemented in the working platform of JAVA (version JDK 1.6). The experiment is done using Eclipse editor configured with JUnit, Emma and Ant. Ant and Emma are used to collect for TCP criteria i.e. execution time and code coverage of individual test cases. The GA algorithm is implemented in MATLAB. Experiments are done for comparisons of GA technique with other existing technique i.e. random prioritization, optimal prioritization, and FEP prioritization. Experimentally it is observed that the GA takes less time as compared other technique, which is depicted in figure 3.9. The comparison indicates that GA-based prioritization is optimal in terms of time. The line representing GA-based prioritization always lies below that of other algorithms. The gap between these comparisons is there between two prioritization techniques in the entire three. The GA-based algorithm is optimal in terms of time because our fitness function prioritizes the test case by considering time as well as code coverage.

![Figure 3.9: Time of execution comparison between random, optimal, FEP test case prioritization with our algorithm](image)

The experiment further shows that the code coverage by test case generated by our algorithm is more than that of the test case generated by random test, goal oriented and path oriented test case prioritization which is depicted in figure 3.10. Here GA line is always present above the rest of the algorithms. More code is covered by the GA-based algorithm as it is designed in that way with its fitness functions. All the rest three do not consider
code coverage as criteria. That is the reason GA algorithm gives optimal code coverage figure 3.10.

![Figure 3.10: Code coverage comparison between random, optimal, FEP test case prioritization with our algorithm.](image)

The sequence which covers a combination of more code combined must expose more fault. To justify this argument we go for another experiment. To do that the APFD value of the test suit generated by all four types of prioritization technique is calculated. Formally, the APFD can be computed according to equation 3.11 as given in section 3.6 of this chapter. We calculate the fault detected using the table 3.6.

**Table 3.6 shows the number of faults detected by a test case in the test suite by each test case**

<table>
<thead>
<tr>
<th>Test case/Fault</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Number of Faults</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
For the above table, the APFD value will be

\[ APFD = 1 - \frac{3+1+2+4+1}{15*5} + \frac{1}{2*15} = 0.028 \]

Here, we have taken 15 test cases and assume that 5 faults are detected. Using this method we find out the APFD value of all the 4 prioritization techniques. To test our algorithm, we have taken 3 programs. Find out APFD of all the four prioritization technique. In table 3.7 the details of the program and APFD calculated for each technique is given.

**Table 3.7: APFD for Different Programs**

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of code</th>
<th>APFD of GA Prioritization</th>
<th>APFD of random prioritization</th>
<th>APFD of optimal prioritization</th>
<th>APFD of FEP prioritization</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>138</td>
<td>61</td>
<td>55</td>
<td>57</td>
<td>59</td>
</tr>
<tr>
<td>schedule2</td>
<td>297</td>
<td>56</td>
<td>48</td>
<td>43</td>
<td>56</td>
</tr>
<tr>
<td>Schedule</td>
<td>299</td>
<td>75</td>
<td>56</td>
<td>61</td>
<td>74</td>
</tr>
</tbody>
</table>

The result shows that our algorithm generates APFD is more than all the rest of three algorithms. The sequence of test case generated by GA-based prioritization technique is able to detect more fault. It is a little bit similar to FEP prioritization technique because FEP prioritizes the test case according to the fault exposing potential.

![Figure 3.11: APFD comparison with other techniques](image)
These experiments show that the GA algorithm is more efficient than the rest of three algorithms in terms of code coverage, time of execution or APFD.

Now we discuss the CRO based algorithm. To investigate the effectiveness of TCP based on CRO, we performed several empirical studies. The techniques used for comparison are random technique, the genetic algorithm-based technique of Sudhir and et al. [29]. In the following, we will refer to this technique as random and GA.

3.7.1 Research Questions for CRO-Pri

Q1: Can the order sequence technique increase the rate of fault detection more significantly than the two compared techniques?
Q2: Can the order sequence technique detect bugs in the loop more quickly than the other techniques?
Q3: Is the order sequence technique, efficient in terms of time and space complexity?

3.7.2 Subject Program and Test Case for CRO-Pri

Table 3 shows the details of subject programs and the collected test case-requirement matrices. Column 1 lists all the subject programs. Column 2 lists the number of lines of code (LOC) of each subject program. Column 3 lists the size of the corresponding subject program’s test suite pool where T denotes the number of all the test cases and R denotes the number of test requirements. Three programs were studied, ranging from 1425 to 3095 lines of code (LOC). These three Java programs in our experiment are power equalizer (PEQ), transmission control (TC), stock index prediction (STOCK), developed by the students in Master of Technology, at SOA University. The feature of these programs has been given in Table 3.8.
Table 3.8 Summary of the subject program used in the experiment

<table>
<thead>
<tr>
<th>Program</th>
<th>Source file (LOC)</th>
<th>Test suite pool (T X R)</th>
<th>Mutation fault</th>
<th>Total statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEQ</td>
<td>1864</td>
<td>169 X 98</td>
<td>520</td>
<td>79.2%</td>
</tr>
<tr>
<td>TC</td>
<td>2987</td>
<td>228 X 96</td>
<td>891</td>
<td>58.1%</td>
</tr>
<tr>
<td>STOCK</td>
<td>3095</td>
<td>397 X 128</td>
<td>993</td>
<td>67.2%</td>
</tr>
</tbody>
</table>

3.7.3 Acquisition of Useful Information of Test Case for CRO-Pri

We use Emma, and ant tool to collect the coverage information of an SUT(System under Test). The information collected are the code coverage, time of execution of the test case. The detail information is given in table 3.8.

3.7.4 Experimental Setup for CRO-Pri

The three techniques are implemented using MATLAB. Once program specific information is gathered, the result is used by MATLAB to generate optimal ordering. All the implemented techniques were executed on a PC with an Intel Pentium 2.26 GHz CPU and 512 M memory running the Windows 2000 Professional operating system.

Q1: In order to compare the rate of fault detection, we used APFD metric. Higher APFD value means faster fault detection rate. We have already discussed APFD in section 3.6 of this chapter.
Figure 3.12: Result of APFD on the three subject programs

The above figure 3.12 shows the box-whisker plots of the three subject programs. From the figures in 3.12, we can derive that the result of CRO based technique is higher than random and GA based techniques. From the above figure, we can conclude that the CRO based technique has higher APFD value than the other two.

Q2: In this section, we detect the potential of the algorithm to detect bugs in loops for the above three prioritization. The reason to choose loops is that the bugs have more chance to be relative to the ordered sequence of program elements measured by execution frequency. The number of bugs present in the loop to be detected is 81, 90 and 778.
Figure 3.13: Detection of bugs in loops on three subject programs
The graphs in figure 3.13 represent the number of a test case in X-axis and bugs in the loops in Y-axis. It is observed from the figure that for all the test programs CRO technique finds more bugs at the beginning. This technique will also be helpful in a situation where regression testing is terminated due to limited resources, because of its early detection of errors.

Q3: In this section, we detect the efficiency of the algorithm in terms of time and space complexity.

![Graph showing the number of generations and the number of molecule per generation](image)

**Figure 3.14:** Graph showing the number of generations and the number of molecule per generation

In the figure 3.14 X-axis represents (Number of generation, Population size), Y-axis represents the time of execution of the CRO algorithm. The test programs are taken in the experiment. The algorithms take maximum 11.66 seconds for the stock program. The memory requirement is less than 100KB as it is found in the MATLAB implementation of the program.

In this section, we investigate the effectiveness of our TCP technique based on ant colony optimization. We performed several empirical studies. The technique used for comparison is random technique. Following are two research question, based on which we carried out our experiment.

**3.7.5 Research Questions for ACO-Pri**

Q1: Can the order sequence technique increase the rate of fault detection more significantly than the random technique?
Q2: Is the order sequence technique, efficient in terms of time and space complexity?

3.7.6 Subject Program and Test Case for ACO-Pri

Table 3.9 shows the details of subject programs and the collected test case-requirement matrices. Column 1 lists all the subject programs. Column 2 lists the number of lines of code (LOC) of each subject program. Column 3 lists the size of the corresponding subject program’s test suite pool where T denotes the number of all the test cases and R denotes the number of test requirements. Three programs were studied, ranging from 1425 to 3095 lines of code (LOC). These three Java programs in our experiment are Automata lift controller (ALC), Traffic signal controlling (TSC), stock index prediction (STOCK), developed by the students in Master of Technology, at SOA University.

Table 3.9: Summary of programs used in experimentation

<table>
<thead>
<tr>
<th>Program</th>
<th>Source file (LOC)</th>
<th>Test suite pool (T X R)</th>
<th>Mutation fault</th>
<th>Total statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALC</td>
<td>1864</td>
<td>169 X 98</td>
<td>520</td>
<td>79.2%</td>
</tr>
<tr>
<td>TSC</td>
<td>2987</td>
<td>228 X 96</td>
<td>891</td>
<td>58.1%</td>
</tr>
<tr>
<td>STOCK</td>
<td>3095</td>
<td>397 X 128</td>
<td>993</td>
<td>67.2%</td>
</tr>
</tbody>
</table>

3.7.7 Acquisition of Useful Information of Test Case for ACO-Pri

We use Emma, and ant tool to collect the coverage information of an SUT(System Under Test). The information collected are the code coverage, time of execution of the test case. The detail information is given in the table.

3.7.8 Experimental Setup for ACO-Pri

The three techniques are implemented using MATLAB. Once program specific information is gathered, the result is used by MATLAB to generate optimal ordering. All the implemented techniques were executed on a PC with an Intel Pentium 2.26 GHz CPU and 512 M memory running the Windows 2000 Professional operating system.
Q1: In order to compare the rate of fault detection, we used APFD metric. Higher APFD value means faster fault detection rate.

To measure the average rate of fault detection (APFD) metric was planned by Rothermel et al. 8. It is used in papers by researcher 21, 24 of TCP for calculation of effectiveness of the prioritization ordering. The weighted average of the quantity of faults detected throughout the execution of the program is considered for the calculation of the average rate of fault detection for a test suite. A higher APFD value represents quicker fault detection rate, where the APFD value lies between 0 to 1.

It is calculated using the following expression. Let T be the initial test suite containing n test cases, and let F be a collection of m faults revealed by T. Let O0 be an ordering of T. In T0, let TFi be the primary test case that reveals a fault i. Then the APFD metric for test suite O0 will be obtained by using the equation 3.11:
APFD calculation of Automata lift controller (ALC) using random and ant colony based prioritization
APFD calculation of Traffic signal controlling (TSC) using random and ant colony based prioritization
APFD calculation of stock index prediction (STOCK) using random and ant colony based prioritization

**Figure 3.15: Result of APFD on the all three subject program**

The above figure 3.15 shows the graphs of all the three subject program. The 1st figure of each subject program represents APFD graph of random ordering which achieves 100% APFD value after execution of 100% of the test case. But when we go for Ant Colony optimization (Present in the second figure of each subject program) after execution of 50% of test suit fraction 100% APDF is achieved. The ant technique detects fault earlier than random ordering.

Q2: In this section, we detect the efficiency of the algorithm in terms of time and space complexity
In the above figure 3.16 X-axis represents (Number of searches, Number of ants), Y-axis represents the time of execution of the Ant algorithm. The test programs are taken in the experiment. The algorithms take maximum 9.06 second for the stock program. The memory requirement is less than 100KB as it is found in the MATLAB implementation of the program.

### 3.7.9 Comparisons among the three Algorithms

In this section of the chapter, we do a comparative study among three of our TCP techniques. Our experiment is based on two aspects. Firstly we calculate APFD detection by our algorithms and secondly, the execution time of the algorithm is considered. For the comparative study of our algorithm we take the following subject program which was used in TCP based on ACO. The entire three algorithms were executed on a PC with an Intel Pentium 2.26 GHz CPU and 512 M memory running the Windows 2000 Professional operating system.

#### Table 3.10: Summary of programs used in experimentation

<table>
<thead>
<tr>
<th>Program</th>
<th>Source file (LOC)</th>
<th>Test suite pool (T X R)</th>
<th>Mutation fault</th>
<th>Total statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSC</td>
<td>2987</td>
<td>228 X 96</td>
<td>891</td>
<td>58.1%</td>
</tr>
<tr>
<td>STOCK</td>
<td>3095</td>
<td>397 X 128</td>
<td>993</td>
<td>67.2%</td>
</tr>
</tbody>
</table>
Figure 3.17: Result of APFD three algorithm on the all subject program
Table 3.11: APFD value for STOCK and TSC program with different percentage of test case

(a)

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Percentage of Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.17</td>
</tr>
<tr>
<td>GA</td>
<td>45%</td>
</tr>
<tr>
<td>CRO</td>
<td>50%</td>
</tr>
<tr>
<td>ACO</td>
<td>45%</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Percentage of Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.17</td>
</tr>
<tr>
<td>GA</td>
<td>45%</td>
</tr>
<tr>
<td>CRO</td>
<td>45%</td>
</tr>
<tr>
<td>ACO</td>
<td>45%</td>
</tr>
</tbody>
</table>

All the three algorithms are executed for the two subject programs. We compared the APFD value among them. The figure 3.16 represents APFD detection vertically and a fraction of test case horizontally. The summary of the APFD is present in table 3.11 a,b. In terms of early detection of fault, the CRO and ACO based algorithm outperformed the GA based TCP technique. For the STOCK program both CRO and ACO based technique detect 100% fault after execution of 50% of the test case. But GA-based technique achieves 100% fault detection when 67% of the test cases are executed. Similarly for TSC program 100% fault is detected after 67% of the test cases are executed in case of CRO and ACO based
technique. In the same time GA-based technique covers 100% fault after 83% of the test case are executed.

Next in terms of execution time the ACO algorithm takes extra time for construction of complete graph. The complete graph is represented by adjacent matrix still it takes more time to construct. When programs of large LOC may be used, the construction of complete graph will be an overhead. If we consider the best algorithm among these three, then it is CRO. CRO execution time is less as compared to ACO and both perform well related to APFD calculation.

3.8 Summary

In this chapter, three different algorithms for prioritizing test cases for regression testing is developed and implemented. It is compared with other existing techniques. It is shown that the techniques developed by us, has the ability detect the faults to more quickly, during regression testing. Our first technique which is GA based, is found from experimental results, as more efficient compared to other prioritization techniques. It is shown that the code coverage and execution time are optimized in this technique. This technique will be equally suitable to all object oriented language, provided the code coverage and execution time are given. The second algorithm is based on CRO. The experimental analysis demonstrates that the approach can create an ordering of the test case with better APFD value by consuming a reasonable amount of time and memory. This technique detects bugs in the early stage of execution which can be used fully in early termination of regression testing due to resource constraint. The third algorithm, an Ant based algorithm, is implemented to prioritize test case. Experimental analysis of the ant-based algorithm demonstrates that the approach can create an ordering of the test case with better APFD value by consuming a reasonable amount of time and memory. We experimented the three proposed methods to find out most efficient among them. The result shows that for the experimental program, the CRO based algorithm is more efficient than other.