Optimization of Test Data for Basis Path Testing using Artificial Intelligence Techniques

Meghansh Sharma

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Optimization of Test Data for Basis Path Testing using Artificial Intelligence Techniques

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Meghansh Sharma

(Roll 211CS3300)

under the supervision of

Prof. S. K. Rath

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela, Odisha, 769 008, India

June 2013

Dedicated to my parents...

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled ***Optimization of Test Data for Basis Path Testing using Artificial Intelligence Techniques*** by ***Meghansh Sharma*** is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: June 3, 2013

(Prof. S. K. Rath)
Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Prof. S. K. Rath for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. He has helped me greatly and been a source of knowledge.
I extend my thanks to our HOD, Prof. A. K. Turuk for his valuable advices and encouragement.
I am really thankful to my all friends, especially my 'puzzle' group of best friends. I would also like to thank all the Ph.D. scholars and separately to Ph.D. scholar Yeresime Suresh for helping me and giving advise. My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism, or their invaluable time, I am truly indebted.
I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.
Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

Meghansh Sharma
Roll: 211CS3300

Abstract

Software testing is a process carried out with the intent of finding errors. This helps in analyzing the stability and quality of a software. Stability and quality can be achieved by suitable test data. Test data can be generated either manually or by automated process. Manual generation of test data is a difficult task. It involves lot of effort due to presence of huge number of predicate nodes in a module. In this report, an automated process is proposed for test data generation in traditional methodology for the automatically constructed control flow graph.

Code Coverage is a measure used in software testing process and is one of the key indicators of software quality. It helps the tester in evaluating the effectiveness of testing. It is achieved by automatically generating test data for various functions. Code coverage is not a method or a test; it is a measure which helps in improving software reliability. Effort has been made to gather code coverage information either by source code or by the requirements specified by the customer. But less attention has been paid to achieve better coverage. This report also emphasizes on code coverage, achieved through the test data generated, using some soft computing techniques.

Here, three soft computing techniques such as Genetic algorithm, Particle swarm optimization and the Clonal selection algorithm techniques have been deployed for automatic test data generation. This test data was in turn used for code coverage analysis. Experimental results show that the test data generated using Clonal selection algorithm was much more effective in achieving better code coverage over Genetic algorithm and Particle swarm optimization.

Keywords: *affinity; antibody; antigen; basis path; clone; code coverage; control flow graph; cyclomatic complexity; fitness function; test data*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 1

Introduction

1.1 Introduction

Software quality is the best indicator about product reliability and customer's satisfaction. A thorough testing ensures the quality of the software. The process of testing consists of number of activities and hence consumes much of the testing resources which account for almost 40 to 50 percent of the total software development cost [1, 2].

Software testing is generally divided into white box testing and black box testing. White box testing is also known as structural testing and basis path testing is one among them in structural testing. The emphasis is on finding specific input data. Today, researchers as well as practitioners use more common methods such as notion to perform, random method and heuristic approaches for test data generation [3]. These methods have some pitfalls in generating test data for larger and complicated programs. So other intelligence techniques have been very much used.

The process of automatic generation of test data plays a major role in software testing. Test data generation in program testing, is the process of identifying a set of test data which satisfies given testing criterion [4]. A test data generator is a tool which helps a tester in generation of test data for a given program. Most of the existing test data generators have been classified into three types viz., path wise test data generators [5, 6], data specification generators [7, 8] and random test data generators [9]. However, practically these techniques require complex algebraic computations. Hence artificial intelligence (AI) based approaches need to be used for reducing testing efforts.

A systematic process of development measures are taken to complete testing and goodness to establish test completion criteria [10]. Exhaustive testing of a program in general is not possible and code coverage is one such criterion (or a measure) to achieve completeness in testing. Code coverage is used as a measuring parameter to check software quality and the probability of defects. Coverage based testing measures the percentage of the software that is exercised in the process of testing. Code coverage analysis is a structural testing technique, where the tester apparently checks the behavior of a program with respect to the source code.

This thesis also consists of a systematic review of the approaches used by various researchers for achieving code coverage. This thesis emphasizes on the application of three algorithms, chosen on their capability to obtain global optimal solution in automatic generation of test data. Test data was generated and optimized using Genetic Algorithm (GA), Particle Swarm Optimization (PSO) and Clonal Selection Algorithm (CSA) for code coverage to indicate their efficiency in achieving better code coverage.

1.2 Test Data Generation and Optimization

Software testing is a very resource consuming task, and automating this process is one of the best way to decrease the time and cost. Automation of the testing process includes a number of steps, such as test data generation, test case execution and analysis of test results. Software testing is defined as a process, or a series of processes, designed to make sure the code does what it was designed to do and that it does not do anything unintended [11].

The main goal of software testing is to prove that the product meets all the pre-established requirements to ensure better functionality. There are two components of this objective. The first component is to prove that the requirements specification from which the software was designed is correct. The second component, is to prove that the design and coding correctly respond to the requirements [12].

1.2.1 Need for test data generation

Manual process of test data generation is a laborious work, which is expensive, time consuming, prone to errors, and not exhaustive. Manual test data generation is a difficult task due to the presence of huge number of predicate nodes in the module. So, this would lead towards a problem of NP-hard [13]. In reality, only human effort is not sufficient to generate a suitable amount of test data which would test the software successfully. The approach that can minimize the human effort is automation of test data generation process. But unfortunately, all the available automated test data generation process are very rarely used because of their inefficiency in achieving adequate coverage by the generated test data. Therefore, some intelligence-based search algorithms need to be used to generate test data.
Automatic generation of test data helps in reduction of:

i. Test case execution time.
ii. Cost of developing test cases.
iii. Manual effort in discovering errors.

1.2.2 Why to optimize

Testing a software thoroughly requires huge set of test data. Random generators do generate huge amount of test data. But testing with this random data involves lot of human effort, cost and time.
So a tester needs to employ a method where he can choose suitable test data. To check the correct execution of a software, in order to satisfy the customer requirements elicited, a tester needs to optimize the test data set by applying optimization techniques such as GA, PSO and CSA.
These AI techniques help a tester to select suitable test data from huge data set based on the fitness value of individual test data. When there is a huge data set, these optimization techniques are the best indicators for a tester to select optimal test data. Hence optimization of test data is one of the essential criteria in the field of testing.

1.3 Motivation

The major motivation of the work includes:

- For a large project, large numbers of test cases are required. Generating optimal test data reduces that testing set.

- Generating reduced set of optimal test data reduces the time and cost involved in testing.

- Automating test data generation process minimizes the labour involved in manual testing.

- Many researchers have generated new technologies and methodologies to generate test data, but they never suggest which technology is better. The work focuses on comparing three artificial intelligence techniques, to determine which technique better suites for the case study taken.

1.4 Organization of thesis

The rest of the thesis is organized as follows: **Chapter-2** discusses about the basic concepts which are used in this thesis. **Chapter-3** describes the literature survey done for the three Artificial Intelligence Techniques and for the code coverage. In **Chapter-4**, proposed work has been discussed and the techniques have been compared on the basis of test data comparison and code coverage. Finally, **Chapter-5** concludes with the summary of work done and future work.

Chapter 2

Basic Concepts

Chapter 2

Basic Concepts

The area of test data generation and optimization has been initiated almost two decades ago and still a lot of research work is going on in this field as it's a very challenging area. Manual testing is a very laborious work hence automating the generation process is needed.
This chapter provides the meaning and definitions of different terms, approaches and basic concepts used in the subsequent chapters. Section 2.1 contains some of the common terms related to software testing which are needed to understand the concept of software testing. It also discusses the two types of software testing. Section 2.2 discusses about control flow graph (CFG) and the terms used in it. And the last section 2.3 describes the definitions and the general terms used in artificial intelligence (AI) techniques. It also describes about the three different AI techniques used, viz. Genetic algorithm (GA), Particle swarm optimization (PSO) and Clonal selection Algorithm (CSA).

2.1 Testing

According to different practitioners and researchers, software testing has been defined as given below:

Testing is the process of executing a program with the intent of finding errors. [11]

A successful test is one that uncovers an as-yet-undiscovered error. [11]

Testing can show the presence of bugs but never their absence. [14]

Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. [15]
From the above definitions, this can be concluded that the software testing is done to enhance the quality of the software under test. The software must perform as per the requirements; in addition to that, it should be free from bugs. Thus software testing can be defined as,

Software testing is a process that detects important bugs with the objective of having better quality. [16]

2.1.1 Types of testing

Testing mainly consists of static testing and dynamic testing. Static testing largely maps to verification and dynamic testing to validation. Static testing is performed without executing the code, and dynamic testing is performed with the execution of code. Dynamic testing is further divided into two categories, namely, black box or functional testing and white box or structural testing.

Black box testing

Black box testing is one of the major techniques in dynamic testing. This technique considers only the functional requirements of the software or module, therefore, it's also known as functional testing. In this testing, the structure or logic is not considered. Input test data is given to the system, and results are checked against the expected outputs after executing the software.

White box testing

On the other hand, white box testing is another effective technique in dynamic testing. In this technique, the whole structure, design and code is tested. Therefore, it is also known as structural testing. Structure here means the logic of the program; hence the intention is to find the bug in the logic ensuring the internal parts are adequately tested.

Logic coverage criteria: The main goal of structural testing is to cover the whole logic. So here are some of the basic forms of logic coverage.

Statement Coverage: The assumption taken is that if all the statements of a software module are executed once, then every bug or error will be noticed. However, it's not a good criterion as it can execute each statement but can't check both outcome of a condition of the executed statement. Thus, it's a necessary but not the sufficient criterion for logic coverage.

Decision or Branch Coverage: It assumes that each decision must be executed for all possible outcomes i.e., True or False at least once. In other words, each decision or branch direction must be traversed at least once.

Condition Coverage: It states that each condition in a decision should takes on all possible outcomes at least once. But it doesn't mean that a decision has been covered as there might be a decision where two conditions are separated with an AND operator. If both conditions take opposite values from true or false then the overall outcome will always be false. The true portion is not covered of the decision in spite of covering both true and false outcomes for both conditions of that decision. [11]

2.1.2 Test data generation

A test data is a set of test cases and test data generation is one of the major phases of software testing process. Test data are used in both black-box as well as white-box testing. In black-box testing, test data are supplied to the black box and get expected output. On the other hand, test data are also needed in white box testing to cover all paths.

2.1.3 Code coverage

Code coverage dates back to 1963, when Miller and Maloney first explained that if a section of a program is not executed by at least one test, the development team has no way of knowing whether that section of code executes correctly or not. Miller and Maloney describe code coverage indirectly by indicating that, "there should be no possibility that an unusual combination of input data or conditions may bring to light an unexpected mistake in the program" [17].

Coverage-based testing gives the measures to what extent the software is exercised in the process of testing. Coverage can be applied during any stage of testing, whether it is unit testing, integration testing or system testing. Test coverage can be based on functional specification (black-box testing) or on internal program structure (white-box testing). Structure-based coverage is more commonly used [1]. Such testing can measure coverage at various granularities, including statements, lines, blocks, conditions, methods and classes. It provides a way to quantify the degree of thoroughness of White-box testing. It describes the degree to which the source code of a program has been tested. Code coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in Communications of the ACM in 1963 [17]. While exhaustive testing is not possible to attain the best possible code coverage, code coverage of 60% to 70% is often considered as an acceptable level. Difficulty in increasing the coverage past 70% is due to presence of huge number of predicate nodes in a basis path of a module. Hence, providing a higher percentage of code coverage is the only effective way to measure the quality of software.

Code coverage analysis consists of [18]:

i. Finding areas of a program that were not exercised by a set of test case,

ii. Generating additional test case to increase the code coverage efficiency, and

iii. Determining a quantitative measure of code coverage, which is an indirect measure of quality.

Code coverage has the following merits:-

i. First, reliability increases with increase in test coverage percentage [19].

ii. Second, it provides quantification of coverage related progress [20].

iii. Third, based on observational results, increase in the percentage of code coverage is one of the motivating factors for improving tests.

Code coverage acts as a report in an on-going testing process. As code coverage is one of the quantitative measures, goals for coverage can be determined, and its application in different phases of testing can also be determined. It also has some pitfalls, such as, firstly there is no underlying theory that predicts how the quality factor improves with coverage and secondly, also there is no parameter to correlate the coverage level and testing effort required.

2.2 Basis Path Testing

It is one of the oldest structural testing techniques [16]. This technique is based on the control structure of the program. On the basis of that control structure, a flow graph is developed and it is assumed that all possible paths can be covered at least once during testing. Here, modified version of path coverage criterion is used which is the most general criterion when compared to other logic coverage criteria. The problem with the path coverage is that program that contains loops can have an infinite number of possible paths and it's impractical to test all those paths.
Basis path testing is the testing technique of selecting the paths providing a basis set of execution paths through the program.

2.2.1 Control flow graph

The control flow graph (CFG) is a graphical representation of the control structure of a program. These can be prepared as a directed graph. It consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V.

Following notations are used for a control flow graph:

Node: It represents one or more procedural statements. The nodes are denoted by circles and are either numbered or labeled.

Edges or links: An edge is represented by an arrow, and it must terminate at a node. It represents the flow of control in a program.

Decision node: A node with more than one arrow leaving from it is called a decision node.

Junction node: A node with more than one arrow entering into it is called a junction node.

Region: The area bounded by some edges and nodes is called region.

As a control flow graph is drawn on the basis of the control structure of a program, Figure-2.1 shows some of the fundamental graphical notations for basic programming constructs.

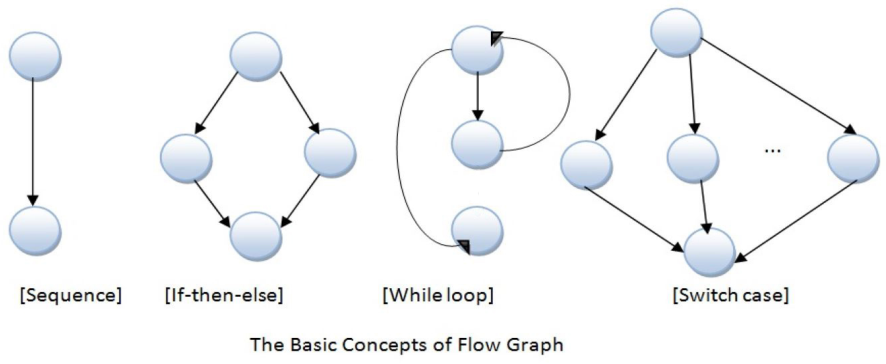


Figure 2.1: Basic Graphical Notations

2.2.2 Cyclomatic complexity

McCabe [21] introduced the concept of measuring the logical complexity of a program by considering its control flow graph . The cyclomatic complexity [22] also known as structural complexity calculates the number of independent paths through a program. It provides the upper bound of the number of test cases that must be designed, in order to ensure that all statements have been executed at least once and all conditions have been tested. McCabe's cyclomatic metric is very useful in finding the total number of independent paths present in any program.

Cyclomatic complexity can be calculated as shown below:

$$V(G) = e - n + 2p \tag{2.1}$$

where,

V(G) is the cyclomatic complexity,

p is the number of graphs,

e is the number of edges in the whole graph, and

n is the number of nodes in the whole graph.

2.2.3 Steps for basis path testing

Following are the steps that should be followed for designing test cases using basis path testing:

- Draw the CFG using the code for which test cases have to be generated.
- Determine the cyclomatic complexity of the graph.
- Cyclomatic complexity provides the number of independent paths. Find a basis set of independent paths through the program control structure.
- The basis set is the base for generating the test cases. Based on every independent path, choose the data such that this path is executed.

2.3 Artificial intelligence techniques

Artificial Intelligence (AI) or Soft computing techniques are the science, and engineering of making intelligent machines, especially intelligent computer programs. These techniques have the ability of computer, software and firmware to do those things that we, as humans, recognize as intelligent behavior [23]. A brief description of the three AI techniques deployed for generating optimal test data is presented in the following sub-sections.

2.3.1 Genetic Algorithm

Genetic algorithms are acknowledged as good solvers for tough problems. GA is a family of computational models inspired by evolution. Genetic algorithms are a population-based search method and was introduced by Holland [24]. Candidate solutions are represented as chromosomes, with the solution represented as genes in the chromosomes. A search space has been formed using possible chromosomes. These are associated with a fitness function representing the value of solutions encoded in the chromosome. The search proceeds by evaluating the fitness of each of a population of chromosomes, and afterwards performing point mutations and recombination of the successful chromosomes. GA can defeat random search in finding solutions to complex problems. GA has been successfully used to automate the generation of test data. The execution of GA begins with a set of random initial population sampled for a particular problem domain. The process of selection, crossover and mutation are applied on the initial population to get a new and fitter generation.
Outline of the basic Genetic Algorithm:

1. [**Start**] Generate random population of n chromosomes (suitable solutions for the problem).

2. [**Fitness**] Evaluate the fitness f(x) of each chromosome x in the population.

3. [**New population**] Create a new population by repeating following steps until the new population is complete:

   A. [**Selection**] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected).

   B. [**Crossover**] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

   C. [**Mutation**] With a mutation probability mutate new offspring at each locus (position in chromosome).

D. [**Accepting**] Place new offspring in a new population.

4. [**Replace**] Use new generated population for a further run of algorithm.
5. [**Test**] If the end condition is satisfied, stop, and return the best solution in current population.
6. [**Loop**] Go to step 2.

The three basic steps for Genetic Algorithm, as shown above, are:

1. **Selection:** In selection (also known as reproduction), chromosomes are selected from the population to be parents to cross over and produce offspring. The various methods of selecting chromosomes for parents to cross over are:
   a) Roulette-wheel selection
   b) Boltzmann selection
   c) Tournament selection
   d) Rank selection
   e) Steady-state selection
2. **Cross over:** After the selection phase, the off-springs are enriched with better individuals. Cross over process is applied to the mating pool, with a hope that it would create a better string. It also has three steps, firstly, the reproduction stage selects randomly a pair of two individual strings for mating. Secondly, a cross-site is selected at random along the string length and at last their position values are swapped between those two strings.

   Different cross over types are:
   a) Single-site cross over
   b) Two-point cross over
   c) Multi-point cross over
   d) Uniform cross over

   e) Matrix cross over

3. **Mutation:** After the cross over process, the strings are mutated. It involves flipping of bits, changing 0 to 1 and vice versa with a small mutation probability Pm. A number between 0 to 1 is chosen randomly and if the number is less than Pm then the bit is changed, otherwise it is kept unchanged.

2.3.2 Particle Swarm Optimization

In comparison with GA search, the PSO is a relatively recent optimization technique of the swarm intelligence paradigm. It was introduced in 1995 by Kennedy et al [25]. Inspired by social metaphors of behavior and swarm theory, simple methods were developed for efficiently optimizing non-linear mathematical functions. Similar to GA search, the system is initialized with a population of random solutions, called particles. Each particle maintains its own current position, its present velocity and its personal best position explored so far. The swarm is also aware of the global best position achieved by all its members.
The iterative appliance of updated rules leads to a stochastic manipulation of velocities. During the process of optimization the particles explore the d-dimensional space, whereas their trajectories depend both on their personal experiences, on those of their neighbors, and the whole swarm respectively. This leads to further explorations of regions that turned out to be profitable. The best previous position of particle i is denoted by p_{best_i}, the best previous position of the entire population is called g_{best}.
The termination criterion can be a specific fitness value, the achievement of a maximum number of iterations or the general convergence of the swarm itself. Since its first presentation, many improvements and extensions have been worked out to improve the algorithm in various ways and have provided promising results for the optimization of well-known test functions. A novel and auspicious approach is the Comprehensive Learning Particle Swarm Optimizer [26]. It applies a new learning strategy, where each particle learns from different neighbors for each dimension separately dependent on its assigned learning rate Pc_i. This happens

until the particle does not achieve any further improvement in a specific number of iterations called the refreshing gap ‘m’; finally yielding a re-assignment of particles. The mentioned learning-rate Pc_i of particle ‘i’ is a probability lying between 0.05 and 0.5, determining whether particle ‘i’ learns from its own or another particle’s p_{best} for the current dimension. This probability is assigned to each particle during initialization and remains unchanged for assuring the particle’s diverse levels of exploration and exploitation abilities [25]. If the considered dimension is to be learned from another particle’s p_{best}, this particle will be appointed in a tournament selection manner: two particles are randomly chosen and the better one is selected among the two. This is done for each dimension d and the resulting list of particles f_i(d) is used in the following velocity update rule for time stamp ‘t’. In each time-stamp, a particle has to move to a new position. It does this by adjusting its velocity through this equation:

$$V_i^{k+1} = w * V_i^k + c_1 * rand_1() * (p_{best_i} - s_i^k) + c_2 * rand_2() * (g_{best} - s_i^k) \tag{2.2}$$

and the position is updated through the equation:

$$s_i^{k+1} = s_i^k + V_i^{k+1} \tag{2.3}$$

where,

V_i^k : velocity of agent ‘i’ at iteration k,

w: weighting function,

c_j : weighting factor,

$rand_j$: uniformly distributed random number between 0 and 1,

s_i^k : current position of agent ‘i’ at iteration k,

p_{best_i} : personal best of agent ‘i’,

g_{best}: global best of the group.

The fitness of a particle often depends on all D parameters. Hence a particle close to the optimum in some dimensions can probably be evaluated with a poor fitness when processed by the original PSO version due to the poor solutions of the remaining dimensions. This is counteracted by the learning strategy presented in, which consequently enables higher quality solutions to be located. It was shown that Comprehensive Learning Particle Swarm Optimization [26] in comparison to some other PSO variants yields significantly better solutions for multi-modal problems [25]. Figure-2.2 shows the working principle of PSO.

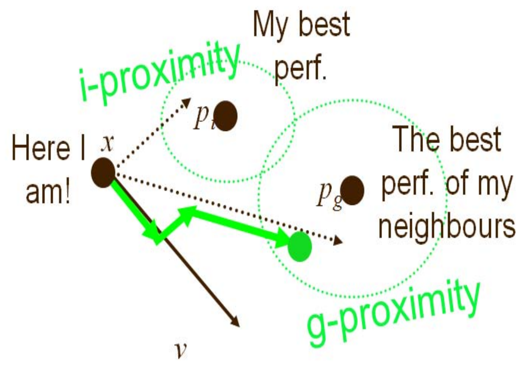


Figure 2.2: Working Principle of PSO

2.3.3 Clonal Selection Algorithm

The Clonal Selection Algorithm (CSA) [27] is an optimization algorithm based on biological immune system, in which the antigen corresponds to the problem under solved and the antibody corresponds to a solution to the problem. Clonal algorithm (CLONALG) is one of the many branches of artificial immune system algorithms with unique inherent properties that make it very efficient optimization technique [28]. In basis path testing, the aim of the tester is to find suitable test data that will satisfy the given target path. To achieve this, the random test data (input values) has been encoded as antibodies and the antigens as the ones satisfying the test data requirements. The CSA begins with a randomly generated initial population (similar to that of GA). The test data are evaluated based on affinity function (referred to as fitness function in GA). This affinity function is a description of how best the individual test data perform in code coverage. The antibody clone each population to produce their own clones based on affinity value. In CSA the process of crossover is overcome by performing hyper-mutation operation. This step helps in achieving diversified test data. The process of cloning and hyper-mutation continue till the stopping criterion is encountered. Each antibody clones a clonal population according to the affinity where better members clone

more antibodies. Diversification of the antibodies is achieved through mutation and selection process.
The process of cloning, hyper-mutation and selection continues until the termination condition is encountered. After that, the test data that satisfies the requirement would come out or the target path is determined to be an unreachable path. Here the input parameters are referred as antibodies and the test data satisfying the requirement as antigen.

Chapter 3

Literature Survey

Chapter 3

Literature Review

This chapter discusses about the related work done in this field.
In this thesis, three soft computing based approaches have been used viz., GA, PSO and CSA for automated generation of test data and in turn this test data is used for code coverage analysis. The following tables give the details about the various techniques used by different authors for generation of test data using GA, PSO and CSA. Table 3.1 shows the literature review of various techniques used by different researchers for test data generation using GA.

Table 3.1: Test Data Generation using GA

Author	**Criteria for Testing**
L. Clarke [5]	Path coverage: executed selected target paths, and then generated test data such that the identified constraints are satisfied.
Bogdan Korel [4]	Dynamic path testing: generated test data by executing the program with different possible test data values.
Srivastava P. R [29]	Focused on path coverage, and generated test cases using GA.
Lin J.C, Yeh P.L [30]	Path testing: automated test data generation using GA.
Ahmed M.A, Hermadi [32]	Path coverage criteria to generate test data using genetic algorithm.

Similarly, Table 3.2 and Table 3.3 show the various issues addressed in generating test data by using PSO and CSA respectively.

Table 3.2: Test Data Generation using PSO

Author	**Criteria for Testing**
Chengying Mao et al. [33]	Test data generation for structural program using swarm intelligence.
Huanhuan Cui et al. [34]	Efficient automated test data generation method.
Sheng Zhang et al. [35].	Hybrid approach using GA and PSO for automatic test data generation.
Rui Ding et al. [36]	Automatic test data generation based on hybrid particle swarm genetic algorithm.

Table 3.3: Test Data Generation using CSA

Author	**Criteria for Testing**
Xiaofeng Xu et al. [37]	Clonal selection algorithm for test data generation based on basis paths.
Ankur Pachauri et al. [38]	Test data generation based on branch distance using clonal selection algorithm.

Table 3.4 gives the literature survey tabulation on code coverage.

Table 3.4: Literature Survey on Code Coverage

Author	**Issues Addressed**
Williams [39]	Code Coverage as a stopping criterion for unit testing.
Qian Yang [1]	Coverage-based testing tools.
Prasanna [40]	Test case generation techniques to satisfy test coverage criteria.
Audris Mockus [41]	Test effectiveness.
Y. Wei [42]	Coverage as Testing effectiveness.
Anna Derezinska [43]	Test suits improvement.
Jun-Ru [44]	MC/DC Coverage Criterion.
Stefan Berner [45]	Impact on Testing.
Lloyd Malloy [46]	Test coverage adequacy.
Joseph Lawarance [47]	Code coverage visualization.

Chapter 4

Proposed Work

Chapter 4

Proposed Work

This chapter discusses about the proposed work and the implementation done. It describes the details of the approaches followed for generating optimal test data and the results depicting the comparison of the three techniques used. This chapter is divided into four section's viz.; Section-4.1 gives a brief explanation about how the Control flow graph is generated. Section-4.2 describes about how the three Artificial Intelligence techniques have been used in generating optimal test data. And the last Section-4.3 shows the comparison of the three techniques on the basis of test data comparison and code coverage.
The following are the steps involved in generating test data using AI Techniques:

1. Using the program's instrumented code, the CFG is constructed automatically by parsing the source code in JAVA using applet and jgraphx library.

2. Determine the Cyclomatic complexity of the flow graph.

3. Determine the basis set of independent paths.

4. Prepare test data that will force the execution of each path in the basis set.

Figure-4.1 depicts the flow chart for the proposed work.

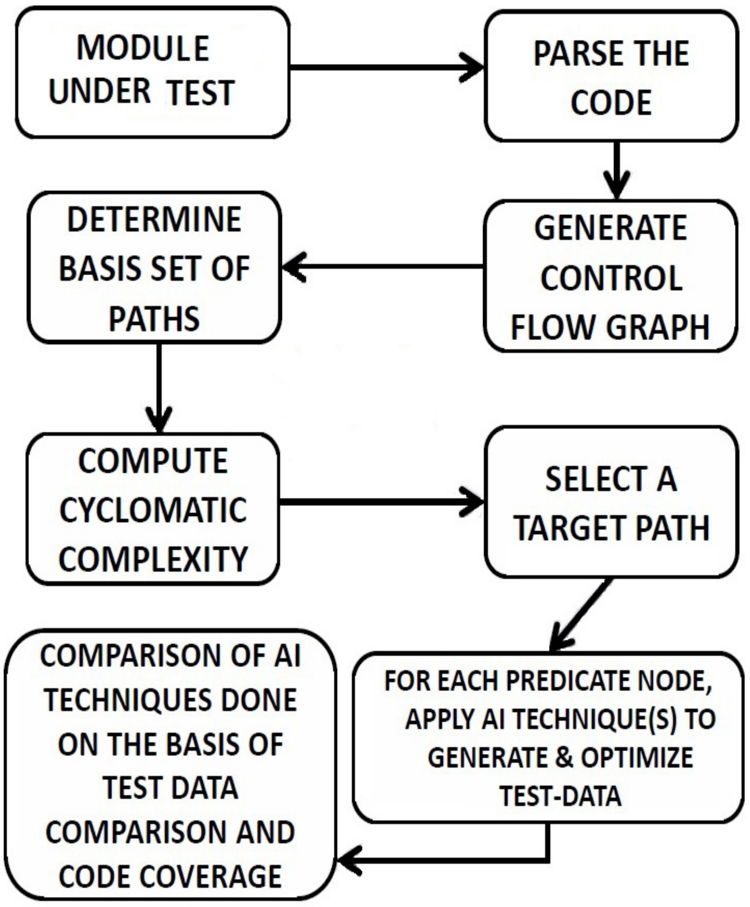
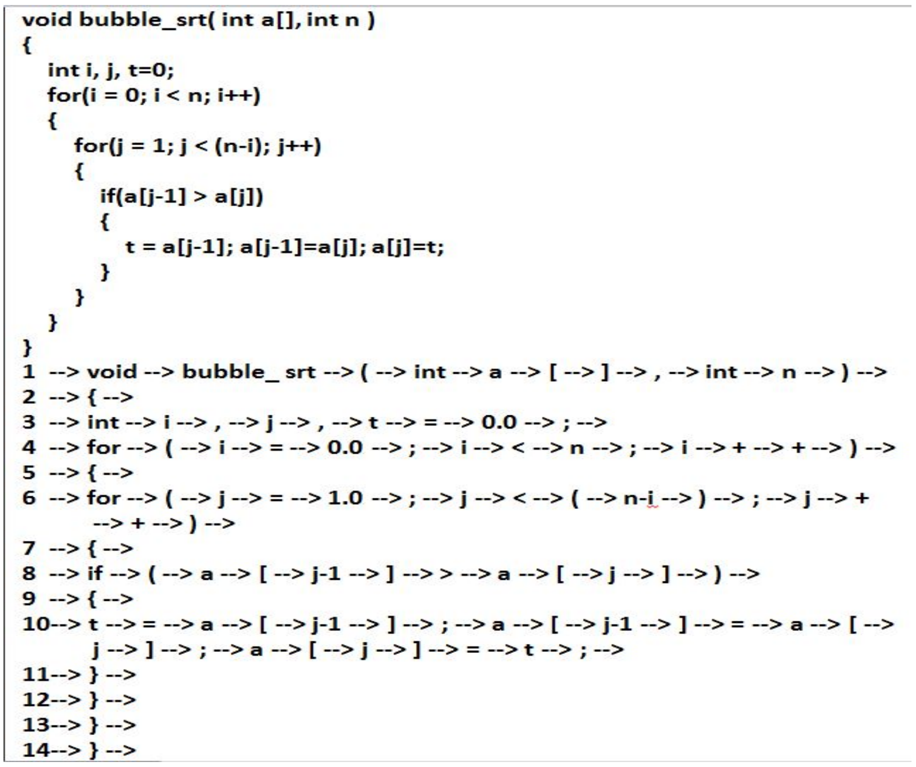

MODULE UNDER TEST
PARSE THE CODE
GENERATE CONTROL FLOW GRAPH
DETERMINE BASIS SET OF PATHS
COMPUTE CYCLOMATIC COMPLEXITY
SELECT A TARGET PATH
FOR EACH PREDICATE NODE, APPLY AI TECHNIQUE(S) TO GENERATE & OPTIMIZE TEST-DATA
COMPARISON OF AI TECHNIQUES DONE ON THE BASIS OF TEST DATA COMPARISON AND CODE COVERAGE

Figure 4.1: Flow Chart for Proposed Work

4.1 Generation of Control Flow Graph (CFG)

The control flow graph was constructed using source code. Source code is taken as input and the instrumented code is generated in which all the blank lines, white-spaces are omitted. After that, instrumented code is parsed and stored, line by line in adjacency list. Stored data are parsed and then represented as an adjacency graph with that line number. Now from the adjacency graph, CFG is generated using applet and jgraphx (built-in library) in JAVA.

Example 1: A small example of bubble sort has been taken and the CFG is generated. The bubble sort program has been parsed and the parsed adjacency list is shown in Figure-4.2.



```
void bubble_srt( int a[], int n )
{
   int i, j, t=0;
   for(i = 0; i < n; i++)
   {
      for(j = 1; j < (n-i); j++)
      {
         if(a[j-1] > a[j])
         {
            t = a[j-1]; a[j-1]=a[j]; a[j]=t;
         }
      }
   }
}
1  --> void --> bubble_ srt --> ( --> int --> a --> [ --> ] --> , --> int --> n --> ) -->
2  --> { -->
3  --> int --> i --> , --> j --> , --> t --> = --> 0.0 --> ; -->
4  --> for --> ( --> i --> = --> 0.0 --> ; --> i --> < --> n --> ; --> i --> + --> + --> ) -->
5  --> { -->
6  --> for --> ( --> j --> = --> 1.0 --> ; --> j --> < --> ( --> n-i --> ) --> ; --> j --> +
       --> + --> ) -->
7  --> { -->
8  --> if --> ( --> a --> [ --> j-1 --> ] --> > --> a --> [ --> j --> ] --> ) -->
9  --> { -->
10--> t --> = --> a --> [ --> j-1 --> ] --> ; --> a --> [ --> j-1 --> ] --> = --> a --> [ -->
       j --> ] --> ; --> a --> [ --> j --> ] --> = --> t --> ; -->
11--> } -->
12--> } -->
13--> } -->
14--> } -->
```

Figure 4.2: Snapshot of Parsed Adjacency List for Bubble Sort

Parsed adjacency list is converted to adjacency matrix from which the CFG is generated which is shown in Figure-4.3.

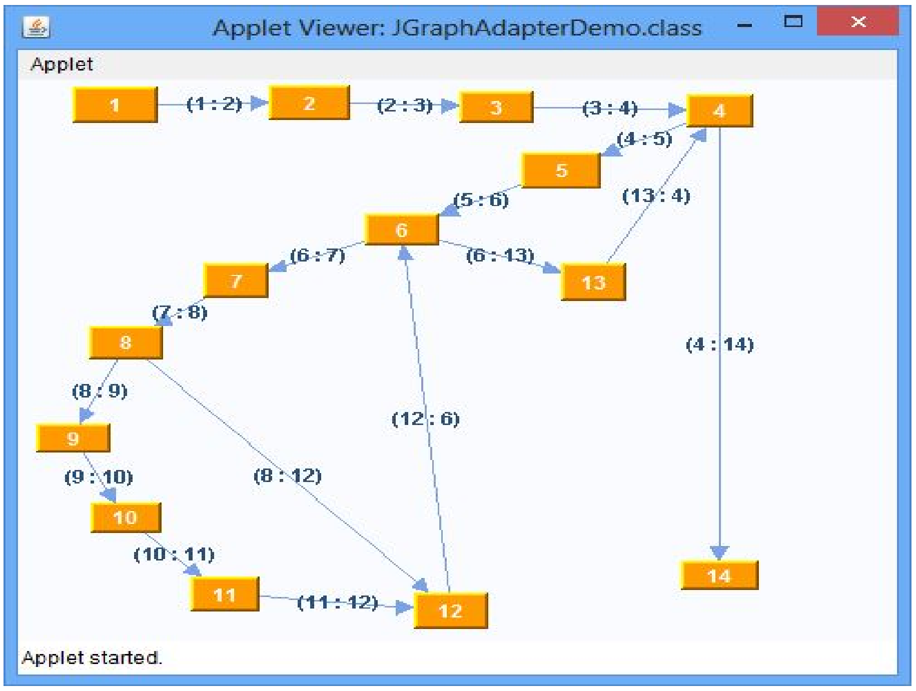


Figure 4.3: CFG for Bubble Sort

Case Study: A case study has been taken, describing a customer's activity of withdrawing money from ATM [48]. The scenario considered here for design of fitness function is that the customer tries to withdraw certain amount from the ATM machine (this withdrawal amount is the initial test data generated randomly, with an assumption that customer entering the withdrawal amount is random). The Figure- 4.4 shows the sequence of operations performed in ATM withdrawal task by the customer, drawn using IBM Rational Rose.

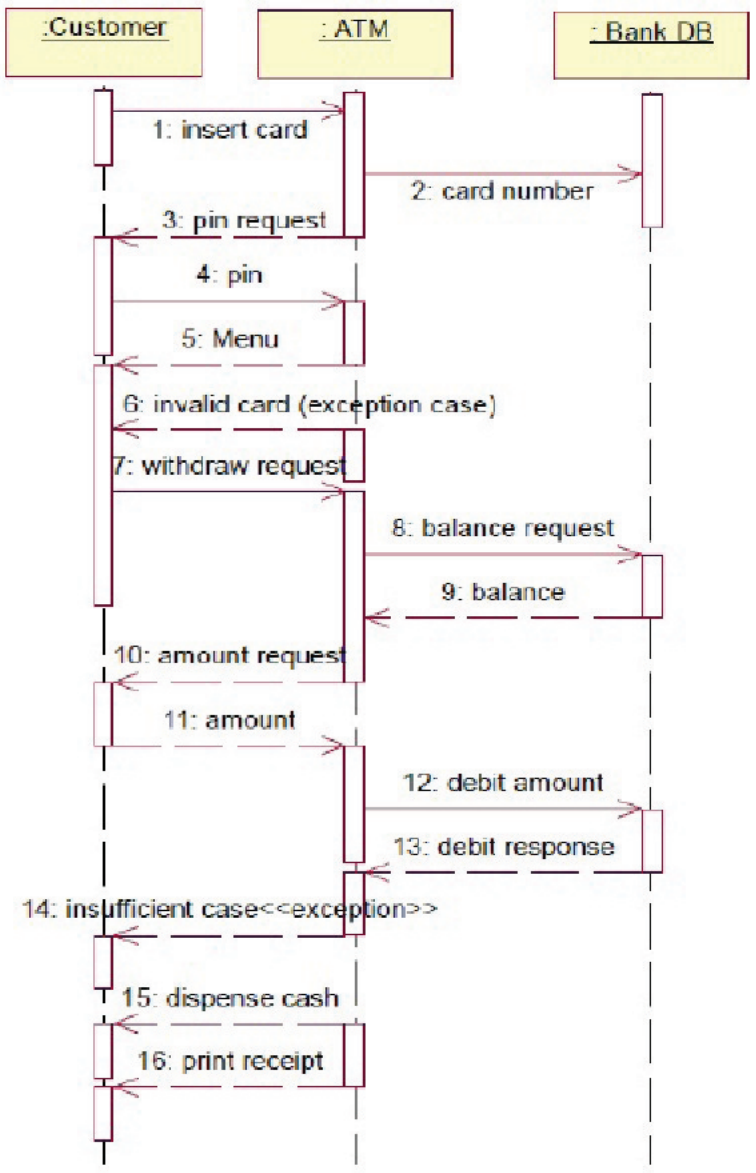


Figure 4.4: Sequence diagram for ATM withdrawal

The ATM system sends the amount and the account number to the bank system. The bank system retrieves the current balance of the corresponding account and compares it with the entered amount. If the balance amount is found to be greater than the entered withdrawal amount then the amount can be withdrawn and the bank system returns true, after which the customer can withdraw the money, otherwise it checks for credit limit if the entered amount is less than the total amount (current balance) then return false. Depending on the return value, the ATM machine dispenses the cash and prints the receipt or displays the failure message.
The following segment shows the source code for ATM withdrawal scenario:

```
1. $net\_amt = 25000, min\_bal = 1000;$
2. $bal(1,i) = net\_amt - wd\_amt(1,i);$
3. if $wd\_amt(1,i) < net\_amt$
4.     if $bal(1,i) < min\_bal$
5.         $fail\_bal(1,k) = bal(1,i);$
       else
6.         $suc\_bal(1,p) = bal(1,i);$
7. $test\_data(1,p) = wd\_amt(1,i);$
```

Taking the source code for ATM withdrawal scenario as an input, the CFG is generated as shown in Figure-4.5 and Table-4.1 shows the alphabetical representation of nodes for Figure-4.5.

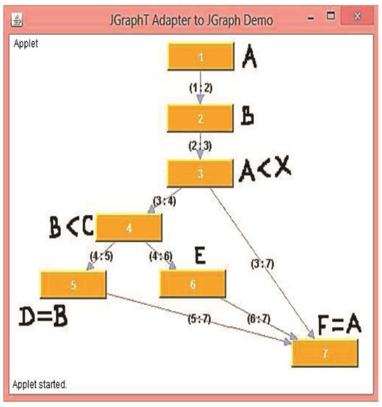


Figure 4.5: CFG for ATM withdrawal

Table 4.1: Alphabetical representation of nodes in CFG for Figure-4.5

Nodes	**Alphabetical Notation**
wd_amt	A
net_amt	X
bal	B
min_bal	C
fail_bal	D
suc_bal	E
test_data	F

4.2 Test Data Generation and Optimization

This section gives the details of the approaches followed for generating optimal test data. This section is subdivided into three subsection's viz., the first sub-section describes about the application of the fitness function for GA, second sub-section highlights the use of fitness function in PSO, and the last sub-section gives the details about test data generation using CSA.

4.2.1 Generating optimal test data using GA based on fitness function

The principle of GA has been applied to generate test data automatically. The developed system generates optimal test data automatically on the basis of basis paths in the CFG. The first generation is generated randomly and then by performing the basic GA steps, fitness of individuals gets improved.

Fitness function derivability based on Korel's Distance Function

A fitness function for test data generation for an ATM withdrawal task is developed based on Bogdan Korel's distance function [4]. A path P is considered in the program execution. The goal of the test data generation problem is to find a program input x on which P will be traversed. Without loss of generality, Korel assumed that the branch predicates are simple relational expressions (inequalities and equalities). That is, all branch predicates are of the form: E1 op E2, where E1 and E2 are the arithmetic expressions and op is one of {$<$, $\leq$, $>$, $\geq$, $=$, $\neq$}. In addition, he assumed that predicates do not contain AND or OR or any other boolean operators. Each branch predicates E1 op E2 can be transformed to the equivalent predicate of the form: F rel O (Operator), where F and rel are given in Table-4.2.

Table 4.2: Relation between F and Operator

Branch Predicate	**Branch Function F**	**rel**
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	abs$(E_1 - E_2)$	$=$
$E_1 \neq E_2$	abs$(E_1 - E_2)$	$\leq$

F is a real valued function, referred to as branch function, which is:

i. Positive (or zero if rel is $<$) when a branch predicate is false or
ii. Negative (or zero if rel is $=$ or $\leq$) when the branch predicate is true [4].

It is obvious that F is actually a function program input. But this process requires a very large and complex algebraic manipulation. For this reason an alternative approach was used in which the branch function was evaluated. Basis path testing includes both statement testing and branch testing. For example, to test "if a $>$ b then", it has a branch function F, whose value can be computed for a given input by executing the program and evaluating 'a $-$ b' expression.

This concept was used in the approach to test the Automatic Teller Machine (ATM) withdrawal task. Test data was generated for a single feasible path in CFG with respect to ATM withdrawal task [48].

The fitness function for the ATM withdrawal scenario was based on the traversal of predicate nodes. For instance, in Figure-4.6 when node-1 is visited, the condition of the predicate node may be either

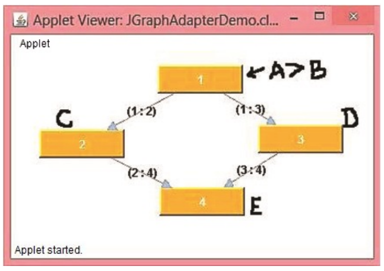


Figure 4.6: CFG for a sample code block

$A > B$ or $B > A$ or even $A = B$. So taking equality condition into consideration, $A = B \Rightarrow A - B = 0$; as GA for test data generation is minimization problem, the fitness function 'f' is given as $1/(A - B)$. But this functional value will evaluate to infinity when $A - B = 0$, so to avoid this condition a small delta value ($\delta = 0.05$) is added to the fitness function. Hence the fitness function in general is given as:

$$f = 1/((abs(A - B) + 0.05)^2).$$

The set of test data generated randomly is the initial population input for an AI technique to start.

The fitness values are calculated for each individual chromosome (test data) and on the basis of these values it performs crossover and mutation. This process continues until all individuals reach to the maximum fitness. The system performs all operations from initial population to last generation automatically; it does not require the user interference. Algorithm-1 shows the approach followed to

Algorithm 1 **- Test Data Generation using GA**

```
Input: Randomly generated numbers based on the target path to be covered.
Output: Test data for the target path.
Begin
Gen = 0.
while Gen < 500 do                              // <—Step 3
    Evaluate the fitness value of each chromosome based on the objective function.
    Fitness function : $f = 1/((abs(suc\_bal(i) - min\_bal) + 0.05)^2)$
    Use Elitism as selection operator, to select the individuals to enter into the mating pool.
    Perform two-point cross over on the individuals in the mating pool, to generate the new population.
    Perform bitwise Mutation on chromosomes of the new population.
    Gen = Gen + 1;
    go to Step 3.
end while
Select the chromosome having the best fitness value as the desired result (test data for target path).
```

generate test data for the basis path derived from CFG using GA. Figure-4.7 shows the basic flow of test data generation using GA.

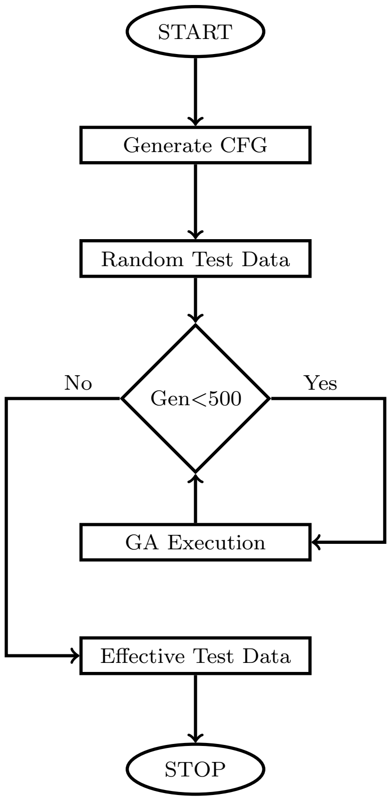


Figure 4.7: Basic flow of test data generation using GA

Test data was derived based on the set of basis paths, depends on the programs structure with an aim to traverse every executable statement in the program. The fitness function used was on the basis of branch distance [4]. The input variables were represented in binary form. The main objective of using GA lies in the ability to handle input data which may be complex in nature. Thus, the problem of test data generation is treated entirely as an optimization problem. The benefit of using GA is that through the search and optimization process, test data sets are improved in a manner that they are at or close to the input domain.

4.2.2 Generating optimal test data using PSO based on fitness function

The particle swarm optimization technique has been applied to generate test data automatically. The same approach which was used in GA is used here to generate

test data based on basis paths in the control flow graph, wherein each particle in swarm updates its p_{best} and g_{best} values. The first generation is generated randomly and then by performing the basic PSO steps the suitable test data is generated. Each particle in the swarm has its personal best fitness value, which gets updated with its neighbor in further iterations which is referred to as global best.
After each and every iteration, the particle updates its velocity and position; and this process continues until the stopping criterion is met. The system performs all operations from initial population to last generation automatically; it does not require the user interference. Figure-4.8 shows the basic block diagram of PSO execution.

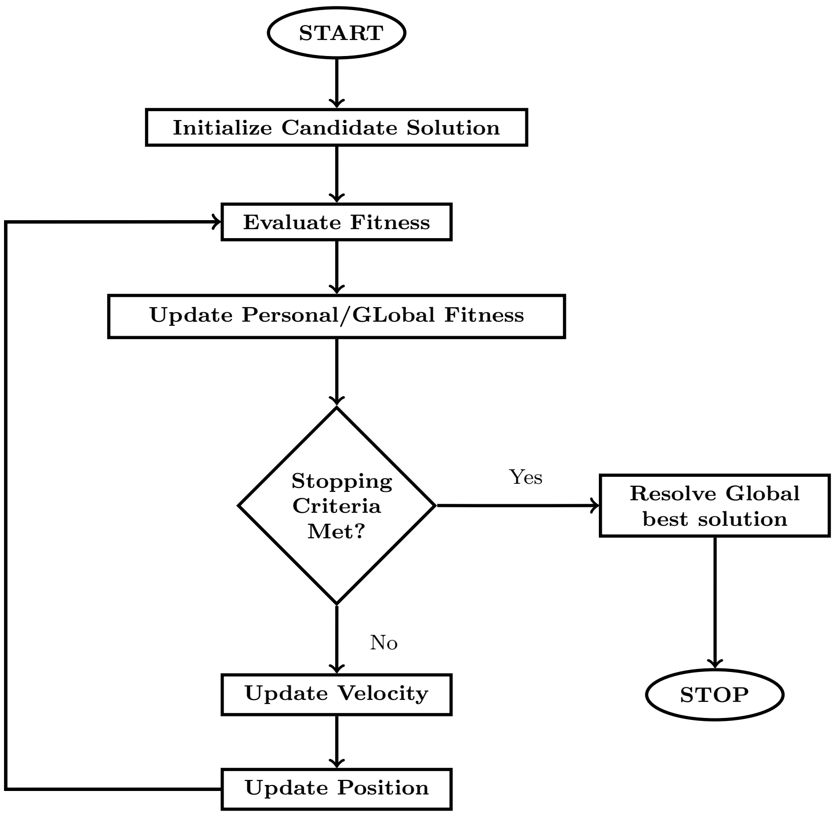

START
Initialize Candidate Solution
Evaluate Fitness
Update Personal/GLobal Fitness
Stopping Criteria Met?
Yes
Resolve Global best solution
No
Update Velocity
STOP
Update Position

Figure 4.8: Basic flow of test data generation using PSO

4.2.3 Generating optimal test data using CSA based on affinity function

The CSA algorithm was modified according to our objective which was given by Castro and Zuben [27]. The experimental settings used are given in the results section. The affinity function was based Korel's branch distance function [4]. The test data was generated by using CSA for a target basis path. Algorithm-2 shows approach to generate test data for the basis path (derived from CFG) using CSA.

Algorithm 2 - **Test Data Generation using CSA**

```
Begin
Initialize the number of generation(Gen) = 0;
Initialize the initial population randomly $A_o$
Evaluate Affinity Function: $F = 1/(abs((net\_bal - (wd\_amt - min\_bal)) + 0.05))$ // <—Step 3
if $Gen > 500$ then
    Print results (test data);
    Exit
else
    Clone: initial population($A_n$) to $A_n$';
    Hyper-mutate $A_n$'to $A_n$";
    Evaluate and Select $A_n$";
    Destroy and renew (based on $N_r$ = N/2, $N_s = N_r/10$)
      to Construct a new population $A_n$;
    Gen++;
end if
goto Step 3.
```

4.3 Experimental Settings and Results

In this section, the experimental settings and results obtained in generating optimal test data are shown. The test data were generated for a target path of Figure-4.5 as explained in Section-4.2. Table-4.3 gives the details of the experi-

Table 4.3: Experimental Setup

Genetic Algorithm	**Particle Swarm Optimization**	**Clonal Selection Algorithm**
Fitness Function = $1/((abs(suc\_bal(i) - min\_bal) + 0.05)^2)$	Fitness Function = $1/(abs((net\_bal - wd\_amt) * 100 - min\_bal + 0.05)^2)$	Affinity Function = $1/(abs((net\_bal - (wd\_amt - min\_bal)) + 0.05))$
Coding: Binary string, Chromosome length:15 bits	$c_1 = c_2 = 2.0$, w = 0.5	Coding: Binary string, Antigen length: 8 bits
Population size (N): 100	Population size (N): 100	Population size (N): 100
Selection method: Elitism, Two-point cross over(p_c): 0.5, Mutation probability(p_m): 0.05	$V_i^{k+1} = w * V_i^k + c_1 * rand_1() * (p_{best_i} - s_i^k) + c_2 * rand_2() * (g_{best} - s_i^k)$ ...(1) $s_i^{k+1} = s_i^k + V_i^{k+1}$ ...(2)	Selection method: Elitism, Hyper Mutation (p_m): 0.15, $Nr = N/2; Ns = Nr/10$; Ns-Worst antibodies; Nr-Renewed antibodies.
Stopping criteria: # of generations = 500	Stopping criteria: # of generations = 500	Stopping criteria: # of generations = 500

mental settings used in the generation of test data using GA, PSO and CSA. Following sub-sections gives us the details of the range of fitness values of test data and the achieved coverage values.

4.3.1 Test data comparison

Automated test data generation was performed for a withdrawal task of an Automatic Teller Machine (ATM) by a customer. The implementation of code coverage and test data generation for a target path using the three soft computing techniques viz., GA, PSO and CSA were carried out in MATLAB.
Tables-4.4, 4.5, and 4.6 give us the tabulation of range of fitness/affinity values obtained by applying the experimental settings shown in Table-4.3 for GA, PSO and CSA respectively. These tables give us a clear indication that chromosomes having higher fitness (affinity) value, lie in the range between 0.7 to 1.0. These fitness values are an indication of optimal test data obtained.
Figure-4.9 gives the graphical representation of the fitness values shown in Table-4.4, 4.5, and 4.6. It shows that CSA is giving better test data when compared to GA and PSO.

Table 4.4: Class of test data having maximum fitness value in GA

Fitness Value Range	**% of Test Data**
$0 \le f(x) < 0.3$	60.82
$0.3 \le f(x) < 0.7$	1.03
$0.7 \le f(x) < 1.0$	38.14

Table 4.5: Class of test data having maximum fitness value in PSO

Fitness Value Range	**% of Test Data**
$0 \le f(x) < 0.3$	50
$0.3 \le f(x) < 0.7$	6.67
$0.7 \le f(x) < 1.0$	43.33

Table 4.6: Class of test data having maximum fitness value in CSA

Fitness Value Range	**% of Test Data**
$0 \leq f(x) < 0.3$	13
$0.3 \leq f(x) < 0.7$	17
$0.7 \leq f(x) < 1.0$	70

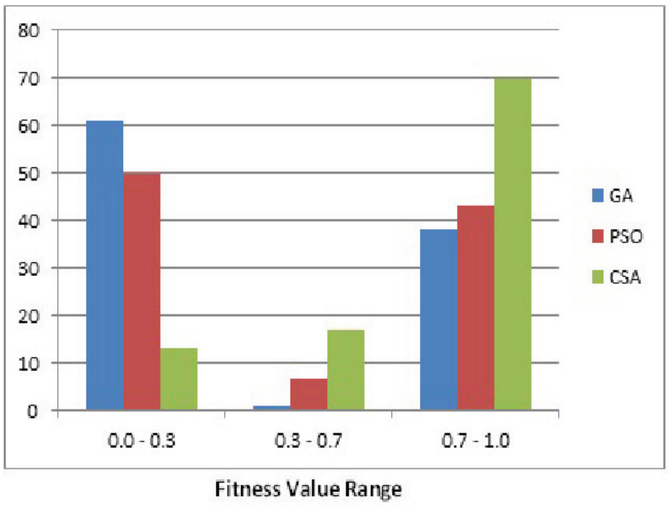


Figure 4.9: Comparison of fitness values vs % of Test Data

4.3.2 Code coverage analysis

Table-4.7 gives the analysis result of the obtained code coverage, by covering the nodes traversed in the CFG. The test data of the respective fitness function values (of GA, PSO and CSA) were used separately to achieve code coverage.

Table 4.7: Code coverage analysis

AI Technique	**# of Nodes Covered Out of 7**	**Code Coverage (%)**
Genetic Algorithm	4	57.14
Particle Swarm Optimization	4	57.14
Clonal Selection Algorithm	6	85.71

Results shown in Table-4.7 indicates that the CSA obtained better coverage of nodes in the basis path when compared to GA and PSO. Figure-4.10 gives the comparison of fitness values of unique test data for GA, PSO and CSA.

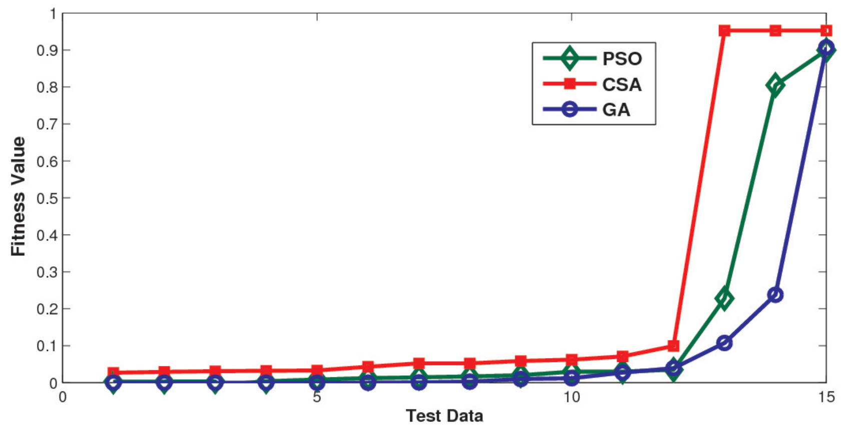


Figure 4.10: Comparison of fitness values of unique test data generated using GA, PSO and CSA

Chapter 5

Conclusion and Future Work

Chapter 5

Conclusion and Future Work

Software testing is one of the critical phases in software development process. Testing is helpful in delivering a quality software product to the customer. This objective can be achieved through thorough testing and choosing suitable test data for testing.

In this thesis, an attempt has been made to generate test data automatically for traditional methodology based on the automated generated control flow graph using three Artificial Intelligence Techniques. To generate suitable data, methods were traversed to cover each node. Test data values were selected based on fitness/affinity values of antibodies which satisfy the predicate node.

Since manual generation of test data consumes much of the computational time, the process of test data generation has been automated. Based on the predicate node condition, test data was generated and these algorithms were applied. CSA was able to generate more suitable test data when compared to Genetic algorithm and Particle swarm optimization on the basis of test data comparison and code coverage.

Future perspective of this work is to achieve better code coverage using hybrid approaches of AI techniques. Another prospective would be to generate optimal test data for large and complex programs.

Bibliography

[1] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[2] F. P. Brooks, *The mythical man month: Essays on software engineering.* Addison-Wesley Professional, 1995.

[3] W. Xibo and S. Na, "Automatic test data generation for path testing using genetic algorithms," in *Measuring Technology and Mechatronics Automation (ICMTMA), 2011 Third International Conference on*, vol. 1, pp. 596–599, IEEE, 2011.

[4] B. Korel, "Automated software test data generation," *Software Engineering, IEEE Transactions on*, vol. 16, no. 8, pp. 870–879, 1990.

[5] L. A. Clarke, "A system to generate test data and symbolically execute programs," *Software Engineering, IEEE Transactions on*, no. 3, pp. 215–222, 1976.

[6] C. V. Ramamoorthy, S.-B. Ho, and W. Chen, "On the automated generation of program test data," *Software Engineering, IEEE Transactions on*, no. 4, pp. 293–300, 1976.

[7] N. R. Lyons, "An automatic data generating system for data base simulation and testing," *ACM SIGSIM Simulation Digest*, vol. 8, no. 4, pp. 8–11, 1977.

[8] E. Miller Jr and R. Melton, "Automated generation of testcase datasets," in *ACM SIGPLAN Notices*, vol. 10, pp. 51–58, ACM, 1975.

[9] D. L. Bird and C. U. Munoz, “Automatic generation of random self-checking test cases,” *IBM systems journal*, vol. 22, no. 3, pp. 229–245, 1983.

[10] H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, pp. 366–427, 1997.

[11] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* Wiley, 2011.

[12] S. Kuppuraj and S. Priya, “Search based optimization for test data generation using genetic algorithms,” pp. 201–205, 2012.

[13] M. Grindal, J. Offutt, and J. Mellin, “On the testing maturity of software producing organizations,” in *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pp. 171–180, IEEE, 2006.

[14] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming.* Academic Press Ltd., 1972.

[15] C. Kaner, “Exploratory testing,” in *Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL*, 2006.

[16] N. Chauhan, *Software Testing - Principles and Practices.* Oxford University Press, 2011.

[17] J. C. Miller and C. J. Maloney, “Systematic mistake analysis of digital computer programs,” *Communications of the ACM*, vol. 6, no. 2, pp. 58–63, 1963.

[18] C. Indumathi, B. Galeebathullah, and O. Pandithurai, “Analysis of test case coverage using data mining technique,” in *Communication Control and Computing Technologies (ICCCCT), 2010 IEEE International Conference on*, pp. 806–809, IEEE, 2010.

[19] M. C. Yang and A. Chao, “Reliability-estimation and stopping-rules for software testing, based on repeated appearances of bugs,” *Reliability, IEEE Transactions on*, vol. 44, no. 2, pp. 315–321, 1995.

[20] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *Software Engineering, IEEE Transactions on*, vol. 29, no. 3, pp. 195–209, 2003.

[21] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[22] R. Mall, *Fundamentals Of Software Engineering 3Rd Ed.* PHI Learning Pvt. Ltd., 2009.

[23] S. Parnami, K. Sharma, and S. V. Chande, "A survey on generation of test cases and test data using artificial intelligence techniques," 2008.

[24] J. Holand, "Adaptation in nature and artificial systems," 1992.

[25] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, pp. 1942–1948, IEEE, 1995.

[26] J. J. Liang, A. Qin, P. N. Suganthan, and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," *Evolutionary Computation, IEEE Transactions on*, vol. 10, no. 3, pp. 281–295, 2006.

[27] L. N. De Castro and F. J. Von Zuben, "Learning and optimization using the clonal selection principle," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 3, pp. 239–251, 2002.

[28] Z. Bayraktar, J. A. Bossard, X. Wang, and D. H. Werner, "A real-valued parallel clonal selection algorithm and its application to the design optimization of multi-layered frequency selective surfaces," *Antennas and Propagation, IEEE Transactions on*, vol. 60, no. 4, pp. 1831–1843, 2012.

[29] P. R. Srivastava and T.-h. Kim, "Application of genetic algorithm in software testing," *International Journal of software Engineering and its Applications*, vol. 3, no. 4, pp. 87–96, 2009.

[30] J.-C. Lin and P.-L. Yeh, "Automatic test data generation for path testing using gas," *Information Sciences*, vol. 131, no. 1, pp. 47–64, 2001.

[31] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[32] M. A. Ahmed and I. Hermadi, "Ga-based multiple paths test data generator," *Computers & Operations Research*, vol. 35, no. 10, pp. 3107–3124, 2008.

[33] C. Mao, X. Yu, and J. Chen, "Swarm intelligence-based test data generation for structural testing," in *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on*, pp. 623–628, IEEE, 2012.

[34] C. Huanhuan, C. Li, Z. Bian, and K. Halei, "An efficient automated test data generation method," in *Measuring Technology and Mechatronics Automation (ICMTMA), 2010 International Conference on*, vol. 1, pp. 453–456, IEEE, 2010.

[35] S. Zhang, Y. Zhang, H. Zhou, and Q. He, "Automatic path test data generation based on ga-pso," in *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, vol. 1, pp. 142–146, IEEE, 2010.

[36] R. Ding, X. Feng, S. Li, and H. Dong, "Automatic generation of software test data based on hybrid particle swarm genetic algorithm," in *Electrical & Electronics Engineering (EEESYM), 2012 IEEE Symposium on*, pp. 670–673, IEEE, 2012.

[37] X. Xu, Y. Chen, X. Li, and D. Guo, "A path-oriented test data generation approach for automatic software testing," in *Anti-counterfeiting, Security and Identification, 2008. ASID 2008. 2nd International Conference on*, pp. 63–66, IEEE, 2008.

[38] A. Pachauri *et al.*, "Use of clonal selection algorithm as software test data generation technique," in *Advanced Computing & Communication Technologies (ACCT), 2012 Second International Conference on*, pp. 1–5, IEEE, 2012.

[39] B. Smith and L. A. Williams, “A survey on code coverage as a stopping criterion for unit testing,” 2008.

[40] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan, “A survey on automatic test case generation,” *Academic Open Internet Journal*, vol. 15, pp. 1–5, 2005.

[41] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, “Test coverage and post-verification defects: A multiple case study,” in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pp. 291–301, IEEE, 2009.

[42] Y. Wei, M. Oriol, and B. Meyer, “Is coverage a good measure of testing effectiveness,” *month*, vol. 6, no. 674, p. 10, 2010.

[43] A. Derezińska, “Experiences from an empirical study of programs code coverage,” in *Advances in computer and information sciences and engineering*, pp. 57–62, Springer, 2008.

[44] J.-R. Chang and C.-Y. Huang, “A study of enhanced mc/dc coverage criterion for software testing,” in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1, pp. 457–464, IEEE, 2007.

[45] S. Berner, R. Weber, and R. K. Keller, “Enhancing software testing by judicious use of code coverage information,” in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 612–620, IEEE, 2007.

[46] E. L. Lloyd and B. A. Malloy, “A study of test coverage adequacy in the presence of stubs,” *Journal of Object Technology*, vol. 4, no. 5, pp. 117–137, 2005.

[47] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel, “How well do professional developers test with code coverage visualizations? an empirical study,” in *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pp. 53–60, IEEE, 2005.

[48] M. Blaha and J. Rumbaugh, *Object-oriented modeling and design with UML*. Pearson Education Upper Saddle River, 2005.

Dissemination of Work

Accepted

1. Code Coverage Analysis for Basis Paths using Soft Computing Techniques, In *International Journal of Bioinformatics & Intelligent Control*, Volume 2, Issue No. 1, March 2013 (In Press: To be published).