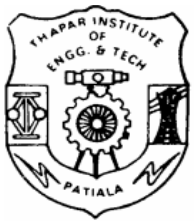
Automatic Generation of Software Test Cases using Genetic Algorithms

A thesis
submitted in partial fulfillment of the requirement for
the award of degree
of

Master of Engineering
In
Software Engineering



Under the Supervision of
Sh. R. S. Salaria
Assistant Professor
Computer Science & Engineering Department
Thapar Institute of Engineering & Technology, Patiala

Submitted By
Harsimran Singh
(8023104)

Computer Science & Engineering Department
Thapar Institute of Engineering & Technology (Deemed University)
Patiala-147004 (India)

May 2004

Declaration

I hereby certify that the work which is being presented in the thesis entitled, "***Automatic Generation of Software Test Cases using Genetic Algorithms***", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Mr. R. S. Salaria.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other University.

Harsimran Singh

This is to certify that the above statement made by the candidate is correct and true to best of my knowledge.

Mr. R. S. Salaria
Assistant Professor
Computer Science and Engineering Department
Thapar Institute of Engineering and Technology, Patiala

Countersigned by

eema Bawa)
Assistant Professor & Head,
Computer Sc. & Engg. Department,
Thapar Institute of Engg. & Technology,
Patiala.

(Dr. D.S. Bawa)
Dean (Academic Affairs)
Thapar Institute of Engg. & Technology,
Patiala.

Acknowledgement

A journey is easier when you travel together. Interdependence is certainly more valuable than independence. This thesis is the result of work carried out during the final year of my course whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

No amount of words can adequately express the debt, I owe to Mr. R. S. Salaria, Assistant Professor, Computer Science & Engineering Department, for his kind support, motivation and inspiration that triggered me for the thesis work. I owe him lots of gratitude for having me shown this way of research. He could not even realize how much I have learned from him.

I wish to express my gratitude to Ms. Seema Bawa, Assistant Professor and Head, Computer Science & Engineering Department, and Mr. Rajesh Bhatia, P.G. Coordinator, for their excellent guidance and encouragement right from beginning of this course. I am also thankful to all the faculty and staff members of the Computer Science & Engineering Department for providing me all the facilities required for the completion of this work.

No thesis could be written without being influenced by the thoughts of others. I would like to thank my colleague Navjyoti Pandhi and Surinder Pal Singh who were always there at the hour of the need and provided with all the help and support, which I needed.

Most importantly, I would like to give God the glory for all of the efforts I have put into this report.

Harsimran Singh
Roll no. 8023104

Abstract

Today, testing is the most challenging and dominating activity used by industry, therefore, improvement in its effectiveness, both with respect to the time and resources, is taken as a major factor by many researchers.

A new technique is presented for automatically generating test cases using genetic algorithms (GAs). This technique extends the random testing by the use of genetic algorithms

Various factors are discovered that distinguishes a Test Suit (Set of test cases) from the other one on the basis of its goodness. A *good test case* is a test case whose chances of finding a bug are more.

The factors discovered are used in evaluating the fitness function of GA for selecting the best possible Test Suit.

This technique takes the program as an input and then evaluates the test cases for that program. That is, it is a white box testing technique. Complete methodology and its effectiveness have been demonstrated with the help of suitable example.

Table of Contents

List of Figures

Chapter1
Introduction

1.1 Overview

Almost 50% of the software production development cost is expended in software testing. It consumes resources and adds nothing to the product in terms of functionality. Therefore, much effort has been spent in the development of automatic software testing tools in order to significantly reduce the cost of developing software. A test data generator is a tool, which supports and helps the program tester to produce test data for software.

Ideally, testing a software guarantees the absence of errors in the software, but in reality it only reveals the presence of software errors but never guarantees their absence. Even, systematic testing cannot prove absolutely the absence of errors, which are detected by discovering their effect. One objective of software testing is to find errors and program structure faults. However, a problem might be to decide when to stop testing the software, e.g. if no errors are found or, how long does one keep looking, if several errors are found.

Software testing is one of the main feasible methods to increase the confidence of the programmers in the correctness and reliability of software. Sometimes, programs that are poorly tested perform correctly for months and even years before some input sets reveal the presence of serious errors. Incorrect software that is released to market without being fully tested could result in customer dissatisfaction and moreover it is vitally important

for software in critical applications that it is free of software faults which might lead to heavy financial loss or even endanger lives. In the past decades, systematic approaches to software testing procedures and tools have been developed to avoid many difficulties that existed in ad-hoc techniques. Nevertheless, software testing is the most usual technique for error detection in today's software industry. The main goal of software testing is to increase one's confidence in the correctness of the program being tested.

In order to test software, test data have to be generated and some test data are better at finding errors than others. Therefore, a systematic testing system has to differentiate good (suitable) test data from bad test (unsuitable) data, and so it should be able to detect good test data if they are generated. Nowadays testing tools can automatically generate test data that will satisfy certain criteria, such as branch testing, path testing, etc. However, these tools have problems, when complicated software is tested. A testing tool should be general, robust and generate the right test data corresponding to the testing criteria for use in the real world of software testing [5]. Therefore, a search algorithm of a tool must decide where the best values (test data) lie and concentrate its search there. It can be difficult to find correct test data because conditions or predicates in the software restrict the input domain that is a set of valid data.

Test data that are good for one program are not necessarily appropriate for another program even if they have the same functionality. Therefore, an adaptive testing tool for the software under test is necessary. Adaptive means that it monitors the effectiveness of the test data to the environment in order to produce new solutions with the attempt to maximise the test effectiveness.

There are number of test-data generation techniques that have been automated earlier. Generally the test data generators are broadly classified as follows:

- Random test-data generators [1, 8, 14, 25, 26] select random inputs for the test data from some distribution.
- Structured or path oriented test data generators [27, 28] typically use the program's control-flow graph, select a particular path, and use a technique such as symbolic evaluation [1, 8, 10, 16, 20, 22] (Appendix A) to generate test data for that path.
- Goal-oriented test data generators [23, 24] select input to execute the selected goal, such as a statement, irrespective of the path taken. Intelligent test-data generators often rely on sophisticated analysis of the code, to guide the search for new test data.

This research has extended the random testing technique by the use of Genetic Algorithms (GAs). The research introduces certain factors for a good test suit.

- Likelihood
- Close to Boundary Value
- Branch Coverage

These factors can be used as the fitness function of the GA to find the optimal solution i.e. the best set of test cases. These factors evaluate the goodness of a test suit (set of test

cases). A good test case has more chances of finding a bug in a program. More the value of factor of any test suit more the chances of finding the bugs of the program and more the chances of its selection for execution.

1.2 Objectives and aims of the Research

The overall aim of this research is to investigate the effectiveness of Genetic Algorithms (GAs) with regard to random testing and to automatically generate best set of test cases of any software module.

The objectives of the research activity can be defined as follows:

- The furtherance of basic knowledge required to develop new techniques for automatic testing;
- To evaluate factors that distinguishes one test case set (test suit) from the other according to their goodness;
- To assess the feasibility of using GAs to automatically generate test data for software testing;

Examples show the efficiency of GAs in automatically generating test data. It has been proven that GAs required less CPU time in reaching a global solution.

1.3 Structure of Thesis

Following this introductory chapter, Chapter 2 gives the brief introduction to software testing. Software testing process is explained in brief. Various software testing techniques are explained. The automation of software testing is described with all its advantages and disadvantages.

Chapter 3 describes the overall idea of GAs. An introduction to GAs is given and how and why they work is explained using an example. Various operators and procedures are explained which are used within a GA. Important and necessary issues of GAs are described.

Chapter 4 gives the factors that distinguish one test case set (test suit) from the other according to their goodness. How theses factors are evaluated mathematically for a particular test suit using an example is given.

Chapter 5 provides the complete description of the research work i.e. generation of test cases using GAs. Complete procedure of the generation of test cases is given. How the factors described can help in evaluating fitness function. The operators used by GAs are described. Complete methodology and its effectiveness are demonstrated with the help of suitable example.

Chapter 2
Introduction to Software Testing

2.1 Introduction

Testing [1, 2, 7] is the most critical phase in the software development life cycle. The testing phase is the final filter for all errors of omission and commission. Testing software is far more complex than exercising a program to see if it works. Each review, inspection, audit, walk-through, group code read, all is in reality a form of test. The more effective that can make early is static testing, the fewer problems will encounter in the dynamic stages of testing. IT has shown again and again that the earlier a fault can be detected and removed, the lower the additional development cost associated with removing the error. Preparation for testing should begin as soon as each software product is defined.

The increasing visibility of software as the system element and the attendant “costs” associated with a software failure are motivating forces for well-planned, thorough testing. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software can cost three to five times as much as all other software engineering activities combined.

2.2 Objective

The main objective of testing is to prove that the software product as a minimum meets a set of pre-established acceptance criteria under a prescribed set of environmental circumstances. There are two components to 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 [1]. Correctness means that function, performance, and timing requirements match acceptance criteria.

Software testing is further complicated by the fact that system acceptance criteria usually involve hardware, procedures, and operators so that acceptance tests involve more than just the software. Software tests are designed to force failures. In that regard, software testing is intrinsically destructive.

Following are the objectives that software testing follows:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet undiscovered error.

Our objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort. If testing is conducted successfully it

will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to the specifications and that performance requirements appear to have been met. In addition, data collected as testing is conducted provides a good indication of software quality as a whole. But there is one thing that testing cannot do:

Testing cannot show the absence of defects, it can only show that software errors are present.

2.3 Testing Principles

Following are principles of Software Testing [2]:

- *All tests should be traceable to customer requirements*. The objective of system testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.

- *Tests should be planned long before testing begins*. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

- *Testing should begin "in the small" and progress toward testing "in the large"*. The first test planned and executed generally focus on individual program modules. As testing progresses, testing shifts focus in an attempt to find errors in integrated clusters of modules and ultimately in the entire system.

- *Exhaustive testing is not possible.* The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.

- *To be most effective, testing should be conducted by an independent third party.* By "most effective ", means testing that has the highest probability of finding errors. For this reason, the software engineer who created the system is not the best person to conduct all tests for the software.

2.4 Characteristics Of a "Good" Test

Following are the characteristics of a good test [2]:

- *A good test has a high probability of finding an error.* To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

- *A good test is not redundant*. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.

- *A good test should be "best of breed"*. In a group of tests that have a similar intent, time and resource limitations may militate for the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

- *A good test should be neither too simple nor too complex*. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

2.5 Testing Process

The IEEE829 standard [6] describes a framework within which the entire testing process can be managed. The framework allows easy communication between members of a testing project, organises the testing process, and outlines the documents that should be made part of any compliant testing process.

Figure 2.1 shows all the activities of software testing according to IEEE829.

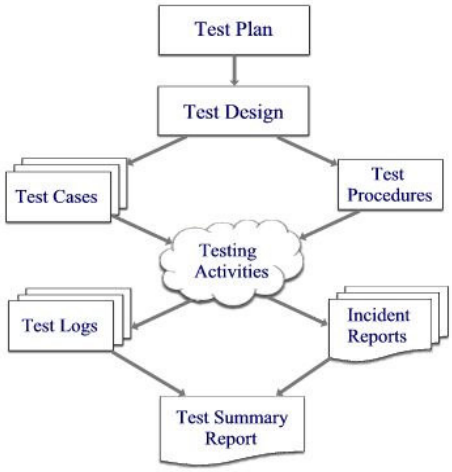


Figure 2.1: Testing activities

2.5.1 Test Plan

Test plan describes the scope, approach, resources, and schedule of testing activities in a given project, and identifies the items to be tested, the features of those items to be tested, the individual testing tasks that are to be performed, and personnel responsible for those tasks, along with the risks associated with the plan.

Test plan should have the following structure:

- Test plan identifier
- Introduction
- Items to be tested
- Features to be tested
- Features not to be tested
- Testing approach
- Test item pass/fail criteria
- Test suspension criteria
- Test resumption requirements
- Testing tasks
- Environmental needs
- Responsibilities

- Staffing and training requirements
- Schedule
- Risks and contingencies
- Approvals

2.5.2 Test Design

Test design, or test specification as it is sometimes known, further refines the testing approach, and identifies the test cases and test procedures, and test item pass/fail criteria.

Test design specification should have the following structure:

- Test design identifier
- Features to be tested
- Approach refinements
- Test identification
- Pass/fail criteria

2.5.3 Test Cases

Individual test cases document the values that will be used as input to individual tests, together with the associated expected outputs. A test case identifies any constraints on the test procedure resulting from the use of the test case, and separated from the test design to allow easy reuse in other situations.

Test cases should have the following structure:

- Test case identifier
- Items to be tested
- Input specifications
- Expected output specifications
- Environmental prerequisites
- Special procedural requirements
- Inter-case dependencies

2.5.4 Test Procedures

Test procedure describes the exact steps required to operate the system and execute test cases in order to implement the test design. Test procedure is kept separate from the test design as it is followed step by step, and does not contain irrelevant detail.

Test procedure should have the following structure:

- Test procedure identifier
- Purpose
- Special requirements
- Procedure steps

2.5.5 Test Logs

Test logs are used to record what occurred during execution of a test or set of tests. Test logs may either be manually created as tests are executed, or automatically by the system as testing processes.

Test logs should have the following structure:

- Test log identifier
- Description
- Activity and event entries

2.5.6 Incident Reports

Incident reports are used to provide a description of any events that occur during testing that require further investigation.

Incident reports should have the following structure:

- Incident report identifier
- Summary
- Incident description
- Impact of the incident

2.5.7 Test Summary Report

Test summary report summarises the testing activities associated with one or more test designs.

Test summary report should have the following structure:

- Test summary report identifier
- Summary
- Variances
- Comprehensive assessment
- Summary of results
- Evaluation

- Summary of activities
- Approvals

2.6 Software Testing Approaches

2.6.1 Top-Down Approach

The top-down approach is based on establishing the top-level control structure first. In top-down strategy, testing starts from the top of the hierarchy, and then incrementally adds modules that it calls and tests the new combined system. This approach of testing requires stubs to be written. A **stub** is a dummy routine that simulates a module [15]. In the top-down approach, a module (or a collection) cannot be tested in isolation because they invoke some other modules. To allow the modules to be tested before their subordinates have been coded, stubs simulate the behaviour of the subordinates.

2.6.2 Bottom-up Approach

The bottom-up approach starts from the bottom of the hierarchy. First the modules at the very bottom, which have no subordinates, are tested. Then these modules are combined with higher-level modules for testing. At any stage all the subordinate modules exist and have been tested earlier. To perform bottom-up testing, drivers are needed to set up the appropriate environment and invoke the module. It is the job of the **driver** to invoke the module under testing with the different set of test cases.

2.7 Software Testing Techniques

2.7.1 Static Testing

The term *static testing* refers to testing the software requirement specification (SRS), software design specification (SDS) and other non-executable items through requirement analysis, audits, desk checks, inspections, walk-through, etc [1]. Static testing is employed to verify the correctness of requirements, designs, and code before execution of test cases. Static testing will also audit newly developed or reused code for adherence to established standards. A successful static test of a software module depends upon several things going right:

1. A correct allocation of requirements to the software components.
2. A correct partitioning and sub allocation of software requirements to the module.
3. Successful (correct) module design.
4. A successful translation of the intermediate code (pseudo-code, POL, etc.) into programming language statements.

However, true representative test cases must be successfully executed before the testing and integration team certifies a software module. Usually this step is not a part of static testing.

The purposes of the static testing are:

- Validating the requirement specifications.

- Looking for omissions, inconsistencies, redundancies in all documents and source code.
- To ensure that the documents of design and coding conform to the specification.

2.7.2 Dynamic Testing

Dynamic testing is a term that describes the development of test cases and test procedures, the execution of test cases, and the structure and use of test logs and anomaly or incident reports. There are two popular ways to perform dynamic testing, namely, black box testing and white (glass) box testing. Either of these two methods requires a set of well-developed and well-structured test cases.

Dynamic testing cannot prove the absolute correctness of a software product unless it is performed in an exhaustive manner [17]. An exhaustive test requires a set of test cases that guarantees the following: explicitly exercises every possible, module path and every possible combination of paths with every possible module input and every possible combination of module inputs.

2.8 Types of Testing

2.8.1 Black Box or Functional Testing

Black box testing is the testing of a piece of software without regard to its underlying implementation. Specifically, it dictates that test cases for a piece of software are to be generated based solely on an examination of the specification (external description) for that piece of software. The goal of black box testing is to demonstrate that the software being tested does not adhere to its external specification.

The objective is to search for interface errors, function or process errors, performance shortcomings, start-up/shutdown errors, and errors in local (module) databases by selecting appropriate inputs and external conditions and monitoring outputs.

Black box testing attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Performance errors
- Initialization and termination errors

Black Box Testing Techniques includes:

- Equivalence Partitioning
- Boundary Value Analysis
- Comparison Testing

2.8.2 White Box Testing

White box testing also known as *glass box testing* .It is a test case design method that uses the control structure of the procedural design to derive test cases [18]. Using white box testing methods, the software engineer can derive test cases that

1. guarantee that all *independent paths* within a module have been exercised at least once.
2. exercise all logical decisions on their true and false sides.
3. execute all loops at their boundaries and within their operational bounds.
4. exercise internal data structures to assure their validity.

Thus white box testing is the testing of the underlying implementation of a piece of software (e.g., source code) without regard to the specification (external description) for that piece of software. The goal of white box testing of source code is to identify such items as (unintentional) infinite loops, paths through the code which should be allowed, but which cannot be executed and dead (unreachable) code.

White Box Testing Techniques includes:

- **Control flow testing:** Testing on the basis of the flow of control of a program. It is of the following types:

- *Statement:* each statement executed at least once
- *Branch:* each branch traversed (and every entry point taken) at least once
- *Condition:* each condition True at least once and False at least once.
- *Branch/Condition:* both Branch and Condition coverage achieved.
- *Multiple Conditions:* Multiple Conditions coverage technique states that test cases must be written such that all possible combinations of conditions in each decision are taken at least once.
- *Loop:* Loop coverage technique states that test cases must be written to test the loop counters.

- **Data flow testing:** Testing on the basis of the flow of control of a program. It is of the following types:
  - *All Definition-Uses:* It requires that every definition of every variable to every use of that definition be exercised under the test.
  - *All Uses:* In this test set include at least one path segment from every definition of every variable to every use of that definition
  - *All p-uses/some c-uses:* In All p-uses/some c-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are definitions of variables that are not covered by the above prescription then add computational use test cases are required to cover every definition.

- *All c-uses/ some p-uses:* In All c-uses/some p-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above prescription then add predicate use test cases are required to cover every definition.

- *All definitions:* In this, test set includes every definition of every variable be covered by at least one use of that variable, be that use the computational use or predicate use.

- *All p-uses:* In All p-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition; if there are definitions of variables that are not covered by the above prescription then leave them.

- *All c-uses:* In All c-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition; if there are definitions of variables that are not covered by the above prescription then, leave them.

2.9 Automation of Software Testing

Software testing is very labour-intensive and expensive; it accounts for approximately 50% of the cost of a software system development. If the testing process could be automated, the cost of developing software could be reduced significantly.

The most critical component of the testing is the generation of test cases. To first order of approximation, this is a completely manual exercise and a prime candidate for savings through automation. However, the technology for automation has not been advancing as rapidly as one would have hoped. While there are automated test generation tools they often produce too large a test set, defeating the gains from automation. On the other, there do exist a few techniques and tools that have been recognized as good methods for automatically generating test cases. The practice needs to understand which of these methods are successful and in what environments they are viable. There is a reasonable amount of learning in the use of these tools or methodologies but they do pay off past the initial ramp up.

Test automation is the use of test tools to robotize the exercising of business and system transactions and requirements to verify application and architecture correctness and scalability / performance [9].

Most automation testing tools have editors, compilers and fully functional programming languages, i.e. C, Basic, Java, or JavaScript languages.

The goal of automated test execution is to minimize the amount of manual work involved in test execution and gain higher coverage with a larger number of test cases. The automated test execution has a significant impact on both the tools set for test execution and also the way tests are designed. Integral to automated test environment is the test oracle that verifies current operation and logs failure with diagnosis information [3]. This is a best practice fairly well understood in some segments of software testing and not in

others. The best practice, therefore, needs to leverage what is known and then develop methods for areas where automation is not yet fully exploited.

Example:

- If a manual test costs Rs.X to run the first time, it will cost Rs.X to run every time thereafter.
- An automated test can cost 3 to 30 times Rs.X the first time, but will cost about NIL after that.

Faster and better testing is an important aim for many test organizations. Automation of the test process by means of the use of test tools is an important instrument in achieving this aim.

A *test tool* is an automated resource that offers support to one or more test activities, such as planning and control, specification, constructing initial test files, execution of tests, and assessment.

The use of the test tool must make it possible to achieve higher productivity levels and/or greater efficiency. This means that a test tool is only a useful resource if its use yields a result; it should not be an aim in itself to use a tool.

Automation within the test process can take place in many ways and generally has one or more of the following aims:

- fewer hours needed.

- shorter lead time.
- more test depth.
- greater flexibility in testing.
- more/faster insight in the status of the test process.
- better motivation for the testers.

2.9.1 Quality attributes of a test case

- How *effective* in detecting defects?
- How *exemplary?* (the more exemplary, the less test cases needed)
- How *economic?*
- How *evolvable?* (maintenance effort)

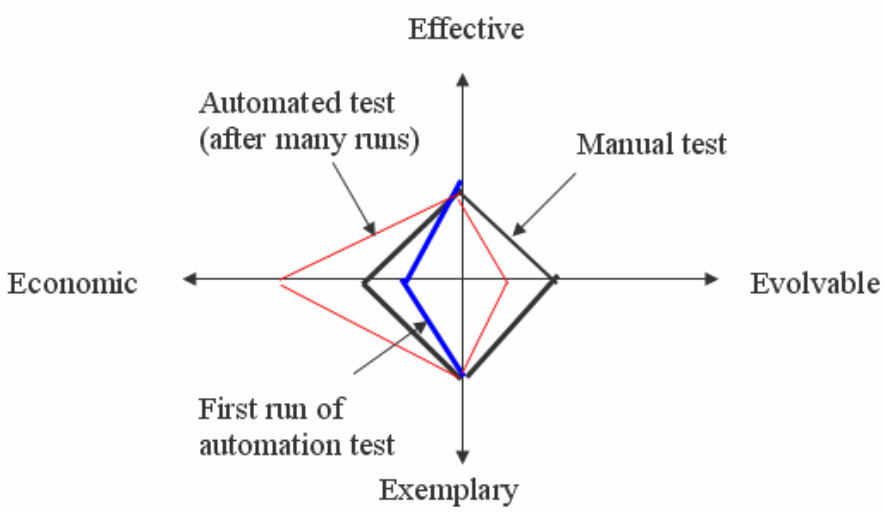


Figure 2.2: Quality Factors

2.9.2 What's wrong with Manual Testing?

The manual testing is not so effective because of the following reasons:

- It covers functionality only at human speed
- Major investment in process and not in errors detection
- Limited by resources availability and not necessity
- Inexact repetition of tests
- Inaccurate result checking
- Difficult regression testing means or NO regression testing

2.9.3 Need of Test Automation

- *Speed and Accuracy* – It's faster and more accurate than manual testing. It can be as much as 50 times faster, depending upon the speed of the driver machine and the speed of the application to process information (inserts, updates, deletes and views). Test tools are much more accurate than manual test inputs. The average typist makes 3 mistakes for every 1,000 keystrokes. Also, automation tools never tire, get bored, take shortcuts or make assumptions of what works.

- *Accessibility* – Automation tools allow access to objects, data, communication protocols, and operating systems that manual testers cannot access. This allows for a test suite with much greater depth and breadth.

- *Accumulation* – Once tests are developed, long-term benefits are derived through reuse. Applications change and gain complexity over time. The number of tests are always increasing as the application/architecture matures. Engineers can constantly add onto test suite and not have to test the same functionality over and over again.

- *Manageability* – Ability to manage artifacts through automation tools.

- *Discovery of issues* – Automated testing assists with the discovery of issues early in the development process, reducing costs.

- *Repeatability* – An automation suite provides a repeatable process for verifying functionality on the functional side and scalability on the performance side.

- *Availability* – Scripts can run any time during the day or night unattended.

- *Formal process* – Automation forces a more formal process on test teams, due to the nature of the explicitness of the artifacts and the flow of information that is needed.

- *Retention of customers* – When sites do not function correctly or perform poorly, customers may leave and never come back. What is the cost to your business of that scenario? Performing correct and systematic automated testing helps assure a quality experience for the customer – both internal and external.

- *Greater job satisfaction for Testers* – The Test Engineers no longer manually execute the same test cases over and over.

2.9.4 What to Automate?

- *Test execution:* Run large numbers of test cases/suites without human intervention.

- *Test generation:* Produce test cases by processing the specification, code, or model.

- *Test management:* Log test cases & results; map tests to requirements & functionality; track test progress & completeness.

2.9.5 Where Test Automation?

- *Load/stress tests* - Very difficult to have very large numbers of human testers simultaneously accessing a system.
- *Regression test suites* - Tests maintained from previous releases; run to check that changes haven't caused faults.
- *Sanity tests* - Run after every new system build to check for obvious problems.
- *Stability tests* - Run the system for 24 hours to see that it can stay up.

2.9.6 Limitations of Testing Automation

- Can not replace manual testing
  - Tests that are run only seldom;
  - The software is very unstable;
  - Tests where the result is easily verified by a human;
  - Tests that includes physical interaction.
- Manual tests find more defects than automated tests
  - It was reported that: automated tests found 15% defects while manual testing found 85%.
- Greater reliance on the quality of the tests

- A tool can only identify differences between the actual and expected output.

- Test automation does not improve effectiveness
- Test automation may limit software development
  - Automated tests are more fragile than manual tests.
- Tools have no imagination
  - Not like human being, tools can only follow instructions.

Chapter 3
Genetic Algorithms

3.1 Introduction

Genetic algorithms [3, 4, 13, 17] are a part of **evolutionary computing**, which is a rapidly growing area of artificial intelligence. Genetic algorithms are inspired by Darwin's theory of evolution. Simply said, problems are solved by an evolutionary process resulting in a best (fittest) solution (survivor) - in other words, the solution is evolved.

Genetic algorithm is adaptive heuristic search method premised on the evolutionary ideas of natural selection and genetics. The basic concept of Genetic Algorithm (GA) is designed to simulate processes in natural systems necessary for evolution, specifically those that follow the principles of survival of the fittest. It is generally used in situations where the search space is relatively large and cannot be traversed efficiently by classical search methods. This is mostly the case with problems whose solution requires evaluation and equilibration of many apparently unrelated variables. As such they represent an intelligent exploitation of a random search space within a defined search space to solve a problem.

Genetic algorithms searching mechanism starts with a set of solutions called a population. One solution in the population is called a chromosome. The search is guided by a survival of the fittest principle. The search proceeds for a number of generations, for each generation the fitter solutions (based on the fitness function) will be selected to form

a new population. During the cycle, there are three main operators namely reproduction, crossover and mutation. The cycle will repeat for a number of generations until certain termination criteria are met. It could terminate after a fixed number of generations, after a chromosome with a certain high fitness value is located or after a certain simulation time.

3.2 History

I. Rechenberg introduced evolutionary computing in the 1960s in his work "***Evolution strategies***" (*Evolutionsstrategie* in original). Other researchers then developed his idea. **Genetic Algorithms** (GAs) were invented by **John Holland** and developed by him and his students and colleagues. This lead to Holland's book "*Adaptation in Natural and Artificial Systems*" published in 1975 [15].

In 1992 **John Koza** has used genetic algorithm to evolve programs to perform certain tasks. He called his method "**Genetic Programming**" (GP) [12]. LISP programs were used, because programs in this language can expressed in the form of a "parse tree", which is the object the GA works on.

3.3 Biological Background

All living organisms consist of cells. In each cell there is the same set of **chromosomes**. Chromosomes are strings of DNA and serve as a model for the whole organism. A chromosome consists of **genes**, blocks of DNA. Each gene encodes a particular protein. Basically, it can be said that each gene encodes a **trait**, for example colour of eyes.

Possible settings for a trait (e.g. blue, brown) are called **alleles**. Each gene has its own position in the chromosome. This position is called **locus**.

Complete set of genetic material (all chromosomes) is called **genome**. Particular set of genes in genome is called **genotype**. The genotype is with later development after birth base for the organism's **phenotype**, its physical and mental characteristics, such as eye colour, intelligence etc.

During reproduction, **recombination** (or **crossover**) first occurs. Genes from parents combine to form a whole new chromosome. The newly created offspring can then be mutated. **Mutation** means that the elements of DNA are a bit changed. These changes are mainly caused by errors in copying genes from parents.

The **fitness** of an organism is measured by success of the organism in its life (survival).

3.4 Basic Concepts

3.4.1 Search Space

If we are solving a problem, we are usually looking for some solution that will be the best among others. The space of all feasible solutions (the set of solutions among which the desired solution resides) is called **search space** (also state space). Each point in the search space represents one possible solution. Each possible solution can be "marked" by its value (or fitness) for the problem. With GA we look for the best solution among a number of possible solutions - represented by one point in the search space.

Looking for a solution is then equal to looking for some extreme value (minimum or maximum) in the search space. At times the search space may be well defined, but usually we know only a few points in the search space. In the process of using GA, the process of finding solutions generates other points (possible solutions) as evolution proceeds.

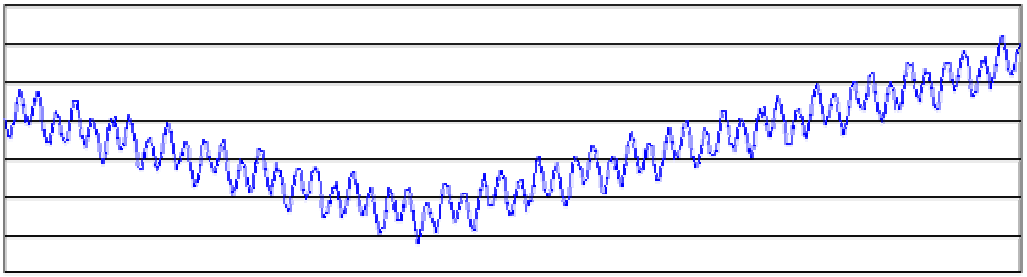


Figure 3.1: Example of a search space

The problem is that, the search can be very complicated. One may not know where to look for a solution or where to start. There are many methods one can use for finding a **suitable solution**, but these methods do not necessarily provide the **best solution**. Some of these methods are **hill climbing**, **tabu search**, **simulated annealing** and the **genetic algorithm**. The solutions found by these methods are often considered as good solutions, because it is not often possible to prove where the optimal solution lies.

3.4.2 NP-Hard Problems

One example of a class of problems that cannot be solved in the "traditional" ways, are NP problems. There are many tasks for which we may apply fast (polynomial) algorithms. There are also some problems that cannot be solved algorithmically.

There are many important problems in which, it is very difficult to find a solution, but once we have it, it is easy to check the solution. This fact led to **NP-complete problems**. NP stands for non-deterministic polynomial and it means that it is possible to "guess" the solution (by some non-deterministic algorithm) and then check it. If we had a guessing machine, we might be able to find a solution in some reasonable time.

Studying of NP-complete problems is, for simplicity, restricted to the problems where the answer can be yes or no. Because there are tasks with complicated outputs, a class of problems called **NP-hard** problems has been introduced. This class is not as limited as class of NP-complete problems.

A characteristic of NP-problems is that a simple algorithm, perhaps obvious at a first sight, can be used to find usable solutions. But this approach generally provides many possible solutions - just trying all possible solutions is very slow process (e.g. $O(2^n)$). For even slightly bigger instances of these types of problems this approach is not usable at all.

Today nobody knows if some faster algorithm exists to provide exact answers to NP-problems. The discovery of such algorithms remains a big task for researchers (maybe you! :-)). Today many people think that such algorithms do not exist and so they are looking for an alternative method. An example of an alternative method is the **genetic**

algorithm. Examples of the NP problems are satisfibility problem, travelling salesman problem or knapsack problem. Compendium of NP problems is available.

3.5 Description

GAs offer a robust non-linear search technique that is particularly suited to problems involving large numbers of variables. The GA achieves the optimum solution by the random exchange of information between increasingly fit samples and the introduction of a probability of independent random change. Compared to other search methods, there is a need for a strategy, which is global, efficient and robust over a broad spectrum of problems. The strength of GAs is derived from their ability to exploit in a highly efficient manner, information about a large number of individuals. This search method is modelled on natural selection by [15] whose motivation was to design and implement a robust adaptive system. GAs are being used to solve a variety of problems and are becoming an important tool in machine learning and function optimisation. Natural selection is used to produce adaptation.

GAs derive their name from the fact that they are loosely based on models of genetic change in a population of individuals, in order to effect a search mechanism with surprising power and speed. These algorithms apply genetically inspired operators to populations of potential solutions in an iterative fashion, creating new populations while searching for an optimum solution. The key word here is population. The fact that many points in the space are sampled in parallel shows that, genetic algorithm is a global optimisation technique. GAs do not make incremental changes to a single structure, but

maintain a population of structures from which new structures are created using genetic operators. The evolution is based on two primary operators: *mutation* and *crossover*.

The power of genetic algorithms is the technique of applying a recombination operator (crossover and mutation) to a population of individuals. Despite their randomised nature, GAs are not a simple random search. GAs take advantage of the old knowledge held in a parent population to generate new solutions with improved performance. Thereby, the population undergoes simulated evolution at each generation. Relatively good solutions reproduce; relatively bad ones die out and are replaced by fitter offspring.

An important characteristic of genetic algorithms is the fact that they are very effective when searching or optimising spaces those are not smooth or continuous. These are very difficult or impossible to search using calculus based methods, e.g. hill climbing. Genetic algorithms may be differentiated from more conventional techniques by the following characteristics:

1. A representation for the sample population must be derived;
2. GAs manipulate directly the encoded representation of variables, rather than manipulation of the variables themselves;
3. GAs use stochastic rather then deterministic operators;
4. GAs search blindly by sampling and ignoring all information except the outcome of the sample;

5. GAs search from a population of points rather than from a single point, thus reducing the probability of being stuck at a local optimum which make them suitable for parallel processing;

GAs are iterative procedures that produce new populations at each step. A new population is created from an existing population by means of performance evaluation, selection procedures, recombination and survival. These processes repeat themselves until the population locates an optimum solution or some other stopping condition is reached, e.g. number of generation or time. The initial population comprises a set of individuals generated randomly or heuristically. The selection of the starting generation has a significant effect on the performance of the next generation. In most GAs, individuals are represented by a fixed-length string over a finite alphabet. The binary alphabet is one of many possible ways of representing the individuals. GAs work directly with this representation and they are difficult to fool because they are not dependent upon continuity of the parameter space and existence of a derivative.

The process is similar to a natural population of biological creatures where successive generations are conceived, born and raised until they themselves are ready to reproduce. This population-by-population approach is very different from the more typical search methods of engineering optimisation. In many search methods, we move gingerly from a single point in the decision space to the next, using some decision rule to tell us how to get to the next point (hill climbing). This point-by-point method is dangerous because it often locates local peaks in the search space. GAs work from a population of points (individuals) simultaneously climbing many peaks in one generation (parallel), thus reducing the probability of finding a local optimum. To get a starting population, we can

generate a number of strings at random, or if we have some special prior knowledge of good regions of the decision space, we may plant seeds within the population to help things along. Regardless of the starting population, the operators of genetic algorithm have found members of high fitness quickly in many applications studied to date.

A string of a population can be considered as concatenated sub-strings (input variables) to represent a chromosome. The individual bits of the string represent genes. The length of a string is S and a population of strings contains a total of P_{SZ} strings**.**

Once the initial population has been created the evaluation phase begins. The GAs require that members of the population can be differentiated according to the string's fitness. A model of the system under test then processes the parameter combinations. The relative fitness of each combination is determined from the model's output. Fitness is defined as a non-negative function, which is to be maximised. An increase in the population average fitness is the overall effect of genetic algorithms.

The members that are fitter are given a higher probability of participating during the selection and reproduction phases and the others are more likely to be discarded. Fitness is measured by decoding a chromosome in the corresponding variables to an objective function (which is specific to the problem being solved). The value returned by the objective function is used to calculate a fitness value. The fitness is the only feedback facility that maintains sufficient selective differences between competing individuals in a population. The genetic algorithms are blind; they know nothing of the problem except the fitness information. In the selection phase, chromosomes of the population may be

chosen for the reproduction phase in several ways: for example, they may be chosen at random, or preference may be given to the fitter members.

During the reproduction phase, two members are chosen from the generation. The evolutionary process is then based on the genetic or recombination operators, crossover and mutation, which are applied to them to produce two new members (offspring) for the next generation.

The crossover operator couples the items of two parents (chromosomes) to generate two similar offspring, which are created by swapping corresponding sub strings of its parents. The idea of crossover is to create better individuals by combining genetic material (genes) of fitter parents. The mutation operator alters one or more genetic cells (genes) of a selected structure with a low probability. This ensures certain diversity in the genetic chromosomes over long periods of time and prevents stagnation near a local optimum. This means a complete new population of offspring is formed. A predefined survival strategy determines which of the parent and offspring survive. The whole process is repeated generation by generation until a global optimum is found or some other stopping condition is reached. One of the disadvantages can be the excessive number of iterations required and therefore the amount of computational time.

3.6 Basic steps of GA

Following steps gives the 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 new offspring (children). If no crossover was performed, offspring is the 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 the new population
4. **[Replace]** Use new generated population for a further run of the algorithm
5. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in current population
6. **[Loop]** Go to step **2**

3.7 Population and Generation

A population is like a list of several guesses (database). It consists of information about the individuals. As in nature, a population in GAs has several members in order to be a healthy population. If the population is too small inbreeding could produce unhealthy members. The population changes from one generation to next. The individuals in a population represent solutions. The advantage of using a population with many members is that many points in a space are searched in one generation. GAs are therefore highly suited for parallel processing. This sets the GAs apart from other search methods.

A generation is said to have premature by converged when the population of a generation has a uniform structure at all positions of the genes without reaching an optimal structure. This means that the GA or the generation has lost the power of searching and generating much better solutions.

Generally, small populations can quickly find good solutions because they require less computational effort (fewer evaluations per generation). However, premature convergence is generally obtained in a small population and the effect is that the population is often stuck on a local optimum. This can be compensated by an increased mutation probability, which would be similar to a random testing method. Larger populations are less likely to be caught by local optima, but generally take longer to find good solutions and an excessive amount of function evaluation per generation is required. A larger population allows the space to be sampled more thoroughly, resulting in more accurate statistics. Samples of small populations are not as thorough and produce results

with a higher variance. Therefore, a method must be set up in order to find a strategy, which will avoid converging towards a non-optimum solution.

3.8 Seeding

In order to start the optimisation method, the first population has to be generated. It can be seeded with a set of parameter values that can influence the search through the space. It can help to speed up the location of an optimum. Some systems have used parameter values from the previous experiments to provide a portion of the initial population. Alternatively values can be used which the user believes are in the right area of the search space to find an optimum faster. Normally the seeding is performed by random selection, which means that random data are generated.

3.9 Representations of Chromosomes (Encoding)

The representation of the chromosome can itself affect the performance of a GA-based function optimiser. There are different methods of representing a chromosome in the genetic algorithm, e.g. using binary, Gray, integer or floating data types. Some of the encoding types are discussed in this section.

3.9.1 Binary Encoding

The most common representation, invented by Holland [15], is the bit format. The variable values are encoded as bit strings, composed of characters copied from the binary alphabet {0, 1}. This kind of representation has turned out to be successful and was applied by Holland. He suggested that GAs are most effective when each gene takes on a small number of values and that binary genes are optimal for GA adaptive searching. A binary representation is not only convenient because of the GA's nature, but also for implementation by computers. In general, all bit strings within a population have the same format. A bit string format describes: how it is sub-divided into contiguously placed binary bit fields, see figure 3.2. A single chromosome is the solution to a problem, which can consist of a set of variables. If the problem consists of three input variables A, B and C, then the chromosome could look like the example in figure 3.2. This implies that all bit strings in the population have the same length. Each of these bit fields (A, B and C) represents an input variable value and its smallest unit is one bit that carries the information, Lucasius and Kateman [1993].

Chromosome														
A				B				C						
0	1	0	1	0	0	1	1	1	0	1	0	A = 5	B = 3	C = 9
0	0	1	1	0	0	1	0	1	0	1	0	A = 3	B = 2	C = 10
0	1	1	0	1	1	1	0	1	1	1	1	A = 6	B = 14	C = 15
0	0	0	0	0	0	0	1	1	0	0	0	A = 0	B = 1	C = 8

Figure 3.2: Binary Encoding

3.9.2 Permutation Encoding

Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem. In permutation encoding, every chromosome is a string of numbers that represent a position in a sequence. Figure 3.3 example of chromosomes with shows Permutation encoding.

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

Figure 3.3: Permutation Encoding

3.9.3 Value Encoding

Direct value encoding can be used in problems where some more complicated values such as real numbers are used. Use of binary encoding for this type of problems would be difficult. In the value encoding, every chromosome is a sequence of some values. Values can be anything connected to the problem, such as (real) numbers, chars or any objects. Figure 3.4 shows example of chromosomes with Value encoding.

Chromosome A	1.2324 5.3243 0.4556 2.3293 2.4545

Chromosome B	ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C	(back), (back), (right), (forward), (left)

Figure 3.4: Value Encoding

3.9.4 Tree Encoding

Tree encoding is used mainly for evolving programs or expressions, i.e. for genetic programming. In the tree encoding every chromosome is a tree of some objects, such as functions or commands in programming language. Figure 3.5 shows example of chromosomes with Tree encoding.

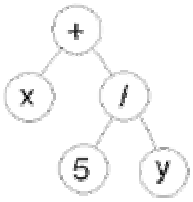
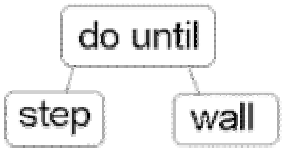
Chromosome A	Chromosome B
	
(+ x (/ 5 y))	(do_until step wall)

Figure 3.5: Tree Encoding

3.10 Selection

As already discussed, chromosomes are selected from the population to be parents for crossover. The problem is how to select these chromosomes. According to Darwin's theory of evolution the best ones survive to create new offspring. There are many methods in selecting the best chromosomes. Examples are roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and some others. Some of them are discussed in this section.

3.10.1 Roulette Wheel Selection

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a **roulette wheel** where all the chromosomes in the population are placed. The size of the section in the roulette wheel is proportional to the value of the fitness function of every chromosome - the bigger the value is, the larger the section is. See figure 3.6 for an example.

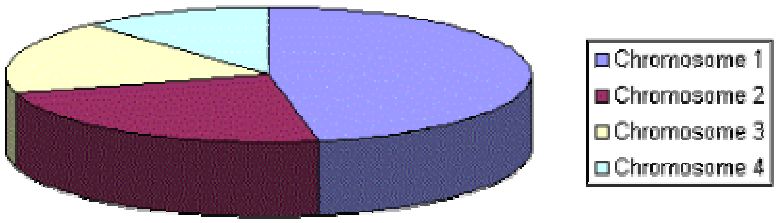


Figure 3.6: Roulette Wheel Selection

A marble is thrown in the roulette wheel and the chromosome where it stops is selected. Clearly, the chromosomes with bigger fitness value will be selected more times.

This process can be described by the following algorithm.

1. **[Sum]** Calculate the sum of all chromosome fitnesses in population - sum ***S***.
2. **[Select]** Generate random number from the interval ***(0,S)*** - ***r***.
3. **[Loop]** Go through the population and sum the fitnesses from ***0*** - sum ***s***. When the sum ***s*** is greater then ***r***, stop and return the chromosome where you are.

Of course, the step **1** is performed only once for each population.

3.10.2 Rank Selection

The previous type of selection will have problems when they are big differences between the fitness values. For example, if the best chromosome fitness is 90% of the sum of all fitnesses then the other chromosomes will have very few chances to be selected.

Rank selection ranks the population first and then every chromosome receives fitness value determined by this ranking. The worst will have the fitness ***1***, the second worst ***2*** etc. and the best will have fitness ***N*** (number of chromosomes in population).

See, how the situation changes from figure 3.7 to figure 3.8 after changing fitness to the numbers determined by the ranking.

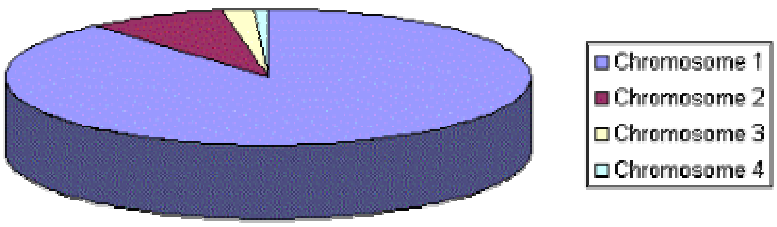


Figure 3.7: Situation before ranking (graph of fitnesses)

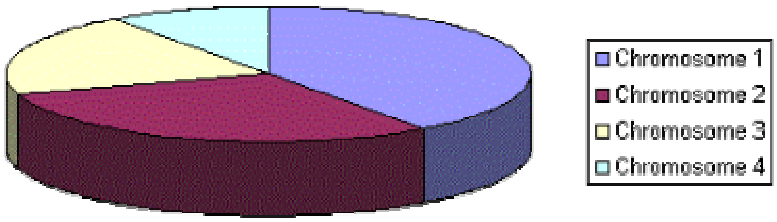


Figure 3.8: Situation after ranking (graph of order numbers)

Now all the chromosomes have a chance to be selected. However this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones.

3.10.3 Steady-State Selection

This is not a particular method of selecting parents. The main idea of this type of selecting to the new population is that a big part of chromosomes can survive to next generation.

The steady-state selection GA works in the following way. In every generation a few good (with higher fitness) chromosomes are selected for creating new offspring. Then some bad (with lower fitness) chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation.

3.10.4 Elitism

The idea of the elitism has been already introduced. When creating a new population by crossover and mutation, we have a big chance, that we will loose the best chromosome.

Elitism is the name of the method that first copies the best chromosome (or few best chromosomes) to the new population. The rest of the population is constructed in ways described above. Elitism can rapidly increase the performance of GA, because it prevents a loss of the best found solution.

3.11 Crossover and Mutation (Operators of GA)

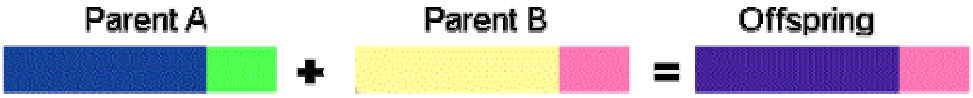
Crossover and mutation are two basic operators of GA. Performance of GA depends on them very much. The type and implementation of operators depends on the encoding and also on the problem.

There are many ways how to perform crossover and mutation. Some of them are explained in this section briefly accordingly with the various encoding schemes.

3.11.1 Binary Encoding

Crossover

Single point crossover - one crossover point is selected, binary string from the beginning of the chromosome to the crossover point is copied from the first parent, the rest is copied from the other parent



11001011 + 11011111 = 11001111

Figure 3.9: Single Point Crossover

Two point crossover - two crossover points are selected, binary string from the beginning of the chromosome to the first crossover point is copied from the first parent, the part from the first to the second crossover point is copied from the other parent and the rest is copied from the first parent again

11001011 + 11011111 = 11011111

Figure 3.10: Two Point Crossover

Uniform crossover - bits are randomly copied from the first or from the second parent

11001011 + 11011101 + 11011111

Figure 3.11: Uniform Crossover

Arithmetic crossover - some arithmetic operation is performed to make a new offspring

11001011 + 11011111 = 11001001 (AND)

Figure 3.12: Arithmetic Point Crossover

Mutation

Bit inversion - selected bits are inverted

11001001 => 10001001

Figure 3.13: Bit inversion mutation

3.11.2 Permutation Encoding

Crossover

Single point crossover - one crossover point is selected, the permutation is copied from the first parent till the crossover point, and then the other parent is scanned and if the number is not yet in the offspring, it is added

Note: there are more ways how to produce the rest after crossover point

For example:

(1 2 3 4 5 6 7 8 9) + (4 5 3 6 8 9 7 2 1) = (1 2 3 4 5 6 8 9 7)

Mutation

Order changing - two numbers are selected and exchanged

For example:

(1 2 3 4 5 6 8 9 7) => (1 8 3 4 5 6 2 9 7)

3.11.3 Value Encoding

Crossover

All crossovers from **binary encoding** can be used

Mutation

Adding a small number (for real value encoding) - a small number is added to (or subtracted from) selected values

For example:

(1.29 5.68 2.86 4.11 5.55) => (1.29 5.68 2.73 4.22 5.55)

3.11.4 Tree Encoding

Crossover

Tree crossover - one crossover point is selected in both parents, parents are divided in that point and the parts below crossover points are exchanged to produce new offspring

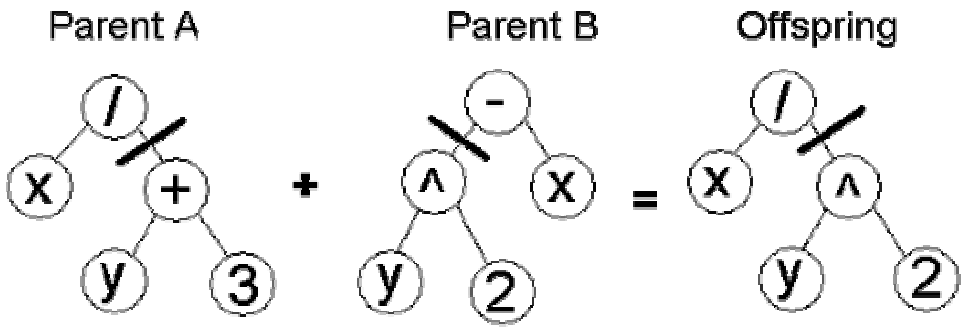


Figure 3.14: Tree Crossover

Mutation

Changing operator, number - selected nodes are changed

3.12 Parameters of GA

3.12.1 Crossover and Mutation Probability

There are two basic parameters of GA - crossover probability and mutation probability.

Crossover probability: how often crossover will be performed. If there is no crossover, offspring are exact copies of parents. If there is crossover, offspring are made from parts of both parent's chromosome. If crossover probability is **100%**, then all offspring are made by crossover. If it is **0%**, whole new generation is made from exact copies of

chromosomes from old population (but this does not mean that the new generation is the same!).

Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better. However, it is good to leave some part of old population survive to next generation.

Mutation probability: how often parts of chromosome will be mutated. If there is no mutation, offspring are generated immediately after crossover (or directly copied) without any change. If mutation is performed, one or more parts of a chromosome are changed. If mutation probability is **100%**, whole chromosome is changed, if it is **0%**, nothing is changed.

Mutation generally prevents the GA from falling into local extremes. Mutation should not occur very often, because then GA will in fact change to **random search**.

3.12.2 Other Parameters

There are also some other parameters of GA. One another particularly important parameter is population size.

Population size: how many chromosomes are in population (in one generation). If there are too few chromosomes, GA have few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many

chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to use very large populations because it does not solve the problem faster than moderate sized populations.

3.13 Performance

Although the goal is clearly to achieve an optimum solution, it is undesirable that the entire population should converge to the same value. This is because valuable information that is developed in part of the population is often lost. Therefore, the population needs to be diverse, so that a wide search of the input domain is guaranteed. If the population converges prematurely or too fast, the penalty will be to lose information that restricts the exploration of the search space. When a particular gene has at least 95% of all individuals the same value in a population, it is called as converged allele. Since the characteristics of the function that is to be optimised, are unknown, such a restriction may prevent the finding of the optimum solution. The metrics for the evaluation of the performance of genetic algorithms are given as follows:

1. The *on-line performance* is the average fitness value of all individuals that have been generated and evaluated. It penalises those search methods which have to test many poor solutions (structures) before generating a solution for the global optimum. In order to have a high on-line performance the GA has to concentrate on those areas in the search space where the best values lie.

2. The *off-line performance* is the average of the best individual fitness from each generation. This method will not penalise exploring poor regions of the search space which may lead to better solutions.

3. The *best individual* is the individual with the highest fitness that has been generated.

If the diversity is increased, the on-line performance measure will be affected quite badly. A restricted convergence impairs the off-line performance. A usual reason for fast convergence is the existence of a very fit (super) individual that dominates the population surviving many more times and hence reducing the variety of genetic material available. In a small number of generations the super individual and its descendants can dominate the entire population. To avoid this phenomenon Baker places a threshold on an individual's expected number of offspring.

3.14 Applications of GA

Genetic algorithms have been used for difficult problems (such as NP-hard problems), for machine learning and also for evolving simple programs. They have been also used for some art, for evolving pictures and music.

The advantage of GAs is in their parallelism. GA is travelling in a search space using more individuals (and with genotype rather than phenotype) so that they are less likely to get stuck in a local extreme like the other methods.

They are also easy to implement. Once you have the basic GA algorithm implemented, you have just to write a new chromosome (just one object) to solve another problem. With the same encoding you just change the fitness function - and you are done. However, for some problems, choosing and implementation of encoding and fitness function can be difficult.

The disadvantage of GAs is in the computational time. GAs can be slower than other methods. But sice we can terminate the computation in any time, the longer run is acceptable (especially with faster and faster computers).

To get an idea about some problems solved by GAs, here is a short list of some applications:

- Nonlinear dynamical systems - predicting, data analysis
- Designing neural networks, both architecture and weights
- Robot trajectory
- Evolving LISP programs (genetic programming)
- Strategy planning
- Finding shape of protein molecules
- TSP and sequence scheduling

- Functions for creating images

Chapter 4
Test Suit Factors

4.1 Overview

It is impossible to test software exhaustively (i.e. for each and every value possible). For the exhaustive testing of Program 1 (three variables, each *16* bit long) a Tester needs to execute $2^{16} * 2^{16} * 2^{16} = 2^{48}$ test cases, which is impossible for even a supercomputer to execute.

Therefore, to test the software, tester uses the selected cases only. The selection criteria of the earlier automated test tools are mainly branch coverage, statement coverage, etc. But there are problem with theses tools when complicated programs are there. They may leave very important test cases that are must to be tested.

So a new technique is proposed over here so that all the important test cases can be selected. Selection is done on the basis of various factors of a good test suit (set of test cases). GA helps in selecting the various test cases whose fitness function is based on these factors. Here the factors that contribute to selection of test cases are as:

- Likelihood
- Close to boundary value
- Branch coverage

These factors contribute to the selection of the test suit. Theses factors here are used as the evaluation of the test suit for its goodness. These factors can be used as the fitness function of the GA to find the optimal solution i.e. the best set of test cases. These factors evaluate the goodness of a test suit (set of test cases). A good test case has more chances of finding a bug in a program.

4.2 Basic Concepts and Notations

In this chapter the following code segment (Program 1) has been taken as an example:

```
        void triangle(int a int b, int c){
1,2,3:  if(a<=0 || b<=0 || c<=0)
4:      printf("\nwrong input");
        else
{
5:              if(a<b)
6:                      swap(a,b);
7:              if(a<c)
8:                      swap(a,c);
9:              if(b<c)
10:                     swap(b,c);
11:             if(a>=b+c)
12:                     printf("\nwrong input");
```

```
13:         else if(a==b)
        {
14:             if(b==c)
15:                 printf("\nEquilateral");
                else
16:                 printf("\nIsoslist");
        }
        else
        {
17:         if(b==c)
18:             printf("\nIsoslist");
            else
19:             printf("\nScalen");
            }
        }}
```

Program 1

Denoting the above code segment as *P*. Consider its any test suit *T* which is the set of test cases $t_1, t_2, t_3,\ldots$ Mathematically,

$T=\{t_1, t_2, t_3,\ldots\}$

A test case is a set of values of all input variables. In the above segment there are three input variables, namely, *a, b & c* declared as integer. So every test case of the above segment is a set of three integer values. Mathematically,

$t=\{i_1,i_2,i_3,\ldots\}$

where i_n is the value of the variable I_n ($n = 1,2,3,\ldots.$), but within its range. For example, in the above code all the three variables are of type integer (*2* bytes) i.e. their values range from *–32768* to *32767*. So its any test case is the set of three values, each ranging from *–32768* to *32767*.

4.3 Likelihood

The paths, which are more likely to be executed then others should be given higher priority for testing. *Likelihood* of any test suit is higher than the likelihood of any other test suit, if the test cases of the test suit follow the paths that are more likely to be executed. The likelihood factor will contribute to the selection of any particular test suite for execution. More the likelihood of a test suite higher is its probability to be selected for executions

Mathematically, the likelihood, *L* of a test suit, *T* is evaluated as follows:

$L(T) = 1-((1-L(P(t_1))) *(1-L(P(t_2)))* \ldots\ldots)$

where $P(t_i)$ is the path followed when the test case t_i is executed. For example, considering Program 1, test case *t={1,0,0}* follows the path *{1,2,4}*, i.e *P(t)={1,2,4}*. And *L(p)* is the likelihood of the path *p,* which is evaluated as follows:

1. Using symbolic evaluation method [1, 8, 10, 16, 20, 22] (Appendix A), evaluate the Boolean expression of that path. For example, for path *{1,2,4}*, Boolean expression is $a > 0 \wedge b \leq 0$.

2. From the complete input domain find the probability of that Boolean expression to be true. For example, the probability of the Boolean expression $a > 0 \wedge b \leq 0$ is $(32767/2^{16})*(32769/2^{16})*(2^{16}/2^{16}) = .25$ (approx.), as all the three variables has range from *–32768* to *32767*.

3. The probability above found gives the likelihood of that path.

4.4 Close to Boundary value

As the chances of bugs are mostly at the boundary values, so the test cases close to boundary values must be given higher priority than the other ones for testing. *Close to boundary value* is the factor that represents the 'how much the test case values are closer to the boundaries'. Consider the path *{1,2,4}* of Program 1, whose Boolean equation is $a > 0 \wedge b \leq 0$. So the boundary values for this path is $a=0, b=0$. Test cases with values of a and b closer to zero are given more priority than with values father than zero. Like likelihood, this factor also contributes to the selection of any particular test case. More the factor of any test suit, higher the probability of the test suit to be selected for execution.

Mathematically, Close to Boundary Value factor, B of a test suit, T is evaluated as follows:

$B(T) = B(t_1) * B(t_2) * \ldots\ldots$

where $B(t)$ is Close to boundary value factor of a test case t, which is evaluated as:

1. For the test case, evaluate the path that is followed, when the test case is executed. For example, for the above program test case, $t=\{1,0,0\}$ follows the path $\{1,2,4\}$, i.e $P(t)=\{1,2,4\}$.

2. Using symbolic evaluation method (Appendix A), evaluate the Boolean expression of that path. For example, for path $\{1,2,4\}$, Boolean expression is $a > 0 \wedge b \leq 0$.

3. From the Boolean expression evaluate the boundary value expressions by converting any comparison operators $\{<,>,\leq,\geq,=\}$ to '=' comparison operator. For example, for Boolean expression $a > 0 \wedge b \leq 0$, the boundary value expressions are $a=0, b=0$.

4. Change the boundary value expressions so that right side of the boundary value expressions becomes zero. For example, the expression, $a+b=c+d$ is converted into $a+b-(c+d)=0$.

5. Now the Close to boundary value factor of a test case is evaluated as

   $B(t)=B(b_1)*B(b_2)*\ldots$

   where $B(b_i)$ is the factor denoting 'how much the test case values are closer to the boundary value expression, b_i (which is of the form $[expression]=0$)'. For example,

$B(t=\{1,0,0\}) = B(a=0)*B(b=0)$

Now, B([exp.]=0) is evaluated as:

$1-(|e(t,[exp.])|/|f([exp.])|)$

where *e(t,[exp.])* is the actual value of expression when the variables are converted to the actual test case values and *f([exp])* is the farthermost value of the expression from zero in the complete input domain. For example, for the test case, *t={1,0,0}* the Close to Boundary Value factor is

$B(t)=B(a=0)*B(b=0)$

Now,

$B(a=0)=1-(|1|/|-32768|)=.99$

$B(b=0)=1-(|0|/|-32768|)=1$

So, the boundary value of the test case is

$B(t)=.99*1=.99$

4.5 Branch coverage

Most of the earlier automated test tools use the branch coverage criterion for selecting test cases. Branch coverage means the percentage of no of edges/branches of the control flowgraph covered by the test suit. *Control flowgraph* is graphical notation of the program that shows the flow of control of that program. The control flowgraph of

Program 1 is shown on figure 4.1. The Branch Coverage factor for any test suit, *T* here is denoted as *R(T)*. To evaluate this, firstly control flowgraph of the program is created. Then, for each test case of a test suit, evaluate the set of branches that has been covered. Take the union of all the evaluated set of branches. Count the number of elements of the resultant set (let it be n). At the last, Branch coverage is evaluated as:

$$R(T)=n/e$$

where *e* is the total number of edges of control flowgraph.

For example, considering Program 1 whose control flowgraph is shown in figure 4.1, and the test suit

T={

{0,5,8},

{3,1,9},

{8,5,6),

{5-3,-6}

}

The paths followed by each test case are:

P({0,5,8})={1,4}

P({3,1,9})={1,2,3,5,7,8,9,10,11,12}

P({8,5,6})={1,2,3,5,7,9,10,11,13,17,19}

P({5,-3,-6})={1,2,3,4}

Here out of twenty-nine branches/edges, the total number of covered branches is eighteen, so the branch coverage factor of the test suit, *T* is:

R(T)=18/29=0.62(approx.).

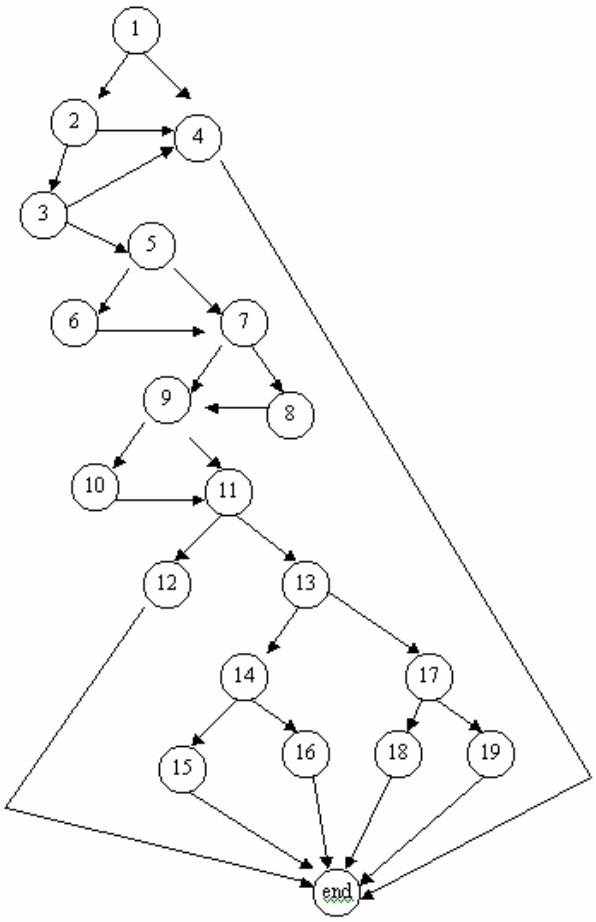


Figure 4.1: Control Flowgraph

Chapter 5
Generation of Test Cases using GA

5.1 Overview

Genetic algorithms (GAs) [3, 4, 12, 13, 15] represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's principal of the survival of the fittest. There is a randomized exchange of structured information among a population of artificial chromosomes. GAs are a computer model of biological evolution. When GAs are used to solve optimization problems, good results are obtained surprisingly quickly.

Here, Genetic algorithms is used to select software test cases. The general factors to be considered in genetic algorithms are:

- Population Initialization
- The Fitness Function
- Selection/Operations used

5.2 General Overview

The central objective of the GA used here is to select the best test suit (set of test cases) for the testing purpose. Here a test suit is taken as an individual (chromosome). A Population is the set of test suits. Figure 5.1 shows the complete structure.

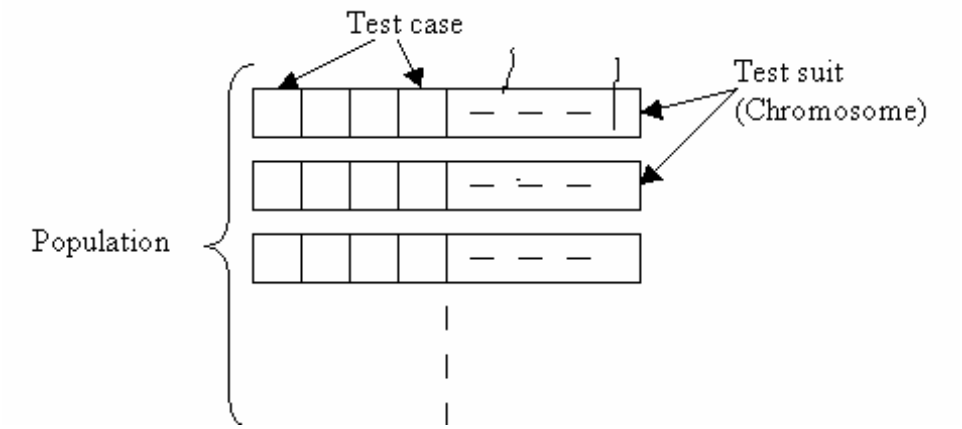


Figure 5.1: Model used in the research

The main thing that should always be taken into the mind is that any test suit cannot have any two same test cases at any time.

5.3 Population Initialization

As at the start, there is no information about the test cases, which are good and which are bad, that is why test cases are generated randomly i.e. population is initialized randomly.

Most of the automated test tools available now days select test cases based on certain criteria, like branch testing, path testing, etc. But these tools are not suitable for every kind of code, as some criteria cannot always find the good test data for every code

segment. This is the reason, why most of the companies are still using random test data generation tools.

That is why; here initialization of the population is done randomly.

5.4 The fitness function

The fitness function used for every individual is the average of three factors discussed above of a test suit i.e. Fitness function F of test suit, *T* is

$$F(T)=(L(T)+B(T)+R(T))/3$$

Here more the fitness function value, more likely the test suit to be selected. The priority of the test suit to be selected is directly proportional to the fitness function value.

As stated, the central objective is to select the best test suit as possible. As already explained the three factors, Likelihood, Close to Boundary Value, and Branch Coverage are the factors that decide whether the test suit is good or bad than others. So their average is taken as a fitness function that decide that whether the test suit is more suitable to be selected than other or not.

5.5 Selection

The selection of the parent chromosomes is done randomly both for crossover and mutation operations, as even a small change in input may change the path of execution. So we cannot say that good parent test cases always yield good child test cases.

5.6 Uniform Crossover

In uniform crossover, any two test suits (parents) are selected randomly. Interchange randomly some of the test cases of parents ignoring test cases that are common to both the test suits. The result is the two child test suits. As shown in figure 5.2, there are two common test cases (yellow and light green) that are ignored and there are three exchanges of test cases (Red & purple, blue & green, and white & grey).

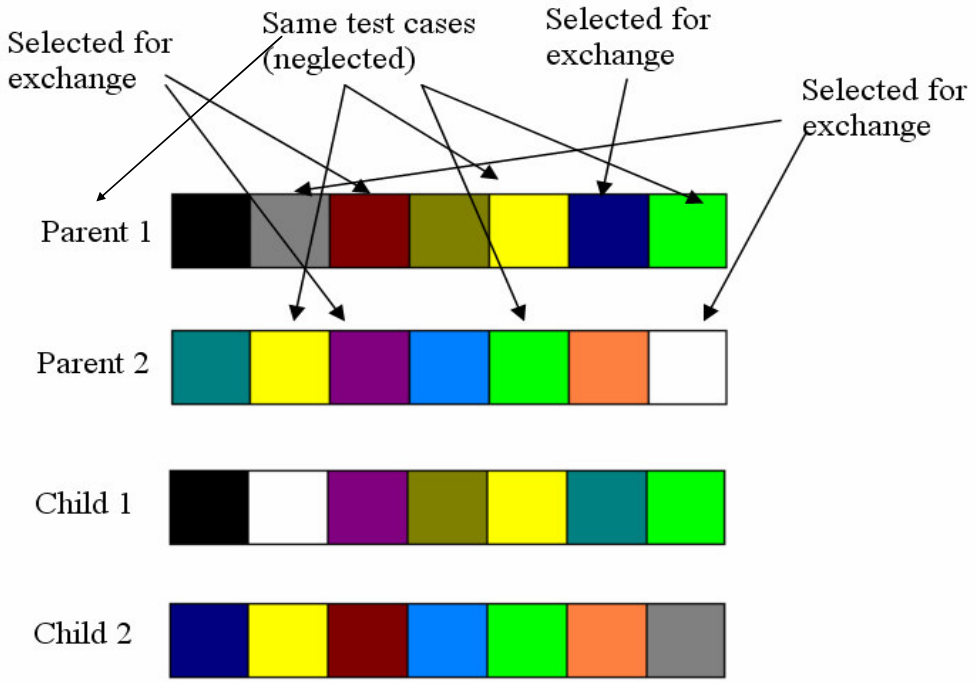


Figure 5.2: Uniform Crossover

5.7 Mutation

In mutation a test suit (parent) is selected randomly. Then randomly select some test cases of the parent test suit and replace them with the new test cases generated randomly that are not present in the parent test suit earlier and the new childe test suit gets created. As shown in figure 5.3, some of the test cases (green, purple and white) of the parent are replaced with newly generated test cases (blue, red and grey) to create a new child test suit.

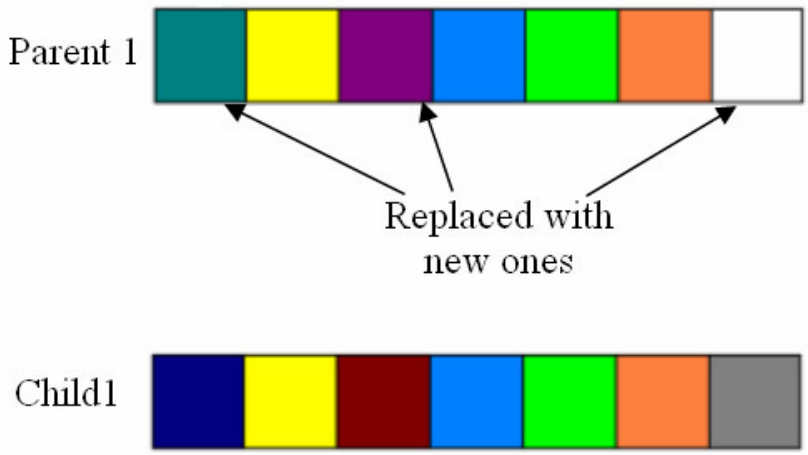


Figure 5.3: Mutation

5.8 An Example

Consider the code segment of Program 2 whose control flowgraph is shown in figure 5.4. This program takes three input parameters each of type integer i.e. their range is from -*32768* to *32767*.

Now at the start population of three test suits are initialised randomly as:

T1={{5, -15, 20}, {1, 165, 165}, {-1, -1, -1}}

T2={{1, 0, 1}, {160, 13, 0}, {3, 50, 10}}

T3={{-32768, 32767, 0}, {15, -5, 2000}, {2378, 17588, -20}}

```
        void greater(inr a, int b, int c){
1:      if(a>b)
        {
2:              if(a>c)
3:                      printf("Greater value is : %d", a);
                else
4:                      printf("Greater value is : %d", c);
        }
        else
        {
5:              if(b>c)
6:                      printf("Greater value is : %d", b);
                else
7:                      printf("Greater value is : %d", c);
        }}
```

Program 2

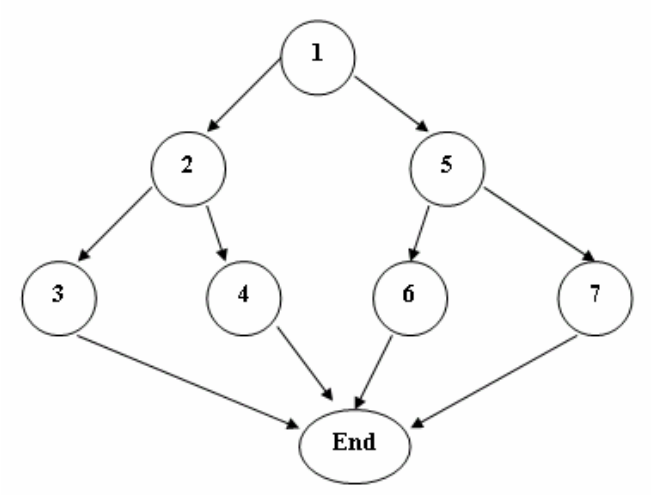


Figure 5.4: Control Flowgraph

Now Likelihood factor of *T1* is evaluated as:

*L(T1)= 1-((1-L(P({5, -15, 20}))) * (1-L(P({1, 165, 165}))) * (1-L(P({-1, -1, -1}))))*

or *L(T1)= 1-(1-L({1,2,4})) * (1-L({1,5,7})) * (1-(L({1,5,7})))*

or *L(T1)= 1-((1-0.25) * (1-0.25) * (1-0.25) = 1-(0.75*0.75*0.75) = 1-0.4219*

or *L(T1)=0.5781*

Similarly *L(T2)= 0.5781* and *L(T3)= 0.5781*.

Now Close to Boundary Value factor of *T1* is evaluated as:

*B(T1)= B({5, -15, 20}) * B({1, 165, 165}) * B({-1, -1, -1}))*

or *B(T1)=((1-20/65535)*(1-15/65535))*((1-164/65535)*(1-0/65535))*((1-0/65535)*(1-0/65535))*

or *B(T1)= 0.99969*0.99977*0.99749*1*1*1*

or *B(T1)= 0.9969*

Similarly *B(T2)= 0.9939* and *B(T3)= 0*.

Now Branch Coverage factor of *T1*is evaluated as:

Total no of edges of control flowgraph, *e=10;*

Edges covered by all the test cases, *n=6;*

R(T1)=n/e=6/10=0.6

Similarly *R(T2)= 0.8* and *R(T3)= 0.6*

No fitness of function of each is

F(T1)=(L(T1)+B(T1)+R(T1))/3=(0.5781+0.9969+0.6)/3=0.725

F(T2)=(L(T2)+B(T2)+R(T2))/3=(0.5781+0.9939+0.8)/3=0.7907

F(T3)=(L(T3)+B(T3)+R(T3))/3=(0.5781+0+0.6)/3=0.3927

Now crossover operator is used on two randomly selected test suits, say *T1* and *T2* and two new test suits are generated as

T4={{5, -15, 20}, {160, 13, 0}, {-1, -1, -1}}

T5={{1, 0, 1}, {1, 165, 165}, {3, 50, 10}}

Their fitness function is evaluated as:

F(T4)=(L(T4)+B(T4)+R(T4))/3=(0.5781+0.9948+0.8)/3=0.791

F(T5)=(L(T5)+B(T5)+R(T5))/3=(0.5781+0.9959+0.8)/3=0.7913

Applying mutation on *T4*, a new test suit is generated,

T6={{8, 8, 2}, {160, 13, 0}, {-1, -1, -1}}

Its fitness function is evaluated as:

F(T6)=(L(T6)+B(T6)+R(T6)/3=(0.5781+0.9952+0.8)/3=0.7911

Applying mutation on *T5*, a new test suit is generated,

T7={{1, 0, 1}, {50, -10, 50}, {3, 50, 10}}

Its fitness function is evaluated as:

F(T7)=(L(T7)+B(T7)+R(T7))/3=(0.5781+0.9978+0.8)/3=0.792

Here in all the test suits the fitness function of T7 is the greatest i.e. 0.792 so this test suit is selected for the execution

Hence, selected test suit is:

T7={{1, 0, 1}, {50, -10, 50}, {3, 50, 10}}

Chapter 6
Conclusion and Future work

6.1 Conclusion

Software testing is the integral, costly, and time consuming activity in the software development life cycle. As testing involves running the system being tested under a variety of configurations and circumstances, automation of testing offers a potential source of savings in complete life cycle.

As it is impossible to test the software exhaustively, so this technique is very useful in selecting the best set of test cases (test suit). The selection is based on three factors that evaluate the test case whether it is good or bad. Genetic algorithms are used for this purpose, GAs are found to be the best technique for selection purpose when there is very large population. The GAs give good results and their power lies in the good adaptation to the various and fast changing environments.

The primary objective of the research was to propose a GA-based software test data generator and to demonstrate its feasibility. The example shows how this can be achieved.

GAs show good results in searching the input domain for the required test sets. GAs may not be the answer to the approach of software testing, but do provide an effective strategy.

6.2 Future Work

As the fitness function is taken as the average of the three discovered factors. But the best results can only be achieved if the factors are assigned with different weights, according to the code segment, as some factor is good for selecting test cases for a particular code segment but some are not so good. So some expert system should be there that can judge the nature of the code segment and then assigns the weights to each factor, which gives the real fitness function value.

New operators, like crossover and mutation can also be defined so that GA can give more efficient results and the optimization process can become much easier and faster.

The one more important aspect of future work is of finding more factors that can compare two test suits for their goodness, so that efficiency of selection process can be improved.

References

[1] B. Beizer. *"Software Testing Techniques",* Van Nostrand Reinhold, 2nd edition, 1990.

[2] R. S. Pressman. *"Software Engineering: A Practitioner's Approach"*, 3rd Edition, McGraw Hill, New York (1992), p. 559.

[3] http://lancet.mit.edu/~mbwall/presentations/IntroToGAs/

[4] http://www.rennard.org/alife/english/gavintrgb.html

[5] R. Ferguson and B. Korel. *"The chaning approach for software test data generation"*. ACM Transactions on Software Engineering Methodology, 5(1):63–86, 1996.

[6] Standard for Software Test Documentation (IEEE829) http://ww.ieee.org.

[7] The Open Group, http://tetworks.opengroup.org/.

[8] R. Boyer, B. Elspas, and K. Levitt. *"SELECT-A formal system for testing and debugging programs by symbolic execution"*. SIGPLAN otices, vol. 10, no. 6, pp. 234-245. June 1975.

[9] C. Rankin. "The Software Testing Automation Framework"

[10] L. Clarke. *"A system to generate test data and symbolically execute programs"*. IEEE Trans. Sofrware Eng., vol. SE-2, no. 3, pp. 215- 222, Sept. 1976

[11] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller. *"SMOTL-A system to construct samples for data processing program debugging"*. IEEE Trans. Software Eng., vol. SE-5, no. 1, pp. 60-66,Jan. 1979.

[12] John Koza. *"Agntic Programming, Ann Arbor*". The University of Michigan Press, 1992.

[13] http://www.cs.qub.ac.uk/~M.Sullivan/ga/ga_index.html

[14] C. Ramamoorthy, S. Ho, and W. Chen. *"On the automated generation of program test data"*. IEEE Trans. Software Eng., vol. SE-2, no. 4. PD. 293-300, Dec. 1976.

[15] Holland J.H. *"Adaptation in natural and artificial system, Ann Arbor*". The University of Michigan Press, 1975.

[16] W. Howden. *"Symbolic testing and the DISSECT symbolic evaluation system"*. IEEE Trans. Software Eng., vol. SE-4, no. 4, pp. 266- 278. 1977. .

[17] Goldberg D. *"Genetic Algorithms"*. Addison Wesley, 1988.

[18] Emmeche C. *"Garden in the Machine. The Emerging Science of Artificial Life"*. Princeton University Press, 1994, pp. 114 ss.

[19] S. R. Ladd. "*Genetic Algorithm in C++*". 1999-2000. Downloadable book. http://www.coyotegulch.com.

[20] D. Ince. *"The automatic generation of test data"*. Comput. J., vol. 30,no. 1,pp. 63-69, 1987.

[21] *"Biological programming is not limited to AG"*, another well-known case is neural networks.

[22] W. Miller and D. Spooner. *"Automatic generation of floating-point test data"*. IEEE Trans. Software Eng., vol. SE-2, no. 3, pp. 223-226, Sept. 1976.

[23] D. L. Bird and C. U. Munoz. *"Automatic generation of random self-checking test cases"*. IBM Systems Journal, 23(3):228–245, 1983.

[24] J.W. Duran and S. Ntafos. *"An Evaluation of Random Testing"*. IEEE Transactions on Software Engineering, 10(4):438–444, July 1984.

[25] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. *"The Dynamic Domain Reduction Procedure for Test Data Generation"*. Software Practice and Experience, 29(2):167–193, January 1997.

[26] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. *"Automated test data generation using an iterative relaxation method*". In ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6), pages 231–244, Orlando, Florida, November 1998.

[27] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. *"Test-data generation using genetic algorithms. Software Testing"*. Verification & Reliability, 9(4):263–282, 1999.

Appendix A
Symbolic evaluation

A.1. Overview

The main idea behind symbolic execution is to use symbols instead of *numbers* as input values, and to represent the values of program variables with symbolic expressions. As a result, the output values computed by a program are expressed as function of the program input symbols.

In order to describe the state of a symbolically executed program the instruction counter and the set of pairs <variable, value> that usually defines the state of a *numeric execution*, is not sufficient.

As an example, let us consider the program 3, in which each statement is labelled for the sake of clarity.

```
L1: int example(int data) {
L2: int out;
L3: if (data > 0)
L4: out = data;
L5: else
L6: out = -data;
L7: return(out);
```

```
L8:  }
```

Program 3

Let the symbol α represents the value of the input parameter data. When executing the conditional statement, the value of data does not allow one to choose along which branch the computation has to continue. Hence, one has to assume either $\alpha > 0$ to execute the then branch or $\alpha \leq 0$, to execute the else branch. Such assumptions are represented by means of a first order predicate referred to as path condition (PC). Thus, the state of a symbolic execution is represented by a triple <IP, PC, V>, where

- IP is the instruction pointer referring to the statement to execute next;
- PC is the path condition;
- V is the set of pairs <variable, expression>.

Initially, IP points to the first statement, the input parameters are initialized using new symbols while other variables are initialized to the special value *Undef*. PC is set to *true*, that is no assumption is made on the values of variables.

Symbolic execution can be formalized by means of a function representing the execution of a single program statement. Let *Exec* denote such a function:

$$Exec: SymbolicState \times Program \rightarrow 2^{SymbolicState}$$

Exec takes as input a symbolic state and a program and returns the symbolic states[2] resulting from the execution of the current statement (i.e., the statement referred to by IP). The semantics of Exec can be informally described as follows:

- *Assignment statement*. The right hand side is evaluated. The resulting expression is then assigned to the left hand side variable in the output state. IP points to the next statement of the code and PC remains unchanged.

- *Conditional statement* (if (C) S_{then} else S_{else}). If PC implies neither C nor ¬C, Exec returns two symbolic states, corresponding to the two different branches, in which both IP and PC have been modified. IP points to the first statement of the then or the else branch, respectively, while PC is obtained as the logical and of the previous PC and the evaluation of C or ¬C, respectively. Conversely, if PC implies C (¬C) only the then (else) branch can be executed. Thus, Exec returns a single symbolic state in which only IP has been modified to point to the first statement of the `then` (`else`) branch.

- *Loop statement* (while (C) S_{loop}). This statement is handled as the following equivalent statement:

  if(C) {

  S_{loop}*;*

  while (C) S_{loop}

  }

- Thus, the loop is unfolded and Exec is recursively applied, so that at each iteration new symbolic states are constructed.

- *Procedure calls*. Procedure calls are managed by means of macro-expansion. The symbolic execution jumps to the invoked sub-program and executes it, making the necessary assignments to represent parameters passing and return.

A.2. Execution Models

An *execution model* of a program P describes all the (symbolic) states that can be reached during a computation. It is a direct graph whose nodes are symbolic states, and it is formally defined as a tuple:

$$EM(P) = \langle N, n_0, E, T \rangle$$

where:

- N is the set of nodes, i.e., symbolic states reached during the symbolic execution of P;
- $n_0 \in N$, is the initial node, i.e., the initial symbolic state;
- $E = \{(n_i, n_j) \in N \times N \mid n_j \in Exec(n_i, P)\}$ is the set of edges connecting the different nodes.
- $T \in N$ is the set of terminal nodes, i.e., final states of the symbolic execution.

Notice that an execution model, in general, can contain an unlimited number of states and therefore the corresponding graph can have paths of unlimited length. The main reason

for this being the way in which loops are handled, that is by unfolding them. For example, Figure A.1 shows the execution model of program 3.

An *execution path* is a sequence of nodes $(n_1;...;n_k)$ such that:

- $n_i \in N$, $1 \leq i \leq k$;
- n_1 equals n0
- $(n_i,n_{i+1}) \in E$, $1 \leq i \leq k-1$;
- $n_k \in T$

Thus, an execution path represents a single terminating computation belonging to an execution model. For instance, in the execution model, it is possible to identify two different execution paths corresponding to executing the *then* and *else* branches of the program, respectively.

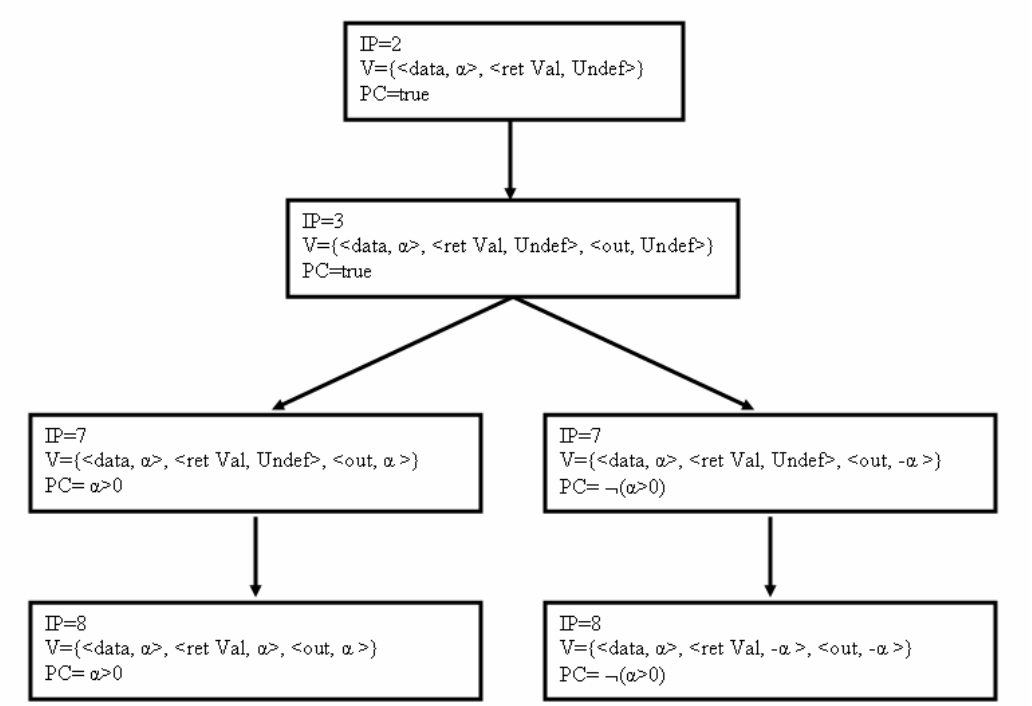


Figure A.1: Execution Model

Automatic Generation of Test Cases using Genetic Algorithms

Harsimran Singh and R. S. Salaria
Dept. of Computer Science,
Thapar Institute of Engineering & Technology, Patiala, India
E-mail: harsimran@eforu.com, rssalaria@yahoo.com

***Abstract* – Today, testing is the most challenging and dominating activity used by industry, therefore, improvement in its effectiveness, both with respect to the time and resources, is taken as a major factor by many researchers. This paper presents a new technique for automatically generating test cases using genetic algorithms (GAs). This technique extends the random testing by the use of genetic algorithms where the fitness function is based on certain factors of a good test suit (set of test cases). A *good test case* is a test case whose chances of finding a bug are more. This technique takes the program as an input and then evaluates the test cases for that program. That is, it is a white box testing technique. An example is given explaining the complete methodology and its effectiveness.**

***Index Terms* – Genetic Algorithms (GAs), software testing, automatic test data generation, random testing.**

I. Introduction

Software testing [1,2] is the most labour-intensive and expensive in the complete software development life cycle, as it costs (both in terms in money and effort) 50% of the cost of a software system development. If the testing process could be automated, the cost of developing software should be reduced significantly. In software testing, a particularly labour intensive component of the complete process is the generation of test cases. *Test case generation* in software testing is the process of identifying program input data. A *test data generator [5]* is a tool that assists a programmer in the generation of test data for a program. The automation of test-data generation is an important step in reducing the cost of software development and maintenance.

There are number of test-data generation techniques that have been automated earlier. *Random test-data generators* [1, 8, 14, 15, 26] select random inputs for the test data from some distribution. *Structured or path oriented test data generators* [27, 28] typically use the program's control-flow graph, select a particular path, and use a technique such as symbolic evaluation [1, 8, 10, 16, 20, 22] to generate test data for that path. *Goal-oriented test data generators* [23, 24] select input to execute the selected goal, such as a statement, irrespective of the path taken. Intelligent test-data generators often rely on sophisticated analysis of the code, to guide the search for new test data.

This paper extends the random test data generation by the use of genetic algorithms (GAs) [3, 4, 13, 17] where the fitness function is based on certain factors like likelihood, close to boundary value, etc. These factors evaluate the goodness of a test suit (set of test cases). A *good test case* has more chances of finding a bug in a program.

For testing software, test data have to be generated and some test data are better at finding errors then others. Therefore, an efficient testing tool has to differentiate good test data from bad test data. Nowadays testing tools are there that automatically generate test data that satisfies certain criteria, such as branch testing, path testing, etc. But, these tools can create problems when complicated software is tested. An efficient testing tool should be general, robust and generate the right test data. Therefore, testing tool should be such that, it must decide where the best test data lies and concentrate its search there. It can be difficult to find correct test data because conditions or predicates in the software restrict the input domain which is a set of valid data.

This paper introduces certain factors for a good test suit. These factors can be used as the fitness function of the GA to find the optimal solution i.e. the best set of test cases.

Deleted: The

Deleted: is a test case that

II. Basic Concepts

In this paper the following code segment (Program 1) has been taken as an example:

```
        int a,b,c;
        scanf("%d,%d,%d",&a,&b,&c);
1,2,3:  if(a<=0 || b<=0 || c<=0)
4:      printf("\nwrong input");
        else
        {
5:          if(a<b)
6:              swap(a,b);
7:          if(a<c)
8:              swap(a,c);
9:          if(b<c)
10:             swap(b,c);
11:         if(a>=b+c)
12:             printf("\nwrong input");
13:         else if(a==b)
            {
14:             if(b==c)
15:                 printf("\nEquilateral");
                else
16:                 printf("\nIsoslist");
            }
            else
            {
17:             if(b==c)
18:                     printf("\nIsoslist");
                else
19:                     printf("\nScalen");
                }
        }
```

Program 1

Denoting the above code segment as *P*. Consider its any test suit *T* which is the set of test cases t_1, t_2, t_3,… Mathematically,

$T=\{t_1, t_2, t_3,\ldots\}$

A test case is a set of values of all input variables. In the above segment there are three input variables, namely, *a*,

_b & c_ declared as integer. So every test case of the above segment is a set of three integer values. Mathematically,

$t=\{i_1,i_2,i_3,\ldots\}$

where i_n is the value of the variable I_n ($n = 1,2,3,\ldots.$), but within its range. For example, in the above code all the three variables are of type integer (2 bytes) i.e. their values range from *–32768* to *32767*. So its any test case is the set of three values, each ranging from *–32768* to *32767*.

III. Test Suit Factors

It is impossible to test software exhaustively (i.e. for each and every value possible). For the exhaustive testing of Program 1 (three variables, each *16* bit long) a Tester needs to execute $2^{16} * 2^{16} * 2^{16} = 2^{48}$ test cases, which is impossible for even a supercomputer to execute.

Therefore, to test the software, tester uses the selected cases only. The selection criteria of the earlier automated test tools are mainly branch coverage, statement coverage, etc. But there are problem with theses tools when complicated programs are there. They may leave very important test cases that are must to be tested.

So a new technique is proposed over here so that all the important test cases can be selected. Selection is done on the basis of various factors of a good test suit (set of test cases). GA helps in selecting the various test cases whose fitness function is based on these factors. Here the factors that contribute to selection of test cases are as:

- Likelihood
- Close to boundary value
- Branch coverage

A. Likelihood

The paths, which are more likely to be executed then others should be given higher priority for testing. *Likelihood* of any test suit is higher than the likelihood of any other test suit, if the test cases of the test suit follow the paths that are more likely to be executed. The likelihood factor will contribute to the selection of any particular test suite for execution. More the likelihood of a test suite higher is its probability to be selected for executions

Mathematically, the likelihood, *L* of a test suit, *T* is evaluated as follows:

$L(T)=1-((1-L(P(t_1)))*(1-L(P(t_2)))*\ldots)$

where $P(t_i)$ is the path followed when the test case t_i is executed. For example, considering Program 1, test case $t=\{1,0,0\}$ follows the path $\{1,2,4\}$, i.e $P(t)=\{1,2,4\}$. And $L(p)$ is the likelihood of the path, *p*. which is evaluated as follows:

1. Using symbolic evaluation method [1, 8, 10, 16, 20, 22], evaluate the Boolean expression of that path. For example, for path $\{1,2,4\}$, Boolean expression is $a > 0 \wedge b \leq 0$.
2. From the complete input domain find the probability of that Boolean expression to be true. For example, the probability of the Boolean expression $a>0 \wedge b\leq 0$ is $(32767/2^{16})*(32769/2^{16})*(2^{16}/2^{16}) = .25$ (approx.), as all the three variables has range from *–32768* to *32767*.
3. The probability above found gives the likelihood of that path.

B. Close to Boundary value

As the chances of bugs are mostly at the boundary values, so the test cases close to boundary values must be given higher priority than the other ones for testing. *Close to boundary value* is the factor that represents the 'how much the test case values are closer to the boundaries'. Consider the path $\{1,2,4\}$ of Program 1, whose Boolean equation is $a>0 \wedge b\leq 0$. So the boundary values for this path is $a=0,b=0$. Test cases with values of *a* and *b* closer to zero are given more priority than with values father than zero. Like likelihood, this factor also contributes to the selection of any particular test case. More the factor of any test suit, higher the probability of the test suit to be selected for execution.

Mathematically, Close to Boundary Value factor, *B* of a test suit, *T* is evaluated as follows:

$B(T) = B(t_1) * B(t_2) * \ldots\ldots$

where B(t) is Close to boundary value factor of a test case t, which is evaluated as:

1. For the test case, evaluate the path that is followed, when the test case is executed. For example, for the above program test case, $t=\{1,0,0\}$ follows the path $\{1,2,4\}$, i.e $P(t)=\{1,2,4\}$.
2. Using symbolic evaluation method, evaluate the Boolean expression of that path. For example, for path $\{1,2,4\}$, Boolean expression is $a > 0 \wedge b \leq 0$.

3. From the Boolean expression evaluate the boundary value expressions by converting any comparison operators $\{<,>,\leq,\geq,=\}$ to '=' comparison operator. For example, for Boolean expression $a>0 \wedge b \leq 0$, the boundary value expressions are $a=0, b=0$.
4. Change the boundary value expressions so that right side of the boundary value expressions becomes zero. For example, the expression, $a+b=c+d$ is converted into $a+b-(c+d)=0$.
5. Now the Close to boundary value factor of a test case is evaluated as

$$B(t)=B(b_1)*B(b_2)*\ldots$$

where $B(b_i)$ is the factor denoting 'how much the test case values are closer to the boundary value expression, b_i (which is of the form *[expression]=0*)'. For example,

$$B(t=\{1,0,0\})=B(a=0)*B(b=0)$$

Now, $B([exp.]=0)$ is evaluated as:

$$1-(|e(t,[exp.])|/|f([exp.])|)$$

where $e(t,[exp.])$ is the actual value of expression when the variables are converted to the actual test case values and $f([exp])$ is the farthermost value of the expression from zero in the complete input domain. For example, for the test case, $t=\{1,0,0\}$ the Close to Boundary Value factor is

$$B(t)=B(a=0)*B(b=0)$$

Now,

$$B(a=0)=1-(|1|/|-32768|)=.99$$
$$B(b=0)=1-(|0|/|-32768|)=1$$

So, the boundary value of the test case is

$$B(t)=.99*1=.99$$

C. Branch coverage

Most of the earlier automated test tools use the branch coverage criterion for selecting test cases. Branch coverage means the percentage of no of edges/branches of the control flowgraph covered by the test suit. *Control flowgraph* is graphical notation of the program that shows the flow of control of that program. The control flowgraph of Program 1 is shown on figure 1. The Branch Coverage factor for any test suit, *T* here is denoted as $R(T)$. To evaluate this, firstly control flowgraph of the program is created. Then, for each test case of a test suit, evaluate the set of branches that has been covered. Take the union of all the evaluated set of branches. Count the number of elements of the resultant set (let it be *n*). At the last, Branch coverage is evaluated as:

$R(T)=n/e$

where e is the total number of edges of control flowgraph.

For example, considering Program 1 whose control flowgraph is shown in figure 1, and the test suit

$T=\{$
$\{0,5,8\},$
$\{3,1,9\},$
$\{8,5,6),$
$\{5\text{-}3,\text{-}6\}$
$\}$

The paths followed by each test case are:

$P(\{0,5,8\})=\{1,4\}$
$P(\{3,1,9\})=\{1,2,3,5,7,8,9,10,11,12\}$
$P(\{8,5,6\})=\{1,2,3,5,7,9,10,11,13,17,19\}$
$P(\{5,\text{-}3,\text{-}6\})=\{1,2,3,4\}$

Here out of twenty nine branches/edges, the total number of covered branches is eighteen, so the branch coverage factor of the test suit, T is:

$R(T)=18/29=0.62(approx.).$

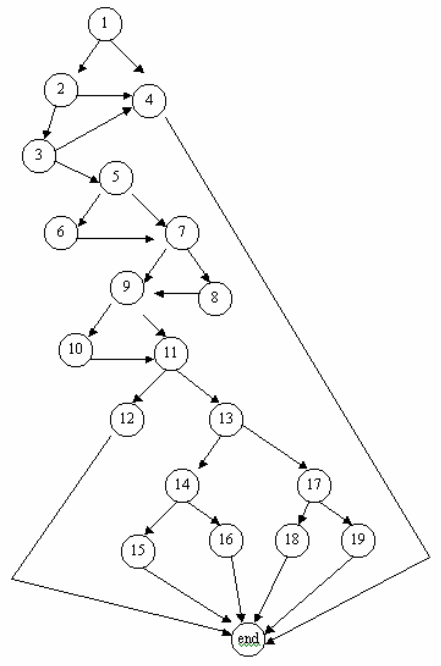


Figure 1

IV. Test Data Generation using GA

Genetic algorithms (GAs) [3, 4, 13, 17] represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's principal of the survival of the fittest. There is a randomized exchange of structured information among a population of artificial chromosomes. GAs are a computer model of biological evolution. When GAs are used to solve optimization problems, good results are obtained surprisingly quickly.

In this paper, Genetic algorithms [3, 4, 13, 17] are used to select software test cases. The general factors to be considered in genetic algorithms are:

- Population Initialization
- The Fitness Function
- Selection/Operations used

The complete process is discussed in this section briefly as below.

A. General Overview

The central objective of the GA used here is to select the best test suit (set of test cases) for the testing purpose. Here a test suit is taken as an individual (chromosome). A Population is the set of test suits. Figure 2 shows the complete structure.

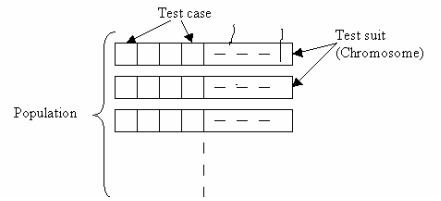


Figure 2

The main thing that should always be taken into the mind is that any test suit cannot have any two same test cases at any time.

B. Population Initialization

As at the start, there is no information about the test cases, which are good and which are bad, that is why test cases are generated randomly i.e. population is initialized randomly.

Most of the automated test tools available now days select test cases based on certain criteria, like branch testing, path testing, etc. But these tools are not suitable for every kind of code, as some criteria cannot always find the good test data for every code segment. This is the reason, why most of the companies are still using random test data generation tools.

That is why; here initialization of the population is done randomly.

C. The fitness function

The fitness function used for every individual is the average of three factors discussed above of a test suit i.e. Fitness function F of test suit, *T* is

$$F(T)=(L(T)+B(T)+R(T))/3$$

Here more the fitness function value, more likely the test suit to be selected. The priority of the test suit to be selected is directly proportional to the fitness function value.

As stated, the central objective of the paper is to select the best test suit as possible. As already explained the three factors, Likelihood, Close to Boundary Value, and Branch Coverage are the factors that decide whether the test suit is good or bad than others. So their average is taken as a fitness function that decide that whether the test suit is more suitable to be selected than other or not.

D. Selection

The selection of the parent chromosomes are done randomly both for crossover and mutation operations, as even a small change in input may change the path of execution. So we cannot say that good parent test cases always yield good child test cases.

E. Uniform Crossover

In uniform crossover, any two test suits (parents) are selected randomly. Interchange randomly some of the test cases of parents ignoring test cases that are common to both the test suits. The result is the two child test suits. As shown in figure 3, there are two common test cases (yellow and light green) that are ignored and there are three exchanges of test cases (Red & purple, blue & green, and white & grey).

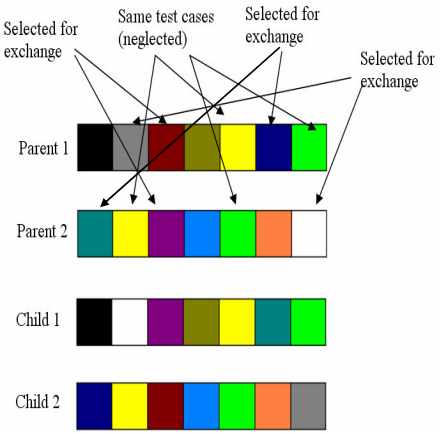


Figure 3

F. Mutation

In mutation a test suit (parent) is selected randomly. Then randomly select some test cases of the parent test suit and replace them with the new test cases generated randomly that are not present in the parent test suit earlier and the new childe test suit gets created. As shown in figure 4, some of the test cases (green, purple and white) of the parent are replaced with newly generated test cases (blue, red and grey) to create a new child test suit.

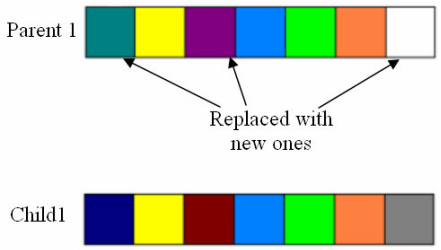


Figure 4

V. An Example

Consider the code segment of Program 2 whose control flowgraph is shown in figure 5. This program takes three input parameters each of type integer i.e. their range is from *-32768* to *32767*.

Now at the start population of three test suits are initialised randomly as:

T1={{5, -15, 20}, {1, 165, 165}, {-1, -1, -1}}

T2={{1, 0, 1}, {160, 13, 0}, {3, 50, 10}}

T3={{-32768, 32767, 0}, {15, -5, 2000}, {2378, 17588, -20}}

```
        void greater(inr a, int b, int c){
1:      if(a>b)
        {
2:              if(a>c)
3:                      printf("Greater value is : %d", a);
                else
4:                      printf("Greater value is : %d", c);
        }
        else
        {
5:              if(b>c)
6:                      printf("Greater value is : %d", b);
                else
7:                      printf("Greater value is : %d", c);
        }}
```

Program 2

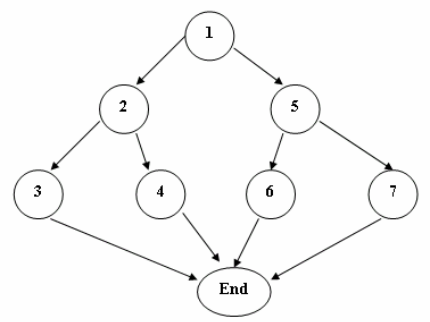


Figure 5

Now Likelihood factor of *T1* is evaluated as:

*L(T1)= 1-((1-L(P({5, -15, 20}))) * (1-L(P({1, 165, 165}))) * (1-L(P({-1, -1, -1}))))*

or *L(T1)= 1-(1-L({1,2,4})) * (1-L({1,5,7})) * (1-(L({1,5,7})))*

or *L(T1)= 1-((1-0.25) * (1-0.25) * (1-0.25) = 1-(0.75*0.75*0.75) = 1-0.4219*

or *L(T1)=0.5781*

Similarly *L(T2)= 0.5781* and *L(T3)= 0.5781*.

Now Close to Boundary Value factor of *T1* is evaluated as:

*B(T1)= B({5, -15, 20}) * B({1, 165, 165}) * B({-1, -1, -1}))*

or *B(T1)=((1-20/65535)*(1-15 / 65535))*((1-164/65535)*(1-0/65535))*((1-0 /65535) *(1-0/65535))*

or *B(T1)= 0.99969*0.99977*0.99749* 1*1*1*
or *B(T1)= 0.9969*

Similarly *B(T2)= 0.9939* and *B(T3)= 0*.

Now Branch Coverage factor of *T1*is evaluated as:

Total no of edges of control flowgraph, *e=10;*

Edges covered by all the test cases, *n=6;*
R(T1)=n/e=6/10=0.6

Similarly *R(T2)= 0.8* and *R(T3)= 0.6*

No fitness of function of each is
F(T1)=(L(T1)+B(T1)+R(T1))/3=(0.5781+0.9969+0.6)/3=0.725
F(T2)=(L(T2)+B(T2)+R(T2))/3=(0.5781+0.9939+0.8)/3=0.7907
F(T3)=(L(T3)+B(T3)+R(T3))/3=(0.5781+0+0.6)/3=0.3927

Now crossover operator is used on two randomly selected test suits, say *T1* and *T2* and two new test suits are generated as

T4={{5, -15, 20}, {160, 13, 0}, {-1, -1, -1}}

T5={{1, 0, 1}, {1, 165, 165}, {3, 50, 10}}

Their fitness function is evaluated as:

F(T4)=(L(T4)+B(T4)+R(T4))/3=(0.5781+0.9948+0.8)/3=0.791

F(T5)=(L(T5)+B(T5)+R(T5))/3=(0.5781+0.9959+0.8)/3=0.7913

Applying mutation on *T4*, a new test suit is generated,

T6={{8, 8, 2}, {160, 13, 0}, {-1, -1, -1}}

Its fitness function is evaluated as:

F(T6)=(L(T6)+B(T6)+R(T6)/3=(0.5781+0.9952+0.8)/3=0.7911

Applying mutation on *T5*, a new test suit is generated,
T7={{1, 0, 1}, {50, -10, 50}, {3, 50, 10}}

Its fitness function is evaluated as:

F(T7)=(L(T7)+B(T7)+R(T7))/3=(0.5781+0.9978+0.8)/3=0.792

Here in all the test suits the fitness function of T7 is the greatest i.e. 0.792 so this test suit is selected for the execution

Hence, Selected test suit is:

T7={{1, 0, 1}, {50, -10, 50}, {3, 50, 10}}

VI. Conclusion

As it is impossible to test the software exhaustively, so this technique is very useful in selecting the best set of test cases (test suit). The selection is based on three factors that evaluate the test case whether it is good or bad. Genetic algorithms are used for this purpose, GAs are found to be the best technique for selection purpose when there is very large population. The GAs give good results and their power lies in the good adaptation to the various and fast changing environments.

The primary objective of this paper was to propose a GA-based software test data generator and to demonstrate its feasibility. The example shows how this can be achieved.

GAs show good results in searching the input domain for the required test sets. GAs may not be the answer to the approach of software testing, but do provide an effective strategy.

VII. Future Work

As the fitness function is taken as the average of the three discovered factors. But the best results can only be achieved if assign different weights to the three factors according to the code segment, as some factor is good for selecting test cases for a particular code segment but some are not so good. So some expert system should be there that can judge the nature of the code segment and then assigns the weights to each factor, which gives the real fitness function value.

New operators, like crossover and mutation can also be defined so that GA can give more efficient results and the optimization process can become much easier and faster.

The one more important aspect of future work is of finding more factors that can compare two test suits for their goodness, so that efficiency of selection process can be improved.

References

[1] B. Beizer. *"Software Testing Techniques",* Van Nostrand Reinhold, 2nd edition, 1990.

[2] R. S. Pressman. *"Software Engineering: A Practitioner's Approach"*, 3rd Edition, McGraw Hill, New York (1992), p. 559.

[3] http://lancet.mit.edu/~mbwall/presentations/IntroToGAs/

[4] http://www.rennard.org/alife/english/gavintrgb.html

[5] R. Ferguson and B. Korel. *"The chaning approach for software test data generation"*. ACM Transactions on Software Engineering Methodology, 5(1):63–86, 1996.

[6] Standard for Software Test Documentation (IEEE829) http://ww.ieee.org.

[7] The Open Group, http://tetworks.opengroup.org/.

[8] R. Boyer, B. Elspas, and K. Levitt. *"SELECT-A formal system for testing and debugging programs by symbolic execution"*. SIGPLAN otices, vol. 10, no. 6, pp. 234-245. June 1975.

[9] C. Rankin. "The Software Testing Automation Framework"

[10] L. Clarke. *"A system to generate test data and symbolically execute programs"*. IEEE Trans. Sofrware Eng., vol. SE-2, no. 3, pp. 215- 222, Sept. 1976

[11] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller. *"SMOTL-A system to construct samples for data processing program debugging"*. IEEE Trans. Software Eng., vol. SE-5, no. 1, pp. 60-66,Jan. 1979.

[12] John Koza. *"Agntic Programming, Ann Arbor*". The University of Michigan Press, 1992.

[13] http://www.cs.qub.ac.uk/~M.Sullivan/ga/ga_index.html

[14] C. Ramamoorthy, S. Ho, and W. Chen. *"On the automated generation of program test data"*. IEEE Trans. Software Eng., vol. SE-2, no. 4. PD. 293-300, Dec. 1976.

[15] Holland J.H. *"Adaptation in natural and artificial system, Ann Arbor*". The University of Michigan Press, 1975.

[16] W. Howden. *"Symbolic testing and the DISSECT symbolic evaluation system"*. IEEE Trans. Software Eng., vol. SE-4, no. 4, pp. 266- 278. 1977. .

[17] Goldberg D. *"Genetic Algorithms"*. Addison Wesley, 1988.

[18] Emmeche C. *"Garden in the Machine. The Emerging Science of Artificial Life"*. Princeton University Press, 1994, pp. 114 ss.

[19] S. R. Ladd. "*Genetic Algorithm in C++*". 1999-2000. Downloadable book. http://www.coyotegulch.com.

[20] D. Ince. *"The automatic generation of test data"*. Comput. J., vol. 30,no. 1,pp. 63-69, 1987.

[21] *"Biological programming is not limited to AG"*, another well-known case is neural networks.

[22] W. Miller and D. Spooner. *"Automatic generation of floating-point test data"*. IEEE Trans. Software Eng., vol. SE-2, no. 3, pp. 223-226, Sept. 1976.

[23] D. L. Bird and C. U. Munoz. *"Automatic generation of random self-checking test cases"*. IBM Systems Journal, 23(3):228–245, 1983.

[24] J.W. Duran and S. Ntafos. *"An Evaluation of Random Testing"*. IEEE Transactions on Software Engineering, 10(4):438–444, July 1984.

[25] Jefferson Offutt, Zhenyi Jin, and Jie Pan. *"The Dynamic Domain Reduction Procedure for Test Data Generation"*. Software Practice and Experience, 29(2):167–193, January 1997.

[26] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. *"Automated test data generation using an iterative relaxation method*". In ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6), pages 231–244, Orlando, Florida, November 1998.

[27] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. *"Test-data generation using genetic algorithms. Software Testing"*. Verification & Reliability, 9(4):263–282, 1999.