Brief Review of Selected Literature on Genetic Algorithm Based Approach for Prioritization Test Case in Static Testing that are Derived from the Source Code

Shehu Malami Sarkin Tudu¹ and Dr Abubakar Sambo Junaidu²

¹Department of Computer Science, Sokoto State University, Sokoto, Nigeria
²Department of Business Administration, Usmanu Dan Fodio University, Sokoto, Nigeria

Abstract

The paper is a review of selected literature on genetic algorithms based approach that shows an extensive research has been made and a relationship has been established between the two major issues in static testing that are derived from the source code.

Keywords: Genetic Algorithms Software Testing.

1. Introduction

Modern technology today makes software takes on dual role. It is a product and at the same time, the vehicle for delivering a product. With this development software organizations must respond to increasingly demanding customers in a globally competitive market and must implement industry standard, at the same time customers are insisting that their systems must be of high quality to satisfied these demand software organizations must have the ability to develop and maintain software to meet customer’s needs. The goal of testing is to find errors, and good test is one that has a high probability of finding errors. Therefore, there is need to design and implement a computer based system or product that will achieve the goal of finding the most errors with minimum of effort. Software testing is one of the most important phases of software engineering, and is a primary technique for achieving high quality software. Hence, software testing involves identifying the test cases which discover errors or detect the presence of fault in the program which eventually cause software failures. However, software testing is a time consuming and expensive task [1], it consumes almost 50% of the software system development resources [3]. Software testing can be done either manually by use of testing tools. Previous research has found that automated software testing is better than manual testing. The process of testing any software system is an enormous task which is time consuming and costly [18] [19]. Software testing is laborious and time-consuming work; it spends almost 50% of software system development resources [18] [19]. Today software testing is expensive, typically consuming roughly half of total cost involved in software development while adding nothing to raw functionality of final product [5]. However, the confidence in software can only achieve through testing, where huge amount of money spent on this phases, to achieve quality test data which resulting in to time consuming. The only remedy to this situation is to minimize time and effort in this process by automating this activity, to satisfy these consumer needs, software organizations must be able to develop and maintain software. However, challenge before the organization is that testing process is time consuming and effort.

2. Software Testing

Software testing remains the primary technique used for achieving high quality software. Software testing is done to detect the presence of faults, which cause software failure. Software testing involves identifying the test cases which discovers errors in the program, software organization must respond to increasingly demanding customers in a globally competitive market and must implement best industry practices by ensuring that before deployment to the consumer they must be sure that the product is of high quality, to satisfy these consumer needs, software organizations must be able to develop and maintain software. However, challenge before the organization is that testing process is time consuming and effort.
2.1 Genetic Algorithms

A genetic algorithm is a search procedure modeled on the mechanics of natural selection rather than a simulated reasoning process. The approach is inspired by Darwin’s theory of evolution, which is based on the survival of fittest. Solution to a problem is considered an individual in a population of solutions. However, in the past, evolutionary algorithms have been applied in many real life situations; GA is an evolutionary algorithm that emerged as a practical, as well as optimization techniques and search base method. A classification schemes based on Genetic Algorithms were proposed to promote the qualities [2][9]. A GA starts with guesses and attempts to improve the guesses by evolution [9]. Genetic algorithm have five basic parts representation of guess called chromosome, initial pool of chromosomes, fitness function, selection function and crossover, mutation operators. Using evolutionary algorithms, some work in the literature address various techniques in developing genetic algorithm based test data generators [10], also a lot work for test data generation have been developed previously and these data can be categorized as structural and functional testing [2]. GA has been previously used in many optimization problems for generating test plans for functionality testing, feasible test cases and in many other areas [3]. Also various techniques have been proposed for generating test data/test cases automatically using GA in structural testing [7], [8]. GA has also used in regression testing object-oriented unit testing as well as in the black box testing for the automatic generation of test cases [12][13][14]. Automated software testing and its implementation cannot only significantly improved the effectiveness and efficiency but also reduce the high cost of software testing [11]. Validation of software through dynamic testing is an area of software engineering where progress toward automation has been slow. On the other hand, software testing and particularly test data generation are labor-intensive, time consuming, and expensive processes [5][15]. Earlier studies in the literature estimated that software testing can consume 50%, or even more, of the development costs [5][16].

2.2 Control Flow Graph

Control flow graph is a simple notation for the representation of control flow. An independent path is any path through the program that introduces at least one new set of processing statements or new condition. A method for generating test sequence using UML state, activity diagram and source code is the transformation of state, activity diagram and source code into Testing Flow Graph (TFG) proposed in [17]. Reference [16] presents a method to drive test scenarios directly from the UML activity diagrams. In another definition, control flow graph known as CFG is a graphical representation of a computer program’s control structure. It uses the elements as follows:- Process blocks, decisions and junctions. The CFC is similar to earlier flowchart, with which it is not to be confused. Weiser [25] used a control flow graph as an intermediate representation of programs for constructing slices. Program graphs are a graphical representation of a program’s source code. The nodes of the program graph represent the statement fragments of the code, and the edges represent the program’s flow of control.

The pseudo code for a given program is simply subtracting the two integers and outputs the result to the terminal. The number subtracted depends on which is the larger of the two; this stops a negative number from being output. In control graph there may be a decision node or vertex is a program point at which the control flow can diverge. Machine language conditional branch and conditional skip instructions are examples of decisions, while most decision node are two-way or binary, example of three-way branches control flow are FORTRAN IF. The design of test cases is generally easier with two-way branches than three-way branches, and there are also powerful test-design tools that can be used.

White-box or logic-driven testing which is our focus on this paper, permits you to examine the internal structure of the program. This strategy derives test data from an examination of the program’s logic (and often, unfortunately, at the neglect of the specification). The goal at this point is to establish, for this strategy, the analog to exhaustive input testing in the black-box approach. Causing every statement in the program to execute at least once might appear to be the answer, but it is not difficult to show that this is highly inadequate, the analog is usually considered to be exhaustive path testing. That is, if you execute, via test cases, all possible paths of control flow through the program, then possibly the program has been completely tested.

There are two flaws in this statement, however. One is that the number of unique logic paths through a program could be astronomically large. To see this,
consider the trivial program represented in Figure 1. The diagram is a control-flow graph. Each node or circle represents a segment of statements that execute sequentially, possibly terminating with a branching statement. Each edge or arc represents a transfer of control (branch) between segments. The diagram, then, depicts a 10- to 20-statement program consisting of a DO loop that iterates up to 20 times. Within the body of the DO loop is a set of nested IF statements. Determining the number of unique logic paths is the same as determining the total number of unique ways of moving from point \( a \) to point \( b \) (assuming that all decisions in the program are independent from one another). This number is approximately \( 10^{14} \), or 100 trillion. It is computed from \( 5^{20} + 5^{19} + \ldots + 5^1 \), where 5 is the number of paths through the loop body. Since most people have a difficult time visualizing such a number, consider it this way: If you could write, execute, and verify a test case every five minutes, it would take approximately one billion years to try every path. If you were 300 times faster, completing a test once per second, you could complete the job in 3.2 million years, give or take a few leap years and centuries.

2.2.1 Path testing

A path through a program is a sequence of instructions or statements that starts at entry, junction, or decision and ends at another, or possibly the same, junction, decision, one or more times. Paths consist of segments. The smallest segment is link- That is, a single process that lies between two nodes.

Path testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through a program. If the set of paths is properly chosen, then we have achieved some measure of test thoroughness. Path testing are the oldest of all structural test techniques because they recorded as being in used at IBM for more than two decade [17].

Basic path testing is a means for ensuring that all independent paths through a code module have been tested [20]. An independent path is any path through the code that introduces at least one new set of processing statements or new condition [17].

To introduce the basis path method, we will draw a flow graph of a code segment. Once you understand
basis path testing, it may not be necessary to draw the flow graph although it may always find a quick sketch helpful. However, using CFG we can compute the number of independent paths through the code using cyclomatic number [20] base on graph theory. The easiest way is to compute the cyclomatic number is to count the number of decision node of the graph and add 1. We want to write a test case to ensure that each of these paths is tested at least once. A path has a loop in it if any node (link) name is repeated.

Basis path testing [20] is a means for ensuring that all independent paths through a code module have been tested. An independent path is any path through the code that introduces at least one new set of processing statements or a new condition [17]. Basis path testing provides a minimum, lower-bound on the number of test cases that need to be written.

The word Path is also used in the more restricted sense of a path that starts at the routine’s entrance and ends at its exit. In practice a test paths are usually entry- to exit path.

2.2.2 Basis Path Testing

Basis path testing [20] is a means for ensuring that all independent paths through a code module have been tested. An independent path is any path through the code that introduces at least one new set of processing statements or a new condition [17]. To introduce the basis path method, we will draw a control flow graph of a code segment. Once you understand basis path testing, it may not be necessary to draw the control flow graph CFG – though you may always find a quick sketch helpful. Using this flow graph, we can compute the number of independent paths through the code. We do this using a metric called the cyclomatic number [20], which is based on graph theory.

2.3 Test case/Test data

A test data are data which has been specifically identified for use in tests, typically of a computer program, some data may be used for confirmation purpose, to verify whether the result obtain base on given input to a function of a program produce as expected, while other data may be used to challenge the ability of a program to respond to unusual. Software testing involves identifying the test cases which discover errors in the program [21]. A set of test data and test programs (test scripts) and their expected results, test cases validate one or more systems requirements and generate a pass or a fail. However, very few test data generation tools are commercially available today [8]. A test case is a documented set of data inputs and operating conditions required to run a test item together with the expected results of the run. The tester is expected to run the program for the test item according to the test case documentation, and then compare the actual results completely agree with the expected results, no error is present or at least has been identified. When some or all of the results do not agree with expected results, a potential error is identified. It is, in principle, necessary to design a test case for each possible TRUE/FALSE combination of the predicates, the predicate are correlated, so not all paths are achievable. If the predicates were all uncorrelated, then each of the $2^n$ combinations would correspond to different path, a different domain, and their sum would correspond to minimum covering set. Software testing is an essential part of software development which guaranteed the validation and verification process of the software. In order to do so we must have to adopt the process of mapping the software for all its transition states and individually validating the output for a set of given input. For a given part of software we will be writing a set of test cases called test suites. In order to find out how a test case is valid we do not have definite mechanism. We basically depend on the testers understanding of the requirement. In this Process we have lot of human error and his basic skill level taken into consideration. This leads to the inclusion of bugs in the system after testing also. To overcome this there are many approach already being taken and getting implemented. Hence we propose the genetic graph based approach to find a better solution of the issue. The graph theory approach will be needed for the implementation of all the graphs to representation of the states of the system. We will implement the total system state as directed graph. A simple graph, denoted $G = (V, E)$, consists of a set of vertices, $V$, and a set of edges, $E$, where the edges are lines that Connect pairs of node. However, one of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. Software testing has two main aspects: test data generation and application of a test data adequacy criterion. A test data generation technique is an algorithm that generates test cases, whereas an adequacy criterion is a predicate that determines whether the testing process is finished. Using evolutionary computations, researchers have previously done some work in developing genetic
algorithms (GA) based test data generators [10] and this can be categorized as structural and functional testing. The purpose of using GA is to identify the most error prone paths in a software construct. However, the software testing techniques is basically the group into two black box testing and white box testing. In black box techniques the test case are generated either formally or informal requirements [21]. Our main concern on this project is the white box which is concern with testing the code of a system with a view to discover error. Testing can be done either manually or automatically by using tools. It is found that automated software testing is better than manual testing. Basically, very few test data generation tools are commercially available today [8]. White box testing is verification techniques software engineers can use to examine if their code works as expected. Also, white box testing is a testing that takes into account the internal mechanism of a system or component.

There are six basic types of testing: unit, integration, function/system, acceptance, regression, and beta. White-box testing which is our main focuses on this project used for three of these six types. white-box testing techniques outlined in this chapter, a software engineer can design test cases that (1) exercise independent paths within a module or unit; (2) exercise logical decisions on both their true and false side; (3) execute loops at their boundaries and within their operational bounds; and (4) exercise internal data structures to ensure their validity [17]. Unit testing is important for ensuring the code is solid before it is integrated with other code. Once the code is integrated into the code base, the cause of an observed failure is more difficult to find. Also, since the software engineer writes and runs unit tests him or herself, companies often do not track the unit test failures that are observed– making these types of defects the most “private” to the software engineer. We all prefer to find our own mistakes and to have the opportunity to fix them without others knowing. Approximately 65% of all bugs can be caught in unit testing [5]. Integration testing, which is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them. Test cases are written which explicitly examine the interfaces between the various units. These test cases can be black box test cases, whereby the tester understands that a test case requires multiple program units to interact. Alternatively, white-box test cases are written which explicitly exercise the interfaces that are known to the tester. Regression testing, which is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

With white-box testing, you must run the code with predetermined input and check to make sure that the code produces predetermined outputs. Often programmers write stubs and drivers for white-box testing. A driver is a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results.

### 2.3.1 Deriving Test Cases

These methods can serve as guidelines for you as you design test cases, while even though it may seem like a lot of work to use these methods, statistics show [5].However, each time code was written there should be test cases for it based on guidelines. That the act of careful, complete, systematic test design will catch as many bugs as the act of testing

<table>
<thead>
<tr>
<th>s/n</th>
<th>Proposal</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Siwen Yu and Jun Ai (2010)</td>
<td>The approach is easy to extend test method, can generate test data based on multiple UML diagrams and high level of Intelligence.</td>
<td>Based on the software test information sequence, test method agents are selected to generate test data through</td>
</tr>
</tbody>
</table>
3. Conclusion

The brief review of selected literature on genetic algorithm based approach shows how extensive research has been made and it shows how genetic algorithm improved and provide significance differences between the existing case generations by ensuring all the processing nodes of the independent paths through the code module has been tested. This approach has made the case easier for the tester to quickly identify the test cases which discover errors in the program.

References


[15] Jose Carlos, Mario, Alberto, evaluating feasible evolutionary (2008), and unfeasible Francisco "A strategy for test cases for the testing of object-oriented software" ACM.


