A Genetic Algorithm is Used for Test Suite Generation Aimed for Evolutionary Testing

[1]Deepa Roshan Thomas, [2]Kishore Bonela
[1,2]Dept. of CSE, Chaitanya Engineering College, Visakhapatnam, AP, India

Abstract
For creating effective programming, testing is a critical part. In programming testing, giving info, executes it and check expected yield. Numerous strategies which naturally deliver inputs have been proposed throughout the years, and today can create test suites with high code scope. In programming testing a typical situation is that test information are produced and an analyzer physically includes test cases. It is a troublesome errand to create test cases physically yet it is critical to deliver little illustrative test sets and this representativeness is normally measured utilizing code scope. Be that as it may, there is a central issue with the regular methodology of focusing on one scope objective at once. Scope objectives are not free, not similarly troublesome, and some of the time infeasible—the aftereffect of test era is in this manner subject to the request of scope objectives and what number of them are achievable. For taking care of these issues, propose a novel worldview which is era of entire test suite in light of pursuit based testing. Rather than developing every experiment independently, advance all the experiments in a test suite in the meantime. Toward the end, the best coming about test suite is minimized.

Keywords
Test Case Generation, Branch Coverage, Search Based Technique. Length, Software-Based Software Engineering, Branch Coverage, Local Search

I. Introduction
There is one celebrated saying that "Over testing is a Sin and Under Testing is a Crime". A percentage of the principle challenges in testing are that comprehensive testing is unrealistic, when to quit testing can't be surveyed and there is no real way to demonstrate the nonappearance of blunders. With the expanded pace of creation timetables, the colossal expansion of programming outline procedures and programming dialects, and the expanded size of programming applications, programming testing has advanced from a standard quality confirmation movement into a sizable and complex test as far as reasonability and adequacy. The significant difficulties to programming testing in today's business surroundings are:

A. Efficiency
Is the test cycle too long? In what manner would you be able to guarantee each test is a decent venture of time and cash?

B. Thoroughness
In what manner would you be able to tell when you are done trying? In what manner would you be able to be sensibly certain the system is sans bug?

C. Resource Management
Are trying assets deliberately dispensed, concentrating on the most astounding danger components of the product?

Are the practically focal parts of the project getting a worthy level of testing? Practically speaking, unit level testing ranges from the impromptu tests done by software engineers as they are composing code to methodical white box testing, where Unit level testing is a piece of an each unit must be tried and recorded by a QA and Test bunch. In either case, the analyzer starts with the objective of scope, for it is the very reason for unit level testing [1] to accomplish the most elevated amount of scope conceivable. Unit testing is imperative since it is performed right on time in the improvement procedure and it is more financially savvy at finding blunders. The best test of unit level testing is to distinguish a base arrangement of unit level tests to run. In a perfect world, each conceivable way of a system would be tried, representing every executable choice in every conceivable mix. Be that as it may, this is incomprehensible when one considers the huge number of potential ways inserted in any given system (2 to the force of the quantity of choices). The test is to confine a subset of ways that give scope to every single testable unit, and to make that subset as insignificant and free of unit-level redundancies as could reasonably be expected. Myers apropos characterizes programming testing as "a procedure of executing a system with the goal of discovering blunders". Utilizing the relationship of a restorative finding, an effective examination is one that looks for and finds an issue, as opposed to one that uncovers nothing and gives a misguided feeling of prosperity. In view of this definition, a great arrangement of experiments ought to be one that has a high risk of revealing beforehand obscure mistakes, while an effective test run is one that finds these blunders. Keeping in mind the end goal to recognize every single conceivable blunder inside a project, comprehensive testing is required to practice all conceivable info and intelligent execution ways. Aside from extremely paltry projects, this is monetarily unfeasible if not outlandish. Along these lines, a commonsense objective for programming testing would be to augment the likelihood of discovering blunders utilizing a limited number of experiments, performed in least time with least exertion. Because of the focal significance of experiment configuration for testing, an expansive number of testing strategies, intended to help the analyzer with the choice of fitting test information, have been created in the course of the most recent decades. Existing experiment plan strategies can basically be separated into discovery tests and white-box tests. Discovery test cases are resolved from the particular of the project under test, though, white-box test cases are gotten from the inward structure of the product. In both cases, complete computerization of the experiment outline is troublesome [4, 9]. Robotization of the discovery test is just seriously conceivable if a formal particular exists, and devices supporting white-box tests are constrained to program code instrumentation and scope estimation because of the cutoff points of typical execution. Experiment plan itself is additionally dependent on the analyzer. In this manner, experiment outline as a rule must be performed physically. Manual experiment outline, in any case, is time-serious and helpless to mistakes. The nature of the test is intensely subject to the execution of the single analyzer.

II. Related Work
In the writing is branch scope, yet on a basic level whatever other

scope foundation or related procedures, for example, transformation testing are managable to robotized test era. Metaheuristic seek procedures have been utilized as a different option for typical execution-based methodologies [6].Search-based systems have additionally been connected to test object-situated programming utilizing technique successions [43] or specifically hereditary programming [8], [10]. At the point when producing test cases for item situated programming, subsequent to the early work of Tonella [9] , creators have attempted to manage the issue of taking care of the length of the test arrangements, for instance, by punishing the length straightforwardly in the wellness capacity. Some other scope rule is managable to mechanized test era. For instance, transformation testing is frequently viewed as an advantageous test objective and has been utilized as a part of an inquiry based test era environment. As of late, Harman et al. [5] proposed a hunt based multi-target approach in which, albeit every objective is still focused on separately. Furthermore, there is the auxiliary goal of augmenting the quantity of guarantee focuses on that are incidentally secured. All methodologies specified so far focus on a solitary test objective at once—this is the overwhelming technique. There are some striking special cases in hunt based programming testing. The works of Arcuri and Yao[1] and Baresi et al. [2] utilize a solitary arrangement of capacity calls to amplify the quantity of secured branches while minimizing the length of such an experiment. A disadvantage of such a methodology is, to the point that there can be clashing trying objectives, and it may be difficult to cover every one of them with a solitary test grouping, paying little mind to its length. With respect to streamlining of a whole test suite in which all experiments are considered in the meantime, we know about just the work of Baudry et al. [3] In that work, test suites are advanced with a pursuit calculation concerning transformation examination. In the writing of testing item arranged programming, there are additionally systems that don't specifically go for code scope, with respect to sample, executed in the Randoop [7] apparatus.

III. Design of GA

GA is a metaheuristic search technique that simulates the evolution of natural systems. It is often exploited to solve search and optimization problems. It is usually infeasible to exhaustively evaluate the entire input space and thus GA is used to produce good solutions in reasonable time by evaluating only a small portion of the input space. The basic GA first constructs an initial population randomly and then iterates through the following procedures until stopping criteria hold. It assesses the fitness value of all the individuals in the population. Individuals with high fitness values have a better chance to evolve into the next generation by applying genetic operators such as selection, crossover, and mutation.

When using GA to solve pairwise test generation problem, the following design decisions have to be made: chromosome encoding, fitness function, and genetic operators.

A. Chromosome Encoding

Chromosome encoding is the representation of an individual which is the candidate solution of the problem. In the scenario of pairwise test generation, the solution is often a suitable test suite of SUT. In the literature, there are several encoding methods such as bit strings, floating point, and integer. We will use integer encoding to represent the individual. This encoding method encodes a set of test cases as an array of integer values. Each integer corresponds to a possible value of a parameter of SUT. Thus, an individual is an array of lists of integers and each list represents a test case. The length of each list is equal to the number of the parameters. The size of an individual, denoted by m, is the number of the test cases. Our goal is to find the optimal m to cover 100% pairwise combinations of parameter value. If m has been reported in literatures, we set this value as the initial test suite size in our proposed algorithm and search for a solution. Otherwise, we use the binary search approach presented by Cohen et al. [11] to determine m.

According to Definition 1, SUT has k parameters and each parameter p_i has v_i(1 6 i6 k) values. When using the above encoding, p_1 has values of 1, 2, 3, ⋯, v_1, p_2 has values of $v_1 + 1$, $v_1 + 2$, $v_1 + 3$, ⋯, v_2, and p_k has values of $v_{k-1} + 1$, $v_{k-1} + 2$, $v_{k-1} + 3$, ⋯, v_k. These parameters and values are expressed by text file as illustrated in Fig. 1. This text file will be used as input for our algorithm.

p_1: 1, 2, 3, ⋯ , v_1
p_2: v_1 + 1, v_1 + 2, v_1 + 3, ⋯ , v_2
⋮
p_k: v_k 1 + 1, v_k 1 + 2, v_k 1 + 3, ⋯ , v_k

Fig. 1: Input Text File of SUT

The total number of pairs to be covered is denoted by AP and is calculated as:

$$AP = v_1 \sum_{i=2}^{k} v_i + v_2 \sum_{i=3}^{k} v_i + \cdots + v_{k-1} v_k$$

B. Fitness Function

A fitness function for pairwise testing is often a given coverage criterion which measures the goodness of an individual. Grindalet al.[24] defined that 100% pairwise coverage requires that every possible pair of interesting values of any two parameters is included in some test case. We will use this 100% pairwise coverage as our fitness function. Thus, the fitness function is the total number of different pairs covered by all the test cases in an individual. If an individual covers more different pairs than others, it is better than others. An individual becomes a solution when it covers all pairs.

C. Genetic Operators

Another important issue of GA is genetic operators including selection, crossover, and mutation. As for the selection operator, we employ fitness proportionate selection to determine which individuals to be chosen as parents for reproduction. In fitness proportionate selection, individuals are selected in proportion to their fitness values: if individuals have higher fitness values, they are selected more often. Let s = Pi fi be the sum of all individuals' fitness values. A random number n is picked from 0 to s. If n falls within the range of some individual in the array of individual ranges, this individual is selected.

After selection, the selected parents are copied and then crossover mixes and matches parts of these two copied parents to form better children. Our crossover mechanism uses both single-point and multiple-point random crossover. Single-point random crossover chooses a number c randomly from 0 to the length of an individual and exchanges all the indexes smaller than c. In multiple-point crossover, individuals are regarded as a ring. Several unique points are picked at random, breaking the ring into several segments. A

segment from one ring is exchanged with one from another ring to produce new offspring. In our testing scenarios, if the size of an individual is small, we use single-point random crossover to produce offspring; otherwise, we use multiple-point random crossover to get better solution.

After crossover, we use integer randomization mutation to change the genes of new generated offspring with a given higher mutation rate as suggested by Tate and Smithto find a solution faster. As for the mutation mechanism, mutation operator uses both single point and multiple-point integer random mutation. In single-point integer random mutation, a gene to be mutated is picked at random and then is replaced with randomly selected valid value of the parameters; while multiple-point integer random mutation picks several genes randomly and replaces them with randomly selected valid values of the parameters. After mutation operation, new mutated offspring will become individuals of the population by replacing two individuals with the lowest fitness values.

In order to find the solution faster, our algorithm mutates the best individual at the end of each generation to generate new mutant that replaces the individual with the lowest fitness value. This best individual is still kept in future population.

IV. Pairwise Testing

When generating test suite with pairwise testing, the input space of the software under test (SUT) can be modeled as a collection of parameters, each of which assumes one or more values. Pairwise testing aims at selecting a subset from the complete set of parameter value combinations such that all pairs of parameter values are in the selected subset. Each selected parameter value combination will generate at least one test case for SUT. The set of test cases is often called test suite which is represented by covering array (CA) defined as below.

Definition 1 (Covering Array). Let SUT have k parameters and each parameter pi have vi(1 6 i6 k) values. A covering array CA $(N; v_1^{p_1} v_2^{p_2} \cdots v_k^{p_k}, t)$ is an N ×k matrix. Each row of this matrix is a test case. N is the number of test cases and t is the strength of the covering array. Each N × t subarray contains at least one occurrence of each t-tuple corresponding to the t columns. If $v_1 = v_2 = \cdots = v_k = v$, the corresponding covering array is said to be uniform. It is denoted as

CA(N;v^k,t).

When t = 2, it is called 2-way covering array. Testing with 2-way covering array is called pairwise testing. The intent of pairwise testing is to reduce the number of test cases.

For example, we assume SUT has three parameters: p_0, p_1, and p_2. The possible values for each parameter are $\{a_0,a_1\}$, $\{b_0,b_1\}$, and $\{c_0, c_1\}$ respectively. The total number of possible combinations of all parameter values is 23:

(a_0,b_0,c_0), (a_0,b_0,c_1), (a_0,b_1,c_0), (a_0,b_1,c_1),
(a_1,b_0,c_0), (a_1,b_0,c_1), (a_1,b_1, c_0), (a_1,b_1,c_1).

When testing this SUT with pairwise testing, the generated test cases will cover each pair of parameter values.

There are 12 such pairs:

$\{a_0,b_0\},\{a_0,b_1\},\{a_0,c_0\},\{a_0,c_1\}$,

$\{a_1,b_0\},\{a_1,b_1\},\{a_1,c_0\},\{a_1,c_1\}$,
$\{b_0,c_0\},\{b_0,c_1\},\{b_1,c_0\},\{b_1,c_1\}$.

In this case, the following set of four combinations suffices: (a_0,b_0,c_1),(a_0,b_1,c_0),(a_1,b_0,c_0),(a_1,b_1,c_1).

Table 1 shows the covering array CA $4; 2^3, 2)$ for this SUT.

Table 1: CA(4;2^3,2)

	$p0$	$p1$	$p2$
$t1$	$a0$	$b0$	$c1$
$t2$	$a0$	$b1$	$c0$
$t3$	$a1$	$b0$	$c0$
$t4$	$a1$	$b1$	$c1$

In the above example, it can be seen that pairwise testing can reduce the required number of test cases from 8 to 4, a 50% reduction. With larger parameter value combinations, the result will be better. In this paper, we try to generate near-minimum covering array (test suite) to reach 100% pairwise coverage for SUT, especially in the scenario of large parameter value combinations.

V. Cyclomatic Complexity Measure

Cyclomatic complexity [11, 17] (or conditional complexity) is software structural metric (measurement) used to measure the complexity of a program using Control flow graph of the program. The cyclomatic complexity of a structured program is defined as M=E-N+2P where M- Cyclomatic Complexity, E- the number of edges of the graph, N- The number of nodes of the graph and P- The number of disconnected components.

It provides lower bound on the number of test cases required to achieve branch coverage. The amount of test effort is better judged Cyclomatic Complexity. If there are fewer test cases than the measure then missing cases are to be found and more test cases than the measure shows that the coverage can be achieved with less number of test cases.

VI. Proposed System

The proposed system develops a tool for test suite generation which takes control flow graph as input and automatically generates test cases from the input domain of various variables using tabu search technique. The architecture of the proposed work is shown in figure Generator takes the source code of programs for which test case is to be generated and generates Control Flow Graphs.

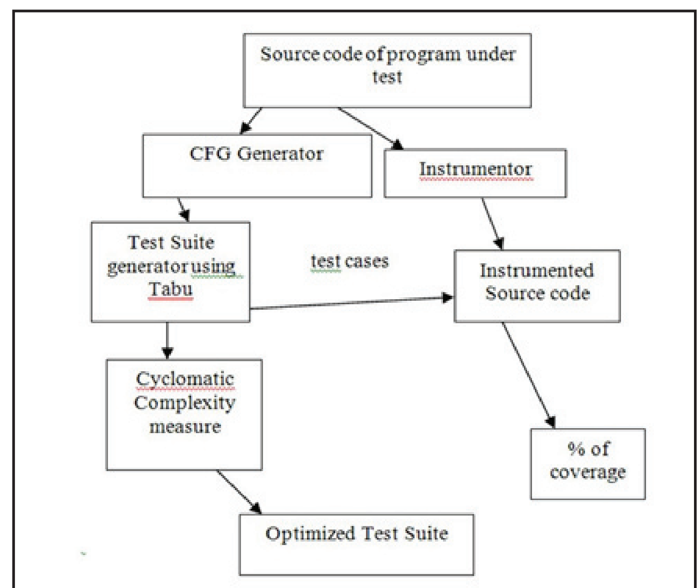

Fig. 2: Proposed SystemArchitecture

VII. Methodology

The various steps in the automated framework of test case generation are,

1. Taking source code under test as input CFG generator generates CFG.
2. Find the Cyclomatic Complexity measure.
3. The CFG is analyzed and the branching condition information is extracted.
4. The test cases are generated for each condition from input domain of thevariables involved in the condition using tabu search technique.
5. Find the compliance of number of test cases with Cyclomatic Complexity measure.
6. The generated test cases are applied to the instrumented source code to check the branch coverage.
7. The best test cases form an effective test suite for the given source code under test.

A. Tabu Search Technique

The tabu search technique is a metaheuristic technique which is proven successful in real world applications such as travelling salesman problem. Recently it is found suitable for test case generation problems in software testing. But only few results have been published with relatively few samples and it must be further proven with all data types of input domain and with more samples. The tabu search algorithm is given as,

```
Begin
 Initialize Current Solution
 Store Current Solution in CFG
 Add Current Solution to tabu list ST
 Select a sub goal node to be covered
 Do calculate neighborhood candidates
 For each candidate do
 If (candidate value in node n <CFG in node n) then
Store candidate in CFG
end if
end for
if (sub goal node not covered) then
Add Current Solution to tabu list ST
else Delete tabu list ST end if
Select a sub goal node to be covered and
Current solution
if (Current Solution is depleted) then
 Add Current Solution to tabu list LT
 Apply a backtracking process:
 New Current Solution and maybe new sub goal node
end if
while (NOT all nodes covered AND number
of iterations<MAXIT)
end
```

VIII. Evolutionary Testing

Evolutionary testing is characterized by the use of metaheuristic search techniques for test case generation. The test aim considered is transformed into an optimization problem. The input domain of the test object forms the search space which a search algorithm explores in order to find test data that fulfils the respective test aim. Neighborhood search methods like hill climbing are not suitable in such cases. Therefore metaheuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing, or tabu search [3, 5, 6]. In this work, evolutionary algorithms are used to generate test data because their robustness and suitability for the solution of different test tasks has already been proven in previous work, e.g. [8]. The most of the previous works in applying search techniques are not taking into account float values for input domain. The first work in applying tabu search to test case generation is in [3] given by Diaz and the cyclomatic complexity is not considered. The proposed work extends the previous work and applies tabu search technique to test case generation in compliance with cyclomatic complexity measure for unit testing and compares the performance with random test case generation based on the measures of test suite size and branch coverage.

IX. Conclusion

Software testing is an important activity and critical too in deciding quality of the software. Test suite generation is vital part of testing process which determines the quality of test. This technique of automated generation of test cases from the input domain can assist the developers and testers in performing unit testing with minimum time and resources. Also the optimized number of test cases generated is much helpful in regression testing which otherwise carried out with greater number of test cases. The technique can be further extended for multiple coverage criteria. Also the effectiveness can be further proven with fault detection effectiveness.

References

[1] Arcuri, X. Yao,"Search Based Software Testing of Object-Oriented Containers", Information Sciences, Vol. 178, No. 15, pp. 3075- 3095, 2008.

[2] L. Baresi, P.L. Lanzi, M. Miraz,"Testful: An Evolutionary Test Approach for Java", Proc. IEEE Int'l Conf. Software Testing, Verification and Validation, pp. 185-194, 2010.

[3] Baudry, F. Fleurey, J.-M. Je´ze´quel, Y. Le Traon, "Automatic Test Cases Optimization: A Bacteriologic Algorithm", IEEE Software, Vol. 22, No. 2, pp. 76-82, Mar./Apr. 2005.

[4] G. Fraser, A. Arcuri,"Evosuite: Automatic Test Suite Generation for Object-Oriented Software", Proc. 19th ACM SIGSOFT Symp. and the 13th European Conf. Foundations of Software Eng., 2011.

[5] M. Harman, S.G. Kim, K. Lakhotia, P. McMinn, S. Yoo,"Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle CostProblem", Proc. Third Int'l Conf. Software Testing, Verification, and Validation Workshops, 2010.

[6] P. McMinn,"Search-Based Software Test Data Generation: A Survey", Software Testing, Verification and Reliability, Vol. 14, No. 2, pp. 105- 156, 2004

[7] Pacheco, M.D. Ernst,"Randoop: FeedbackDirected Random Testing for Java", Proc. Companion to the 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems and Application, pp. 815-816, 2007.

[8] J.C.B. Ribeiro,"Search-Based Test Case Generation for Object- Oriented Java Software Using StronglyTyped Genetic Programming", Proc. GECCO Conf. Companion.

Deepa Roshan Thomas is pursuing M.Tech on Software Engineering inthe department of Computer Science & Engineering from Chaitanya Engineering College, Visakhapatnam, A.P, INDIA.

Kishore Bonela, Working as Assistant Professor in Department of Computer Science & Engineering, Chaitanya Engineering College, Visakhapatnam. A.P, INDIA.