# Optimized test suite generation using tabu search technique

**Geetha Devasena M.S.*[1] and Valarmathi M.L.[2]**
[1*]Department of CSE., Sri Ramakrishna Engineering College, Coimbatore, Tamilnadu, India,
aishumma@yahoo.co.in
[2] Department of CSE., Government College of Technology, Coimbatore, Tamilnadu, India

**Abstract-** Software testing is inescapable activity of software development and crucial to the software quality. It is widely deployed by programmers and tester but it is difficult due to the complexity of software. The program with the moderate complexity cannot be tested completely. Innovative methods are emerging to perform testing as a whole and unit testing in particular with minimum effort and time. Unit testing is mostly done by developers under a lot of schedule pressure since the software companies find a compromise among functionality, time to market and quality. Thus there is a need for reducing unit testing time by optimizing and automating the process. Test suite generation is an error-prone, tedious and time consuming part of unit testing. A novel technique is proposed to automatically generate test cases from the input domain using metaheuristic search technique tabu search for branch coverage criteria with respect to cyclomatic complexity measure**.**
**Key words:** Software testing, Unit testing, Branch Coverage Criteria and Tabu Search.

## Introduction

There is one famous saying that "Over testing is a Sin and Under Testing is a Crime". Some of the main challenges in testing are that exhaustive testing is not possible, when to stop testing cannot be assessed and there is no way to show the absence of errors. With the increased pace of production schedules, the tremendous proliferation of software design methodologies and programming languages, and the increased size of software applications, software testing has evolved from a routine quality assurance activity into a sizable and complex challenge in terms of manageability and effectiveness. The major challenges to software testing in today's business environment are,

• **Efficiency.** Is the test cycle too long? How can you ensure every test is a good investment of time and money?

• **Thoroughness.** How can you tell when you are done testing? How can you be reasonably sure the program is bug-free?

• **Resource Management.** Are testing resources strategically allocated, focusing on the highest-risk elements of the software? Are the functionally central parts of the program receiving an acceptable level of testing? In practice, unit level testing ranges from the ad hoc tests done by programmers as they are writing code to systematic white box testing, where Unit level testing is part of a every unit must be tested and documented by a QA and Test group. In either case, the tester begins with the goal of coverage, for it is the very purpose of unit level testing [1] to achieve the highest level of coverage possible. Unit testing is important because it is performed early in the development process and it is more cost-effective at locating errors. The greatest challenge of unit level testing is to identify a minimum set of unit level tests to run. In an ideal world, every possible path of a program would be tested, accounting for all executable decisions in all possible combinations. But this is impossible when one considers the enormous number of potential paths embedded in any given program (2 to the power of the number of decisions). The challenge is to isolate a subset of paths that provide coverage for all testable units, and to make that subset as minimal and free of unit-level redundancies as possible. Myers aptly defines software testing as "a process of executing a program with the intention of finding errors". Using the analogy of a medical diagnosis, a successful investigation is one that seeks and discovers a problem, rather than one that reveals nothing and provides a false sense of well-being. Based on this definition, a good set of test cases should be one that has a high chance of uncovering previously unknown errors, while a successful test run is one that discovers these errors. In order to detect all possible errors within a program, exhaustive testing is required to exercise all possible input and logical execution paths. Except for very trivial programs, this is economically unfeasible if not impossible. Therefore, a practical goal for software testing would be to maximize the probability of finding errors using a finite number of test cases, performed in minimum time with minimum effort. Due to the central importance of test case design for testing, a large number of testing methods, designed to help the tester with the selection of appropriate test data, have been developed over the last decades. Existing test case design methods can essentially be differentiated into black-box tests and white-box tests. Black-box test cases are determined from the specification of the program under test, whereas, white-box test cases are derived from the internal structure of the software. In both cases, complete automation of the test case design is difficult [4, 9]. Automation of the black-box test is only meaningfully possible if a formal specification exists, and tools supporting white-box tests are limited to program code instrumentation and coverage measurement due to the limits of symbolic execution. Test case design itself is also reliant on the tester. Thus, test case design usually has to be performed manually. Manual test case design, however, is time-intensive and susceptible to errors. The quality of the test is heavily dependent on the performance of the single tester.

## EXISTING SYTEMS

### RANDOM TEST DATA GENERATION

Random test data generation techniques [2] select inputs at random until useful inputs are found. This technique may fail to find test data to satisfy the requirements because information about the test requirements is not incorporated. The various disadvantages of this method are such as it is appropriate only for simple and small programs, many sets of values may lead to the same observable behavior and are thus redundant and the probability of selecting particular inputs that cause buggy behavior may be astronomically small.

### STATIC METHODS

Sstatic methods [14] generate test cases without execution from several constraints based on the input variables of the program under test. Static techniques have several problems, such as treatment of loops, resolution of computed storage locations and computational cost.

### DYNAMIC METHOD

Dynamic test-data generation technique [14] collects information during the execution of the program to determine which test cases come closest to satisfying the requirement. Then, test inputs are incrementally modified until one of them satisfies the requirement. Most dynamic techniques use search based software techniques.

### SEARCH BASED SOFTWARE TESTING

Search-Based Software Engineering (SBSE) is the application of optimization techniques (OT) in solving software engineering problems. Optimization is the process of attempting to find the best possible solution amongst all those available. The percentage of application of search based techniques to software testing is 70% as shown in figure 1.

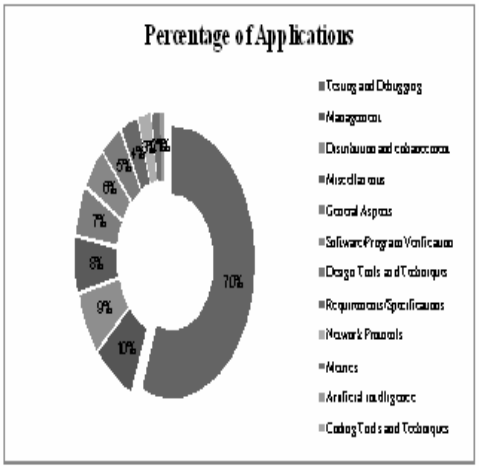


Fig. 1- Application of SBSE

Software testing is a suitable candidate for Search-Based Software Engineering because the generation of software tests is an undecidable problem [15, 16] and a program's input space is very large, exhaustive enumeration is infeasible. In order to carry out evolutionary testing, the task of test case design is transformed into an optimization problem that, in turn, is solved with meta-heuristic search techniques, such as evolutionary algorithms or simulated annealing. The input domain of the system under test represents the search space in which test data fulfilling the test objectives under consideration is sought. The aim of evolutionary testing is to increase the quality of the tests and to achieve substantial cost savings in system development by means of a high degree of automation. It has been proved in various case studies that evolutionary testing has the potential to improve the effectiveness and efficiency of the test process significantly. An overview of different applications of evolutionary testing is provided by McMinn [12].

### SYMBOLIC TEST CASE GENERATION TECHNIQUE

Symbolic test-data generation techniques [7, 8] assign symbolic values to the variables to create algebraic expressions for the constraints in the program, and use a constraints solver to find a solution for these expressions that satisfies a test requirement. Symbolic execution cannot determine which symbolic values of the potential values will be used. The key ingredients of the symbolic technique include the choice of representation for the problem, the definition of a neighborhood on the configuration space and the definition of a cost-function. Symbolic execution cannot find floating point inputs because the constraint solvers cannot produce floating point constraints.

### HILL CLIMBING TECHNIQUE

Hill climbing test data generation technique improves solution by investigating neighbors. Hill climbing can be used to solve problems that have many solutions, some of which are better than others. It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. Hill climbing Technique stucks in local minima and plateaux.

### STRUCTURAL TESTING

**Bug statistics**

The bug statistics[17] through SDLC collected from various sources given by Boris Beizer for a program of 1,00,000 lines of code shown in figure 2, among the other bugs structural bugs are the highest and half of the structural bugs are control flow and sequence bugs as shown in figure 3.2.

**THE TAXONOMY OF BUGS**

SIZE OF SAMPLE—6,877,000 STATEMENTS (COMMENTS INCLUDED)
TOTAL REPORTED BUGS—16,209—BUGS PER 1000 STATEMENTS—2.36

| | | |
|---|---|---|
| 1XXX REQUIREMENTS | 1317 | 8.1% |
| 2XXX FEATURES AND FUNCTIONALITY | 2624 | 16.2% |
| 3XXX STRUCTURAL BUGS | 4082 | 25.2% |
| 4XXX DATA | 3638 | 22.4% |
| 5XXX IMPLEMENTATION AND CODING | 1601 | 9.9% |
| 6XXX INTEGRATION | 1455 | 9.0% |
| 7XXX SYSTEM, SOFTWARE ARCHITECTURE | 282 | 1.7% |
| 8XXX TEST DEFINITION AND EXECUTION | 447 | 2.8% |
| 9XXX OTHER, UNSPECIFIED | 763 | 4.7% |

Sample Bug Statistics.

-BORIS BEIZER

Fig. 2- Bug Statistics

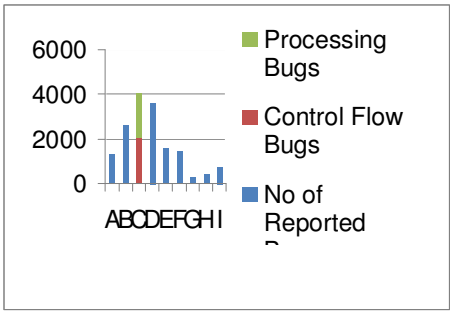


Fig. 3-Bar Graph representation of Bug Statistics

A-Requirements
B-Features and Functionality
C-Structural Bugs
D-Data
E-Implementation and Coding
F-Integration
G-System and software Architecture
H-Test Definition and Execution
I-Other, unspecified

## 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.

## 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 meta-heuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing, or tabu search [5, 6, 13]. 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. [10]. 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.

## 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 4.1.The Control Flow Graph Generator takes the source code of programs for which test case is to be generated and generates Control Flow Graphs.

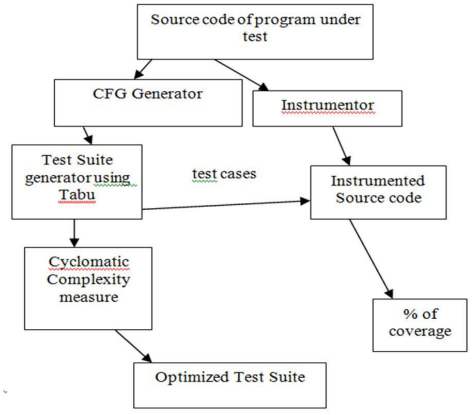


Fig. 4-Flow diagram of Proposed System

## 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 the

variables 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.

**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
```

**RESULTS**

The proposed technique has been tested with 12 benchmarking samples including the triangle classifier program which is widely used in various research papers [1, 3, 13] in the test suite generation. The results obtained are encouraging and tabu search technique performs better than random technique. The Performance measures such as the Test Suite Size, Percentage of branch coverage are considered for comparison of the techniques. Also the test suite size is compared with the cyclomatic complexity of the program structure under test which gives the measure of test cases required to cover the program.

$$\%\ \text{of Test suite size} = \frac{\text{Total number of Test cases}}{\text{Total number of branches}} \times 100$$

$$\%\ \text{of branch coverage} = \frac{\text{100- Total no. of unfeasible branches}}{\text{Total number of branches}} \times 100$$

The results got by random technique can be given in table 1.

Table 1- Results of Random Technique

| Samples | Test Suite Size | % Of Branch Coverage | Cyclomatic Complexity |
|---|---|---|---|
| S1 | 8 | 75 | 3 |
| S2 | 5 | 80 | 2 |
| S3 | 7 | 100 | 3 |
| S4 | 3 | 100 | 2 |
| S5 | 9 | 77.77 | 3 |
| S6 | 11 | 81.8 | 3 |
| S7 | 5 | 100 | 2 |
| S8 | 6 | 100 | 3 |
| S9 | 5 | 100 | 2 |
| S10 | 8 | 87.5 | 3 |
| S11 | 10 | 88.88 | 3 |
| S12 | 15 | 93.33 | 4 |

The results show that the branch coverage varies from75% to a maximum of 100% and that is achieved with more number of test cases than the calculated Cyclomatic Complexity measure. The results got by tabu search technique are given in table 2.

Table 2- Results of Tabu Search Technique

| Samples | Test Suite Size | % Of Branch Coverage | Cyclomatic Complexity |
|---|---|---|---|
| S1 | 3 | 100 | 3 |
| S2 | 2 | 100 | 2 |
| S3 | 3 | 100 | 3 |
| S4 | 2 | 100 | 2 |
| S5 | 3 | 100 | 3 |
| S6 | 3 | 100 | 3 |
| S7 | 2 | 100 | 2 |
| S8 | 3 | 100 | 3 |
| S9 | 2 | 100 | 2 |
| S10 | 3 | 100 | 3 |
| S11 | 3 | 100 | 3 |
| S12 | 4 | 100 | 4 |

The result clearly shows that the branch coverage is high and that is achieved with as many numbers of test cases as calculated by Cyclomatic Complexity measure. The performance analysis graph based on the number of test cases in the test suite and the percentage of branch coverage of both the techniques is given in figure 5 and 6 respectively.

## 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.

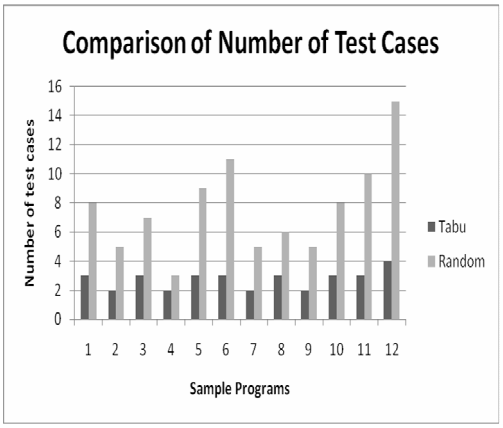


Fig. 5- Test Suite Size Comparison

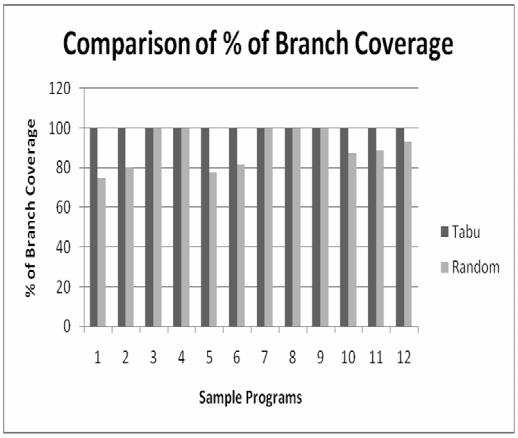


Fig. 6- Percentage of Branch Coverage Comparison

## References


[1] Chilenski, John Joseph Chilenski and Steven P. Miller (1994) *Software Engineering Journal*. 9 (5), 193-200.

[2] Edvardson J. (1999) *Proceedings of the second conference on computer science and engineering* 2 (1), 343-351.

[3] Eugenia Diaz, Javier Tuya, Raquel Blanco, Jose Javier Dolado, (2008) *Computers and Operations Research* 14(3), 38-69.

[4] Ferguson and Korel B. (1966) *ACMTOSEM,* 5, 63-86.

[5] Glover F. (1989) *ORSA Journal on Computing* 3(1), 190-206.

[6] Glover F. (1990) *ORSA Journal on Computing* 4(2), 4–32.

[7] Howden W.E. (1977) *IEEE Transactions on Software Engineering* 3(4), 266-278.

[8] John Clarke, Mark Harman, Bryan Jones, (2000) *IEEE Computer Society Press* 42(1), 247-254.

[9] Lindquist T.E. and Jenkins J.R. (1998) *IEEE Software* 5(1), 72-79.

[10] Lin Yeh, P.L. (2001) *Information Sciences* 4(13), 47-64.

[11] McCabe Tom, (1976) *IEEE Trans. Software Eng* 2(6), 308-320.

[12] McMinn p. (2004) *Journal on Software Testing, Verification, and Reliability* 14(2), 105-156.

[13] Raquel Blanco, Javier Tuya, Belarmino Adenso-Díaz (2009) *Information and Software Technology* 51(1), 708-720.

[14] Edvardsson J. (1999) In proceedings of 2nd conference on computer science and Engineering in Linkoping, 21-28.

[15] Voas J.M., Morell J. and Miller K.W. (1991) IEEE Transactions on Software Engineering vol. 8,41-48.

[16] Wegener Baresel DeMillo R.A., Offutt A.J. (1991) IEEE Transactions on Software Engineering Vol.17(9),900-910.

[17] Boris Beizer (2000) 'Software Testing Techniques',2ndedition, Dreamtech publisher, New Delhi.