Optimum Selection of GA Algorithm's Parameters for Software Test Data Generation

Sonam Kamboj[1] Mohinder Singh[2]
[1]M.Tech Scholar [2]Professor
[1,2]Department of Computer Science & Engineering
[1,2]Maharishi Vedvyas Engineering College.

Abstract— The proposed research work implemented and fine-tuned meta-heuristic based search algorithms namely GA algorithm for automatic test case generation using path testing criterion. The three parameters namely size of population, crossover rate and mutation rate have been chosen for setting GA algorithm for test data generation. For input generation, symbolic execution method has been used in which first, target path is selected from Control Flow Graph (CFG) of Software under Test (SUT) and then inputs are generated using search algorithms which can evaluate composite predicate corresponding to the target path true. We have experimented on two real world programs showing the applicability of these techniques in genuine testing environment. The algorithm is implemented using MATLAB programming environment. The performance of the algorithms is measured using average test cases generated per path (ATCPP) and average percentage coverage (APC) metrics. Experimentations found that crossover rate and mutation rates should be 80% and 5% respectively and population size can vary from 20 to 40 for best results.

Key words: Black box testing, White box testing, Dynamic execution , Symbolic execution, GA search algorithms , Metaheuristic Search Algorithms

I. INTRODUCTION

Test cases can be generated in several ways: automated, semi-automated or manual. While automated test cases generation saves a lot of resources and these don't encompass human biases and are less intelligent. On the other side manual test cases, although consist human intelligence, take much efforts and time. It is relatively easy to generate test cases manually which are able to detect many faults but at the same time it is very slow, boring and costly process. This process of automation requires a good amount of intelligence in order to handle propagation and constraint satisfaction either implicitly or explicitly [McMinn2004].

There are two basic approaches through which a program can be analyzed. First approach is called as functional testing in which program is treated as a black box and inputs are generated from input domain to cover all the functionalities/specifications of the program. Contrary to this, structural testing considered the program coded-behavior to generate inputs.

A. Black Box Testing:

Black box testing treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure. Or in other words the Test engineer need not know the internal working of the "Black box". This technique of testing is also termed testing to specifications, data-driven, behavioral, opaque-box, and closed-box, functional or input/output-driven testing. Black Box Testing focuses on the functionality part of the module. Although there are some bugs that cannot be found using only black box or only white box. If the test cases are extensive and the test inputs are also from a large sample space then it is always possible to find majority of the bugs through black box testing.

- Selection of input data: If a software unit (or module) is supposed to handle any integer between 1 and 25, then we assume that the chances of a fault being detected are equally good between 1 and 25. This range constitutes an equivalence class, i.e. a set of data such that any one member is as good a test value as any other. There are three equivalence classes that the unit must handle:

(1) <1
(2) 1…25
(3) >25

We conclude that three test inputs are adequate, that is, any one member of each of the three equivalence classes can be selected as a test input, e.g. -567, 1 and 2356.

However, a high payoff approach uses the programmer's experience that faults frequently exist at and on either side of the boundaries of the equivalence classes. Therefore the following test inputs should be used:

Input 1: 0
Input 2: 1
Input 3: 2
Input 4: 17
Input 5: 24
Input 6: 25
Input 7: 26

- Selection of output data: Similarly to the selection of input data, boundary conditions for outputs should also be taken into account. For example, if a library loan system computation results in the following outputs:
  Fine = 1.00 if the overdue period is 1...21 days
  Fine = 2.00 if the overdue period is >21

Test data should include data which not only partition valid and invalid inputs but also test the boundary conditions for the outputs.

In general, for each range R1 … R2 listed in either the input or the output specifications, five test cases should be selected, corresponding to:

less than R1
equal to R1
greater than R1 but less than R2
equal to R2
greater than R2

In the case where the specifications stipulate a precise value (e.g. a character string must be terminated by a carriage return), there are two test values, the specified value and any other value.

B. White Box Testing:

The other extreme is to focus on the internal structure of a software product to select test cases. Other names for this strategy are glass box, logic-driven or path-oriented testing. The most common form of white box testing requires that each possible path through the code is executed at least once. Consider the hypothetical flowchart below in figure 1.1.

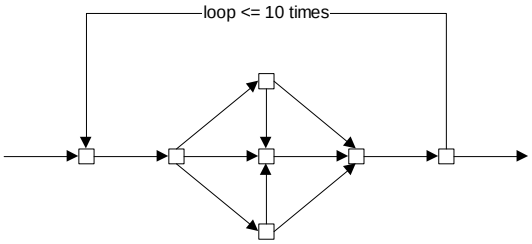

Fig. 1**:** Hypothetical Flow Chart

Although the flowchart appears trivial, there are over 12 million possible paths. There are five paths through the diamond. The total number of paths through the diagram is therefore:

$5^1 + 5^2 + 5^3 + \ldots + 5^{10} = 12207030$

C. Symbolic Execution:

The symbolic execution method also called as static execution, doesn't prescribe execution of the program under test with actual input values. In this method, a set of constraints (predicates) on the symbolic inputs is collected along an execution trace of the 'target path' (the complete path or portion of the path which needs to be tested) and then a constraint solver is used to assign values to the symbols that satisfy the symbolic constraints. These assigned values form a test case.

```
1. larger_of_two(in
   t a, int b){
2. if (a>b)
3.     printf("%d is
       greater than
       %d",a,b);
4. else
5. if(b>a)
6.     printf("%d
       is greater
       than
       %d",a,b);
       //error
       statement
7. else
8.     printf("%
       d is equal
       to
       %d",a,b);
9. }
```

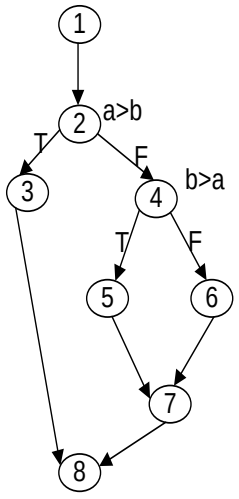

Fig. 2**:** A simple c procedure and corresponding control flow graph

To clearly understand the concept, let's take a simple Clanguage procedure as shown in figure 1.2, where the larger of two input values is printed. The corresponding control flow graph (CFG) of the procedure is also shown in figure 1.2. A CFG is a directed graph represents program's flow of control structures, i.e. order of execution, branching, and looping, where non-branch nodes stand for simple processing nodes and branch nodes represent predicates. Path traversal is depicted by arc between two nodes. If path is taking false branch (denoted by a 'F' on arc) of a node then predicate represented by the source branch node is negated before aggregating it with entire path constraint.

D. Dynamic Execution:

Another method of structural testing is dynamic execution based testing in which program is executed with the actual values of input variables. If two sets of input values invoke different program behavior then these form distinct test cases. The different program behavior depends on the test criterion. If the test criterion recommends the execution of all branches in the CFG then two distinct test cases if they execute different set of branches, form two distinct test cases. Similar is the case with other test criteria such as all-nodes, all-conditions and all-paths execution. In the above example, if one set of input, set1 is {a=10, b=12} and another set of input, set2 is {a=15, b=13} and test data is generated for fulfilling all-paths adequacy criterion then both sets form separate test cases as both execute different paths (set1 executes path 1-2-4-6-7-8 and set2 executes path 1-2-3-8) in execution trace.

E. Genetic Algorithm (GA) Search Algorithm:

GA algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and form a new population through repetitive application of mutation, crossover, inversion and selection operators. This is motivated by a hope, that the new population will be better than the old one.

F. Genetic Encoding:

A representation (genetic representation) of solutions to the program. For this an encoding method is used. A chromosome can be a binary string or a more elaborate data structure or real values can also be assigned.

G. Initial Population:

A way to create an initial population of individuals (chromosomes). The initial pool of chromosomes can be randomly produced or we can create it manually by using some seed values.

H. Fitness Function:

A measure which rates solutions in terms of their fitness, and a selection mechanism. The fitness function or objective function measures the suitability of a chromosome to meet a specified objective for which we are applying GA. Fitness is probably the main concept in Darwinian evolution.

I. The Genetic Operators:

There are four major operators in genetic operation; Selection, crossover, mutation and recombination operators. Selection operator based on which we decide which chromosomes will participate in the evolution stage of the genetic algorithm made up by the crossover and mutation operators.

II. RELATED STUDY

A. Random Test Data Generator[1]:

First class belongs to random testing based search algorithms. In this type of testing, random values are generated from domains of inputs and program is executed using these values. If these inputs are able to satisfy the testing criterion then they form a test case. A comprehensive overview of random testing is provided by Hamlet [Hamlet1994]. He concludes that misapplication of random testing is the major cause of its failure and poor performance. Duran and Ntafos [Duran1981] reported random testing to be satisfactory for small as well as large programs. Thayer et al [Thayer1978] used it to measure reliability of the system. Demillo et al [DeMillo1978] also used random testing for identifying seeded faults in programs. There exist also directions of research based on the idea of random testing, but which try to improve its performance by adding some guidance to the algorithm. This is the case for Adaptive Random Testing [Cen2004] and its recent extension to object-oriented software [Ciupa2006], and for quasi-random testing [Chen2005].

B. Algorithmic Test Data Generator[2]:

Another class of search methods used in testing is algorithmic such as numerical maximization techniques which were used by Miller and Spooner to optimize the test data [Miller1976], alternate variable method which was employed by Korel for its dynamic test data generator TESTGEN [Korel1989] and domain reduction procedure used by Demillo et al for its fault based test data generator named as GODZILA [Demillo1991]. Last decades experienced a number of research attempts in automation of test cases generation [Díaz2007, Duran1981, Harman2001, Jones 2005, Korel1990, Lin2001, McMinn2004, and Roper1997], specifically heuristic approaches were used to fulfill the requirements of this activity. Korel's technique is based upon the dynamic execution of software under test (SUT). On the other hand, Demillo et al used symbolic execution for testing purpose.

C. TESTGEN[3]:

Korel built upon the work of Miller and Spooner for automatic generation of test cases for Pascal programs. In Miller and Spooner method, a straight line version corresponding to target path of program is executed using test inputs and then numerical maximization techniques are used to optimize the solution. Fitness of candidate solutions is decided through an objective function, which is defined using constraints evolved from branch predicate of target path and was subsequently named as branch distance. It measured how close the branch predicate (to be evaluated) is to the true value. Korel also used the same branch distance based objective function with a one dimensional function minimization method known as alternating variable to propose a dynamic program execution technique for his path based automated test data generator "TESTGEN" [Kore1989].

D. GODZILA[4]:

Around the same time, Demillo and Offutt[Demillo1991] proposed a mutation analysis based testing system named as 'GODZILA' which is built upon the architecture of an already proposed 'Mothra testing system'[Demillo1989] to automate the generation of test data that approximates relative(mutation) adequacy criterion. Their strategy is also called as constraint based testing strategy. In this, first a path is selected from the program control flow graph for which test data need to be generated in order to execute the selected path itself. This method used symbolic execution testing method and can be used for both goal-oriented and full-path-oriented testing approaches. From the concerned path, a constraint system is derived by concatenation of all the branch node predicates and other statement constraints derived for a particular type of testing systems (such as in mutation analysis mutated statement is represented by a constraint). Branch node predicates are called reachability constraints and statements constraints are called necessity constraints.

E. Soft Computing Based Test Data Generator[5]:

Windisch et al [Windisch2007], have reported the application of particle swarm optimization (PSO) based technique for test data generation. They have conducted experiments to prove the usefulness and utility of search algorithm toward test case generation but did not fine tune the parameters of PSO algorithm for increasing the efficiency for test case generation.

Recently researchers have tried to merge some of the soft techniques in order to come out with new and efficient search algorithm for software testing requirement. Bin et al [Bin2007] have proposed a similar hybrid method where global search is provided by GA and local search is provided by SA. Arcuri et al [Arcuri2007] have applied memetic algorithm for test case generation in object oriented scenario, which have both type of search capabilities; local and global for testing requirement. Several authors in their experimentations have strived to establish the merit of soft computing based techniques over other techniques such as random testing, domain analysis etc. [Sthamar2007].

F. Hybrid Test Data Generator[6]:

Recently, the researchers have started exploring the use of hybrid techniques for automatic testing and solving several issues of symbolic execution such as the use of dynamically allocated data and scalability. Godefroid et al [Godefroid2005] used both types of testing strategies; concrete as well as symbolic (called concolic testing) in Directed Automated Random Testing (DART) algorithm for test generation of programs written in C using. The DART starts with a random (concrete) input and collects symbolic path constraints (conditions) during its execution and then uses these to attempt direct execution down an unexplored path on the next execution by negating a predicate in the path condition. Thus a new path condition is solved, generating the new set of test inputs. When execution reaches a branching statement the underlying decision procedure cannot decide, the symbolic condition is replaced by its concrete value. Randomization is also used when automated reasoning is not possible. The DART handles constraints only on integer types, but does not handle constraints on programs with pointers or complex data structures. For these cases, it only uses randomly generated data. Sen et al [Sen] proposed a concolic unit test generation

framework "CUTE" which is similar to DART but it uses a logical input mapping table for handling pointer variables also.

III. PROPOSED WORK

In this paper, we have worked in two areas, one on test case generation methodology and another on GA based search algorithm implementation for this purpose. Although there are so many different metaheuristic algorithms used for test data generation but till now no significant research has been taken for assessing optimum selection of various GA parameters for test data generation, hence we plan to propose to generate test cases using GA algorithm and find optimum value of various types of GA parameters such as population size, type of selection, crossover and mutation operators and probability for these operators for test data generation problem.
Objectives:

(1) To implement GA algorithm for software test data generation.
(2) To compare performance of various parameters of GA algorithm and thus select optimum values of parameters for test data generation.

A. Test Measures:

The performance of algorithms is evaluated using two parameters: Average percentage coverage (APC) and Average test case generation per path (ATCPP). The APC is used to measure effectiveness of test case generation process and efficiency of process is measured by the ATCPP. A good test data generation process will try to give 100% APC with less number of ATCPP. For each test object, we have measured average test case generation per path (ATCPP) and average percentage coverage (APC). ATCPP tells the level of effort a search algorithm has to make for test data generation and is taken by sum of test case generated for all paths divided by number of feasible paths. APC tells about the efficiency of test data generator and is calculated as fraction of paths covered. High figure of APC and low figure of ATCPP is desirable.

B. Test Objects:

We have taken two real world programs for test data generation activity. Some of these are frequently used by researchers. These are called test objects here and brief explanation for each test object is given below.

1) Triangle classifier (TC):
It is one of the most used programs for experimentation of test data generation in structural testing environment. It accepts three inputs as sides of a triangle and then decides whether these sides form a triangle and if yes then of what type.

2) 2.Line-rectangle classifier (LRC):
This program identifies whether a line cuts a rectangle or lies completely outside or lies completely inside of the rectangle. In this program total eight inputs are entered; four for co-ordinates of rectangle and other four inputs to define the line. Some of the nodes in CFG of this program have very high level of nesting. This is main reason of using this program so that the difficulty of testing nested structures can be found. Test cases for TC and LRC programs are generated from inputs by taking small domain of size 103 for each path.
Detail characteristics of these test objects are given in table below:

Name of Program	Lines of Code	Cyclomatic Complexity	Number of Decision Nodes	Highest Nesting Level	Total Paths in CFG	Feasible Paths	Singular equality Predicates Exist
TC	35	07	06	05	07	07	Yes
LRC	56	19	18	12	17	17	Yes

Table. 1: Test Object characteristics

IV. RESULTS

We have experimented on two most standard benchmark programs regularly used in testing research [Diaz2007, Lin2000]. These are triangle classifier and rectangle classifier programs. Table 6.4 lists the results of test cases generation GENETIC ALGORITHM method applied on two programs. Twenty seven mutant programs differ in selection of various parameters such as size of population (NIND), rate of crossover (CR) and rate of mutation (MR) for genetic algorithm. For TC program, we have considered two domains large and small. The large domain is of the order of -10^7 to $+10^7$ while the small domain is of the order of -10^3 to $+10^3$.

Table shows the experiments results. First and second column of each category programs register invalid average test cases per path (ATCPP) and average percentage coverage (APC) of all paths for both programs in 10 attempts of test case generation for each path.

	GA parameters			TC(Small domain)		TC(Large Domain)		RC	
S. No.	**NIND**	**CR**	**MR**	**ATCPP**	**APC**	**ATCPP**	**APC**	**ATCPP**	**APC**
1	20	0.9	0.04	2071	100	6961	73	734	99
2	20	0.9	0.03	2564	100	7841	63	929	99
3	20	0.9	0.05	**1474**	**100**	5279	**87**	745	99
4	20	0.85	0.04	1843	100	6427	72	**541**	100
5	20	0.85	0.03	2540	98	7528	65	584	100
6	20	0.85	0.05	**1561**	100	5314	**78**	**495**	100

7	20	0.8	0.04	2058	98	6211	76	641	99
8	20	0.8	0.03	2431	100	7002	64	673	99
9	20	0.8	0.05	1670	100	5642	73	563	100
10	30	0.9	0.04	1696	100	5612	70	766	99
11	30	0.9	0.03	2530	100	6972	63	784	97
12	30	0.9	0.05	1447	100	4733	88	519	100
13	30	0.85	0.04	1597	100	4981	82	578	100
14	30	0.85	0.03	2734	100	6795	64	689	99
15	30	0.85	0.05	1809	100	5240	78	567	100
16	30	0.8	0.04	1817	100	5039	78	610	99
17	30	0.8	0.03	2581	98	6432	68	631	99
18	30	0.8	0.05	1503	100	5285	85	482	100
19	40	0.9	0.04	1819	100	5522	82	510	100
20	40	0.9	0.03	2524	100	5935	74	747	99
21	40	0.9	0.05	1762	100	5619	78	563	100
22	40	0.85	0.04	1984	100	5713	78	624	100
23	40	0.85	0.03	2768	100	6530	67	678	99
24	40	0.85	0.05	1694	100	5743	75	543	100
25	40	0.8	0.04	1842	100	5938	72	674	100
26	40	0.8	0.03	2308	100	6943	70	698	99
27	40	0.8	0.05	1528	100	5637	78	547	100

Table. 2: Average Test cases generation per path (ATCPP) and Average percentage coverage with simple GA

Following results can be interfered from the table.

- GA is not able to find results frequently where input domain is very large and the path constraint contains equality based predicates. A further study is done to find out the cause and effect relationship between nature of predicates and efforts made toward generation of test data.
- From Sr. No. line 3, 6, 12 and 18, it can be clearly deduced that for different value of Number of Individual and crossover rate, the mutation rate MR of 5% gives the excellent results. So from experiments we can say that for test case generation problems we should choose mutation rate of 5%.
- If we compare performance of different values of CR, then value of 80% seems to give better result as compared to other value.

Number of individual or population size doesn't affect performance much but it should be between 20 to 40.

V. Conclusion

In this paper we have observed that GA performance greatly affected by mutation rate for test case generation problem. Ideal value of mutation rate can be 5%. Crossover rate also effects performance but to a lesser extent. 85% can be a safe value for this purpose. Number of individual or population size doesn't affect performance much but it should be between 20 to 40. It has also been found that the performance of the GA method is not good for path constraints which are having large number of equality predicate. Hence, it has been observed to be ideally suited for test case generation problem where less equality predicates has to be solved.

References

[1] Ahmed MA and Hermadi I, GA-based multiple paths test data generator. *Computers and Operations Research* (2007),(article in press)

[2] Alba E and Chicano F, Software Project Management with Gas. *Information Sciences*, 2007; 177(11):2380-2401

[3] Amoui M, Mirarab S, Ansari A and Lucas C, A Genetic Algorithm Approach to Design Evolution using Design PatternTransformation. International Journal of Information Technology and Intelligent Computing 1(2), 2006; 235-244.

[4] Ayari K, Bouktif S and Antoniol G, Automatic Mutation Test Input Data Generation via Ant Colony. *GECCO'07*, July 7–11, 2007, London, England, United Kingdom.

[5] Beizer B. *Software testing techniques*. 2nd ed., Dreamtech publication New Delhi. 1990.

[6] Burgess CJ and Lefley M, Can Genetic Programming Improve Software Effort Estimation? A Comparative Evaluation.,*Information & Software Technology*, 2001; 43(14):863-873

[7] Chong CS, Low MYH, Sivakumar AI and Gay KL, A Bee Colony Optimization Algorithm to Job Shop

Scheduling.Proceedings of the 37th Winter Simulation, Monterey, California, 1954-1961,2006.

[8] Clow B and White T, An evolutionary race: A comparison of genetic algorithms and particle swarm optimization fortraining neural networks. *In* Proceedings of the International Conference on Artificial Intelligence, IC-AI '04, Volume 2,pages 582–588. CSREA Press, 2004.

[9] Dahiya SS, Chhabra JK and Kumar S, Application of Particle Swarm Optimization Algorithm to Symbolic SoftwareTesting. *ADCOM 2009*, to be held in Bangalore on 14-17 December 2009. (Communicated for publication)

[10] Demillo RA, and Offutt AJ, Constraint-based automatic test data generation. *IEEE transaction on Software engineering*.1991; 17(9): 900-910

[11] DeMillo RA, Lipton RJ and Sayward FG, Hints on Test Data Selection: Help for the Practicing Programmer. *IEEEComputer*, 1978; II(4): 34-41.

