Design Multiple Path Test Data Generator Based on Meta-Heuristics

Dr. Yousif Basuony Elmahdy
Department of Computer Sciences, Faculty of Computer and Information Sciences, Assuit University, Assuit, Egypt
Dr. Abdelrahman Hedar
Department of Computer Sciences, Faculty of Computer and Information Sciences, Assuit University, Assuit, Egypt
Esraa Farouk Abou-Elmagd
Department of Computer Sciences, Faculty of Computer and Information Sciences, Assuit University, Assuit, Egypt

Abstract- **The complexity of software systems has been increasing dramatically in the past decade. Software testing as a labor-intensive component is becoming more and more expensive. Specifically, testing costs account for up to 50% of the total expense of software development. Searching software errors, by using evolution based methods like genetic algorithms and other meta-heuristics is one attempt towards these goals. The most laborious part of software testing is the generation of test data. Currently, this process is principally manual process. The automation of test data generation can significantly cut the total cost of the software testing and the software development cycle in general. Test data generation can roughly account for 40% of the total cost of software testing. The automation will increase software reliability which will boost the developers' and customers' confidence in the software system being developed. Existing approaches of automatic test data generation have achieved some success by using evolutionary computation algorithms. This paper uses genetic algorithm to automate test data generation which hope to reach the desired goal more efficiently.**

Keywords - Genetic Algorithm, Automatic Test Data Generation, Genetic Algorithm Operators, Software Testing.

I. INTRODUCTION

Computers and programs have been expanded in many sections of our life. Software testing is a way for verifying the correctness and appropriateness of a software system for ensuring that a program meets its specifications. While software testing is very significant, it is also very expensive, a laborious and time-consuming work. The goal of software testing is to generate a set of minimal number of test cases such that it reveals as many defects as possible by satisfying a particular criteria called test adequacy criteria e.g. path coverage. Criteria called test adequacy criteria e.g. path coverage. Automated software testing can significantly reduce the cost of developing software. Testing is defined as the process of executing a program with the intent of finding errors. A test Case is a pair of input and its expected output. Good test case is one that has a high probability of detecting an undiscovered error (path coverage). Researchers have shown the suitability of using evolutionary computations in software testing. One of these computations is genetic algorithm (GA). The GA-based test data generator can generate only one test input case at a time. This means that any of the existing test data generators should be used more than one time. So, some of the required test data can be readily available when trying to find other test data. All this makes the existing test data generators inefficient. In this paper, we are concerned with the problem of taking an arbitrary program and automatically generating test data to achieve a certain level of coverage of the program. This paper uses the ideas of Genetic Algorithm (GA) to automatically develop a set of test data to achieve a level of coverage. The goal of this paper is to present GA based test data generator that is capable of generating multiple test data to cover multiple target paths. The remainder of the paper is organized as follows. Section 2 gives a background on software testing. Section 3 discusses the GA-based test data generator using standard (GA). Section 4 discusses our GA-based multiple path test data generator. Section 5 discusses the experiments to validate the approach. Section 6 discusses comparison between our work and other works. Finally, section 7 for references.

II. SOFTWARE TESTING

Software testing is one of the significant components of software system with many complex and inter-related constraints. The quality of a software system is primarily determined by the quality of the software process that produced it. It has been estimated that an error caught during the system specification phase may be about 50 times cheaper to repair than an error not detected until in the system testing phase. The definition of testing according to IEEE/ANSI standard is: "The process of operating a system or component under specified conditions observing or recording the results, and making evaluation of some aspect of the system or component". The definition of software testing according to IEEE/ANSI standard is: "The process of analyzing a software item to detect the difference between existing and required conditions and evaluate the features of the software items". Software testing techniques are classified into two categories: static analysis and dynamic testing. In static analysis, a code reviewer reads the source code of the programmer software under test, statement by statement, and visually follows the program logic flow by feeding an input. This type of testing is highly dependent on the reviewers' experience. Dynamic

testing techniques execute the program under test on test input data and observe its output. Dynamic testing can be classified into two categories: black-box testing and white-box testing. White-box testing is concerned with the degree to which test cases exercise or cover the logic flow of the program. This type of testing is also known as logic-coverage testing or structural testing because it considers the structure of the program. Black-box testing tests the functionalities of the software irrespective of its structure. It is interested only in verifying the output in response to given input data. Adequacy of logic-coverage testing can be judged using different criteria: statement, decision, condition, decision/condition, multiple condition coverage and path coverage. A stronger criterion is known as decision or branch coverage. It requires that each decision has a True and False outcome at least once. The strength of this type is that it takes on all possible outcomes at least once. The weakness of this type is certain conditions mask other conditions. Path coverage test adequacy criterion is concerned with the execution of (selected) paths in the program.

III. GA-BASED TEST DATA GENERATOR

A variety of methods to develop test data generators have processed over the years. Each method was meant to satisfy a certain test adequacy criterion, and conform to a desired testing objective. In the early age of automation of software testing, most of the test data generators were using gradient descent algorithm. These algorithms are inefficient and time-consuming, and could not escape from local optima in the search space of possible input data. Meta-heuristic search algorithms have been employed in test data generators as a better alternative. A small number of test data techniques have already been automated random, structural or path-oriented, goal-oriented, analysis-oriented test data generators. The limitations of these approaches have stopped their general acceptance. Random generators create large amounts of test data because no information exists about the testing objectives; the generators often fail to find data that satisfy the stated objectives of the testing process. A structural or path-oriented generator first identifies a path for which test data is to be generated. The path is often infeasible causing the generator to fail to find an input that will traverse the path. Analysis-oriented generators have the ability to generate high quality test-data, but rely upon their designer having a great insight upon the domain of operation, and hence are not readily general is able to arbitrary software systems. Goal-oriented generators provide an industrial- strength solution. Initial work with goal-oriented approaches has achieved some limited success producing high quality test-data for small programs. The process of automatic test data generation using GA has two main steps:

1- Instrumentation: it is the process of inserting probes (tags) at the beginning/ending of every block of code e.g. at the beginning of each function, before and after the true and false outcomes of each condition. These tags are used to monitor the traversed path within the program while it is being executed with certain test input data.
2- test data evolution: this is a loop where the program is executed with some initial input data, feedback is collected and the input data is adjusted until satisfactory criteria are achieved.

The feedback is sort of a fitness value assigned to the current input data to reflect its appropriateness according to the given test criteria. The basic steps of GA are shown in Fig1.Before GA can be used , these are four domain-dependent things to do : defining genetic representation of the problem solutions , defining the fitness function , selecting chromosome selection methods , and defining genetic operators[cross over , mutation]. Pei et al. in 1994 [4] observed that most of the test data generators were using symbolic evaluation. They developed a single-path- coverage test data generator that employs GA. Roper et al. in 1995[8] developed a GA-based test data generator that has an aim to traverse all the branches within a target program. Jones et al. in 1995[3] developed GA-based test data generator to achieve branch coverage by using a sequence of binary strings for individual representation. In this paper, an approach satisfies the path coverage. The work tries to generate multiple test data to cover multiple target paths at one run.

IV. GA-BASED MULTIPLE PATH TEST DATA GENERATOR

Many GA-based test data generators adopt statement or branch coverage. However, path coverage criterion covers statement and branch coverage criteria. A more effective software structural testing should have path coverage as its objective. This work focuses covering a set of target paths in a single run of GA covering multiple paths in one run would require incorporating these paths within the fitness calculation. Accordingly, trying to satisfy multiple paths at a time is expected to greatly increase the effectiveness and efficiency of the test data generator. In section 4.1, we discuss the genetic algorithm. In section 4.2, we discuss the structure of problem. In section 4.3, we discuss the representation chosen to the chromosome. In section 4.4, we discuss the fitness function design. In section 4.5, we discuss the selection criteria. In section 4.6, we discuss the cross over operator design. In section 4.7, we discuss the mutation operator design.

A. Genetic Algorithm Design

Genetic algorithms are applied to a verity of problems involving search and optimization within the AI domain. The principle of GA is that they create and maintain a population of individuals represented by chromosomes. These chromosomes are typically encoded a solution to a problem. These chromosomes processed according to the rules of selection, cross over and mutation. Each chromosome in population has a measure of its fitness in the population. This fitness value indicates how successful the chromosome is as a

solution to the problem. The selection process selects chromosomes with high fitness values. The new population is derived from these selected chromosomes after processing by crossover and mutation processes. The cross over process is analogous to the process of sexual reproduction which involves two chromosomes swapping chunks of data (genetic information). Mutation process introduces slight changes into a small proportion in the population. The pseudo code for a simple GA is in Fig2.The algorithm will iterate until the population has evolved to from a solution to the problem, or until a maximum number of iterations have taken place.

B. The Structure of the Problem

The problem represented as a tree. Each node in the tree represents condition that has two values (True, False).From Fig3.To calculate the number of paths in these program, we use the edge-matrix technique [m, m] where m is the number of nodes in the source program (conditions).Because the paths are in sequence order, we replace the 1 by 2. After that we multiply the diagonal. In this example, the number of paths is 8, Fig5.From Fig4. In the edge-matrix, the first one in the second row indicates that the second condition is dependent on the first condition. In the case of dependent nodes, we replace 1 by 3 in the edge-matrix, Fig5.

C. The Representation of Chromosomes

The representation of the chromosomes (individuals) can affect the performance of GA-based test data generator. The most common representation is the binary string which represents the node visited by this chromosome. The chromosome represented as a test input and genes of it represent the nodes and if the node is visited or not [one chromosome equal one path]. The number of genes equal to the number of nodes. In this work, the representation of chromosome chosen as a string of test data where each one represents one path and the number of test data equals to the number of paths generated. This way of representation increases the range of testing which as a result will increase the reliability and trust in the test data generator. One of the properties of this new representation is the mask of chromosome. This mask represents the paths covered by this chromosome. For example, if we have two input variables x and y and one condition in a node is (x < y). Table1 shows a chromosome with the previous condition. The value 1 indicates that the first path is covered. The two values 0 indicate that the second and the third paths are not covered by these chromosomes. Our target in this work is to make the mask of the chromosome ones which indicate that all paths are covered. In this case, this chromosome is a good (required) solution to the problem.

D. The Fitness Function Design

The fitness expresses how good the solution of the chromosome is in relation to the global optimum. This kind of values will be calculated for each chromosome in each population by a fitness function. The fitness value is used to compare the individuals and to differentiate their performance. An individual which is near an optimum solution gets a higher fitness value than an individual which is far away. A difficult issue in using GA is often the attempt to find a suitable fitness function which calculates the fitness value and expresses the problem as well as possible. In this work, we choose the fitness function as follows

Fitness value (chromosome)
= number of node visited + PU (chromosome) (1)
PU (chromosome)
= ∑ (Probability (Gene) + Utilization (Gene)) (2)
Probability (Gene)
= number of nodes visited/number of all nodes (3)
Utilization (Gene)
= number of these path occurrence/number of all tests (4)

The fitness function composes of two parts: the first part is the number of nodes visited by this chromosome (sum of nodes visited by all paths in this chromosome without redundancy), the second part is the sum of gene probability and gene utilization for all paths. The gene probability calculated from giving each gene a value representing its importance corresponding to the others. The gene utilization calculated from the ratio of using this gene (path) in the test cycle.

E. The Design of the Selection Operator

The selection operator chooses two individuals from a generation to become parents for the recombination process (cross over and mutation). There are different methods of selecting individuals, e.g. randomly or with regard to their fitness value. The random selection chooses the two parents randomly from the population to generate a new population without knowing any information about the fitness of these parents and if they are good solution or not. This method gives the same chance to all chromosomes. This method would guarantee diversity and healthy mating. If the individuals are selected with regard to their fitness value, this guarantees that the chromosomes with a higher fitness value have a higher likelihood of being selected; the others are more likely to be discarded. Using this method, the chromosome with the highest fitness value has to be selected. The selection method used in these work is linear ranking. For linear ranking selection, the individuals are sorted according to their fitness values and the rank N is assigned to the best individual and the rank 1 is assigned to the worst individual. The selection probability is linearly assigned to the individuals according to their rank:

$$Pi = (1/N)(\eta- + (\eta+ -\eta-) ((i-1)/(N-1))); \quad i€\{1, 2,\ldots, N\} \qquad (5)$$

Here η-/N is the probability of the worst individual to be selected and η|+/N is the probability of the best individual to be selected. As the population size is held constant, the conditions η|+ = 2- η|- and η|- >=0 must be fulfilled. Note that all individuals get a different rank (different selection probability), even if they have the same fitness value. The probability is determined the gradient of linear function. η-= 2/rm+1 and η+ = 2rm/rm+1. The algorithm of linear ranking selection is shown in Fig 6. The gradient of the linear

function rm has a value 1.7 as it must satisfy 1<= rm <= 2 and it is a good value used in linear ranking up to now.

F. The Design Of Cross Over Operator

Cross over operates at the individual level. The two parents (chromosomes) exchange substring information at a random position in the chromosome to produce two new strings representing offspring. The crossover operators search for better within the genetic material. The objective is to create better individual and as a result create better population. The cross over operator used in these work depend on a new idea. After selecting the two parents, the best one from the two individuals will be survive. The best one is determined by the number of paths covered by this chromosome. The other one becomes the result of an OR-process on the masks of the two parents and then reorder the paths in these chromosomes. For example, these are two chromosomes' masks (which are chosen as parents) as shown in Table2. As shown in the previous example, the second chromosome survived because it covers three paths but the first one cover two paths only. The new first chromosome is the result of the process (11000 OR 01101) is (11101) which cover four paths.

G. The Design of Mutation Operator

Mutation is the occasional random alteration of a bit value which alters some features with unpredictable consequences. Mutation is used to maintain diversity in the population and to keep the population from converging on one solution. The mutation operator used in these work depend on a new idea. After cross over, the new chromosomes mutate by choosing the paths that are not covered in two chromosomes and try for four or five times to be covered. This is done by giving a new input data and retests the path. If we find a covered path, replace it with the old one. If the coved path is not found, remain it not covered. For example, in the previous example, the two chromosomes after crossover become as shown in Table3. The first path in the second chromosome and the fourth path in the two chromosomes are not covered. If we found a covered fourth path, the two chromosomes become as shown in Table4. This means that we find a solution to the problem which is represented by the first chromosome. The test stops in these case and prints the solution. But if we don't find a covered fourth path, we will try again. If we found a first covered path, the two chromosomes will become as shown in Table5.

V. THE EXPERIMENTS

In this section, we present and asses the performance of our work. We present experimental results and analysis.

A. Experiments Design

We have four different experiments. Each experiment is comprised of a set of three runs. We assess the performance of linear ranking selection, new cross over operator and new mutation operator. We use three programs which are suitable for this problem like the maximum-number, the temperature measurement and a small test program used to indicate all the steps of the multiple test data generator.

B. GA Parameters Setup

The values of GA parameters are shown in Table6.These parameters vary from one experiment to another.

C. Extended CFG (Context Free Grammar)

The language used in the source program has the rules in Fig7.

D. The Standard GA

In the first, we test the standard GA rules on the three programs. The standard GA uses random selection, one-point cross over and random mutation. The results are shown in Table7.

E. The New Selection Method

The result of using a linear ranking selection instead of random selection is shown in Table8.

F. The New Cross over Method

The result of using our new cross over method instead of the one-point cross over is shown in Table9.

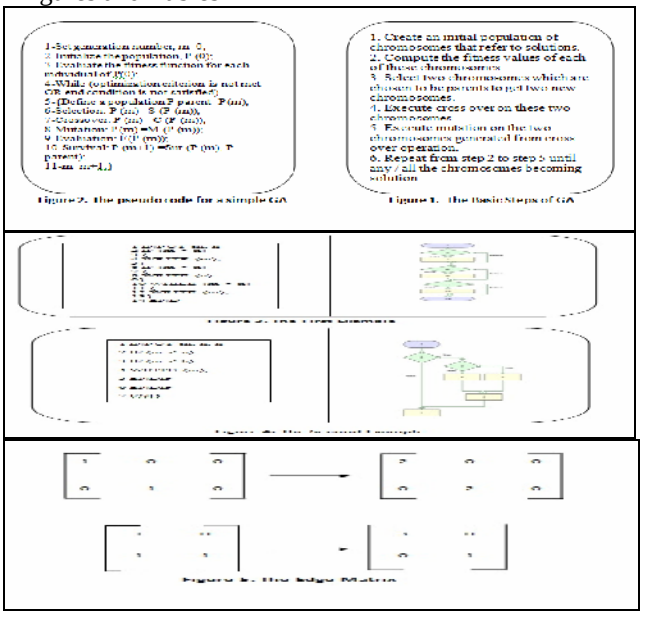
G. The New Mutation Method

The result of using our new mutation method instead of the random mutation is shown in Table10.

H. The Final Experiment

Finally, we increase the chance of using mutation chances to 10 to get new covered paths. This means that we doubled the chance of this method. The result of using new mutation is shown in Table11.In the first program, we find a solution to the problem as all paths covered. In the second program, the number of individuals reached to this ratio of coverage increased. In the third program, the number of covered paths is, as shown in all previous table, the best results at all.

I. Figures and Tables



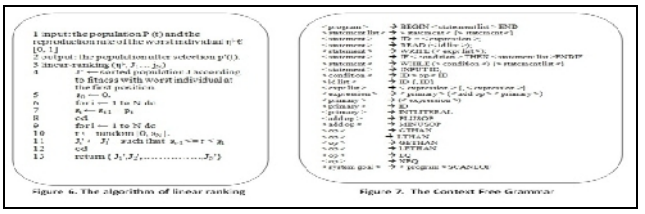


Figure 6. The algorithm of linear ranking

Figure 7. The Context Free Grammar

TABLE 1. Chromosome Example

Chromosome	*X=10 Y=11*	*X=15 Y=9*	*X=13 Y=1*
Mask	1	0	0

Table 2. The two parents

Chromosome1	*1*	*1*	*0*	*0*	*0*
Chromosome2	0	1	1	0	1

Table 3. The Two Chromosomes After Cross Over

Chromosome1	*1*	*1*	*1*	*0*	*1*
Chromosome2	0	1	1	0	1

Table 4. The Two Chromosomes if the fourth path covered

Chromosome1	*1*	*1*	*1*	*1*	*1*
Chromosome2	0	1	1	1	1

Table 5. The Two Chromosomes if the first path covered

Chromosome1	*1*	*1*	*1*	*0*	*1*
Chromosome2	1	1	1	0	1

Table 6. GA Parameters Setup

Number	*parameter*	*Values*
1	Population Size	6
2	Number of Generations	10
3	Selection Method	Linear Ranking
4	Chromosome Size	Number of Paths
5	Gradient of Linear Function	1.7
6	Number of Tests	5

Table 7. Standard GA

Programs	*Number of covered paths*	*Number of all paths*
Maximum Number Program	2	4
Temperature Measure Program	1	4
Own Test Program	3	16

Table 8. Linear Ranking Selection

programs	*Number of covered paths*	*Number of all paths*
Maximum Number Program	3	4
Temperature Measure Program	2	4
Own Test Program	3	16

Table 9. New Cross Over method

Programs	*Number of covered paths*	*Number of all paths*
Maximum Number Program	3	4
Temperature Measure Program	2	4
Own Test Program	5	16

Table 10. New Mutation method

Programs	*Number of covered paths*	*Number of all paths*
Maximum Number Program	3	4
Temperature Measure Program	3	4
Own Test Program	9	16

Table 11. Final Experiment

Programs	*Number of covered paths*	*Number of all paths*
Maximum Number Program	4	4
Temperature Measure Program	3	4
Own Test Program	11	16

Table 12.Comparsion between our work and Moataz's work

Programs	*Number of covered paths (our work)*	*Number of invisible paths (our work)*	*Number of covered paths (Moataz's work)*	*Number of invisible paths (Moataz's work)*
Binary search	7	1	7	0
Bubble sort	25	20	14	11

VI. COMPARISON WITH OTHER WORKS

Finally, after testing our algorithm on some programs, we will compare its results to the results of Moataz's work[1].The results are shown in Table 12.

REFERENCES

[1] Moataz A.Ahmed, Irman Hermadi. GA-based multiple paths test data generator.2007.

[2] Myers GJ. The art of software testing. New York: Wiley; 1979.

[3] Jones B, Sthamer H, Eyres D. *Automatic structural testing using genetic algorithms*. Software Engineering Journal 1996 ; 11(5): 299-306.

[4] Pei M, Goodman ED, Gao Z , Zhong K. *Automated software test data generation using a genetic algorithm*. Technical Report, GARAGE of Michigan State University ; June 1994.

[5] Harmen, Hinrich Sthamer. *The automatic generation of software test data using genetic algorithms*; November 1995.

[6] James Miller, Marek Reformar, Howard Zhang. *Automatic test data generation using genetic algorithm and program dependence graphs*; Canada ; November 2005.

[7] C.C.Michael, G.McGraw, M.Schatz. *Generating software test data by evolution*, IEEE transactions on software engineering ; 2001.

[8] Marc Roper, Iain Maclean, Andrew Brooks, James Miller and Murray Wood. *Genetic algorithms and the automatic generation of test data*.

[9] Bogdan Korel. *Automated software test data generation*. IEEE transactions on software engineering, August 1990.

[10] Baowen XU, Xiaoyuan Xie, Liang Shi, and Changhai Nie. *Application of genetic algorithms in software testing*; 2007.