10.17932/ IAU.IJEMME.m.21460604.2015.5/4.1035-1042

A GA based Software Test Data to Generator Suitable Test Cases

Ali NOROUZI[1]

Abstract
This work presents a new concept that is applied in genetic algorithm based tester. Genetic algorithm is sort of evolutionary algorithm that uses in searching problem for optimal solution. In presented test input data generation application, the solution sought by genetic algorithm is a set of test data that causes execution of all possible paths of a given program under test. The proposed concept, repetition frequency is a tool to measure high frequented and also dead parts of program. To experiment with proposed genetic generator, random test data generator was implemented too. Both of these generators tested three programs that are benchmark for many researchers in order to show our proposed tester efficiency.

Keywords: *Software engineering, genetic algorithm, dynamic repetition frequency, test data generator.*

1. Introduction

Nowadays, technology incorporates software systems which become complicated over and over. These systems applied in several fields such as business and government that their failure is high cost, hence distributed system has most troublesome problems, due to any component interacts in unpredictable ways.

The study is part of continuous research stream focused on the application of Genetic Algorithm (GA) in structural testing. We intend to discover bugs and errors through generating large volumes of test data. Generally, the genetic algorithm is a process which explores different execution of a program to find misbehaviors. Several bugs are discovered after many generations. This is how, genetic algorithm based methods can obtain solution which show rare error.

GA based testing is an optimization technique used in test case generation to find optimum solution. The considered test is characterized by the use of Evaluation Testing, where input domain of the test data evaluates respected search aim. Traditionally, GA operates on binary string but other coding is possible. In this case, each digit is represented by a gen that makes up chromosomes. A collection of chromosomes make up population. Each has fitness value that determines probability of

1 Department of Computer Engineering, Istanbul University, norouzi@ieee.org

presence in the next generation as parent. In fact, fitness value is a assigned weight to each chromosome based on its content pattern. After, the next generation is attained from the percentage of current population in which mainly chromosomes are mated and rest of the percentage is mutated. This cycle is terminated whenever stop criterion is satisfied.

Generally, evolutionary testing including GA based one are inspired by evolutionary biology such as mutation, selection, crossover, and inheritance. The study focuses on selection process a.k.a. fitness function which adopts robust and suitable chromosomes which have fitness value on or above specified threshold value. Therefore GA based testing is categorized as adoptive search technique where are not guaranteed obtaining of optimal solution. However, GA based operator with effective fitness function finds very good solution in a limit of time and definitely improves the individuals over several generation iteration, according to the Scheme theorem [4]. The article is arranged as follows. Main idea in software test is the subject of section2, followed new algorithm is under consideration in section 3. Section 4 deals with the results of simulation and finally come conclusion and future work suggestions at section 5.

2. Main Idea

Path testing is a comprehensive structural testing where source code of program can be represented by a directed graph called Control Flow Graph (CFG). In this graph, every statement and possible control flow between statements are shown as node and edge respectively and therefore path is a sequence of node which is limited between two special node called start and exit node.

This kind of testing is designed to execute all paths of CFG, i.e. every statement and every condition which has two paths in true and false sides, are traversed at least once time[1, 3]. In presented approach, GA based testing is an application of genetic algorithm to the path testing where utmost paths of program are covered by generated test data.

There are two fundamental strategies for improving GA based test data generation to test software. The first one is direct improvement of genetic algorithm components. Selection component is one the critical process in the GA operators where individuals are adopted according to their weight or other qualifications. The second important component is stop process which diagnoses suitable time to stop.

The second strategy is applying a system oracle, models or simulation as a substitute to the actual system. Any improvement to the genetic algorithm component could significantly effect on the solution qualification and execution time.

3. New Algorithm

Formula 1 is proposed to evaluate each test case, applying dynamic repetition frequency concept. Given a CFG of program, each edge which is represented by a character has repetition number in traversed paths collection called dynamic repetition frequency. The more dynamic repetition

frequency of every edge, the more coverage probability

This phenomenon causes to decrease own fitness value, i.e. the fitness value of a path is determined by summation of inversed dynamic repetition frequency or briefly repetition frequency number of edges included. Clearly, fitness value of repeated edge of a path (especially with cycle) will be 0 when it is traversed by a test data. In actual, the fitness function implies that the aim is to traverse low repetition frequency edges.

$$Fitness(p_j) = \sum_{1}^{n} (1/f_j) \quad (1)$$

In this approach, Fitness Function is a procedure which assigns a weight to each chromosome based on covered edges. This weight is called fitness value and is a criterion to select chromosome from current population to generate new population.

As fitness function has rationale role in the GA based generator and biases candidate data toward optimum ones, we introduce a function which is explained below, according to the repetition frequency concept:

(1) Form CFG of program and determine every frequency edges, (2) determine branches and then find conditional statements related to those branches in the program, (3) form predicate according to the distinct statements of previous step, using conjunction (AND) and disjunction (OR) operators. We apply disjunction operator when you have composite conditions, such as IF and CASE conditions which simultaneous execution of all parts of a condition with one test data is impossible. For example, we use disjunction operator between THEN and ELSE statements of an IF.

We also apply conjunction operator to remained statements, :(4) determine fitness function: 1- we now manipulate remained predications similar to previous step so that replace any conjunction and disjunction operators with minimum and maximum ones in order, and 2- we finally replace every condition statement with corresponding function in Fig. 1. Final function will be fitness function. Fig. 1 represents fitness function formation formula [6, 7, 8]. Suppose 5th function. If difference between x and y is equal to zero, the function will return zero, unless it remains the real result. Consider third function. We use very small value ε (e.g. 10^{-6}) to distinguish between x $\neq$ y and x=y state.

$$if\,(x).....\Rightarrow \xi = \begin{cases} 0 & True \\ K & False \end{cases}$$

$$if\,(x == y)....\Rightarrow \xi = \begin{cases} 0 & x = y \\ abs(x-y) & x \neq y \end{cases}$$

$$if\,(x <> y)....\Rightarrow \xi = \begin{cases} 0 & x \neq y \\ K & x = y \end{cases}$$

$$if\,(x < y).....\Rightarrow \xi = \begin{cases} 0 & x < y \\ (x-y)+\varepsilon & x \geq y \end{cases}$$

$$if\,(x \leq y).....\Rightarrow \xi = \begin{cases} 0 & x \leq y \\ (x-y) & x > y \end{cases}$$

Figure 1: Standard fitness function for conditional relations

3.1. GA-based Solution Pseudo code

Fig. 3 represents psudocode where steps 4 through 8 indicate genetic algorithm. In first step, using Fig. 2 which represents standard fitness functions, we make a fitness function and calculate binary of decimal numbers which is generated by user or random function as candidate (primary) chromosomes. We identify edges of program CFG to calculate the repetition frequency of edges while tester gets running. These edges are mentioned in coverage table that we initialize with primary chromosomes. Next step indicates main cycle of generation.

```
Procedure GC ( )
{
Input: Program: Changes version of program to be tested;
InitData: Set of test data; EF: Edge Frequency;
Output: Final: A solution test case set;
CovTable: recorded EFs with status;
#define MaxTimes m; //Max acceptable time;
Variables declaration:
CanCH1&2: Candidate chromosomes;
TP:Traversed Path;
CovTable: Coverage Table;
NextPop, CurPop: a set of test data;
OpCH1&2: Optimal Chromosomes;
Counter: iteration;
   Begin
Step1: Make CanCH 1&2 by InitData;
        Get fitnessFUN() to Initial OpCH1 and CovTable;
        Initial CurPop;
Step2: While (! fill CovTable with Y || counter <
        MaxTimes ||! User request) {
        Use Crossover and Mutation operations;
        Compute fitnessFUN ();
        Compute NextPop and Save OpCH2;
Step3: for each chromosome of NextPop
        If (IsDefect (NextPop))
                Replace with OpCH1 one;
        CurPop = NextPop; OpCH1 = OpCH2;
Step4:if(counter mod 10 == 0){
        Compute number of TPs by CovTable;
        Compute EF of every edge in CovTable;
        Show CovTable and Ask to continue;} Counter++;
        } Final = CurPop;
        Return Final and CovTable;
    End.
}
Boolean IsDefect (chromosome)
{       v = Fitness value of best OPCH1;
        if fitness value is less than v/l return true;
        else return false; //l is an optional value
}
```

Figure 2: Proposed GA based-Algorithm

Step 2, the algorithm calls crossover and sometimes mutation to generate new population. In fact, crossover generates offspring and mostly gets two chromosomes and remains two new ones. In next line, fitness function evaluates each

chromosomes based on its dynamic repetition frequency of coverage table.

The more fitness value, the more chance to participate in the next generation

Since some suitable chromosomes probably will not present in the next generation, we save them in OPCH2 variable, i.e. however suitable chromosomes is saved or presented in the next generation but less fitness value one just can be mutated and the others will be removed from population.

Step 3: if fitness value of some chromosome is less than threshold, it will be removed from current population because it has been corrupted, i.e. it is not enough qualified to traverse some uncovered part. This chromosome will be replaced with one from OPCH2 is being completed from previous population.

In the last step, we compute required iteration and update coverage table too in order to present it to the user. Algorithm can stop the process based on user request. This step is controlled by counter variable that controls number of iteration.

4. Discussion of Results

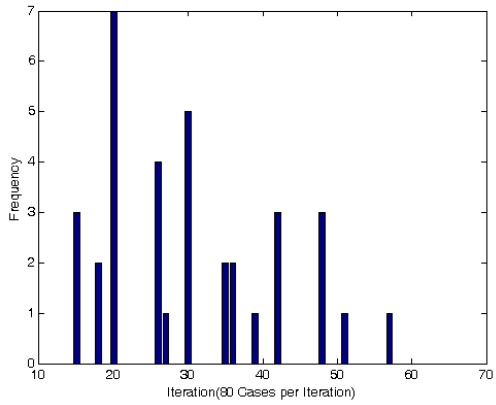
To investigate the efficiency of proposed algorithm, we experimented with some programs such as triangle classifier. These programs are benchmark for many researchers in software testing [2, 5, 10]. Below, we explain briefly these programs.

Find.c: Lines of code: 66, Given array P[], and index I, places all elements less than or equal to P[I] to the left of P[I], and all elements that are greater to or equal to P[I] to the right of P[I]. [2]

Fourballs.c: Lines of code: 82. Given four integers representing the weights of balls, determines the weights of the balls relative to each other. [2]

Triangles-classifier.c: Lines of code: 61. Given three integers representing triangular side lengths subdivide into valid, equilateral, isosceles and scalene classes.

In the first step, the fitness function was supposed to operates based on presented algorithm. These Figurs show the average of iteration by our tester and random one to cover the program under the test. The population was 500 individuals with mutation rate .5 in two random and presented methods.



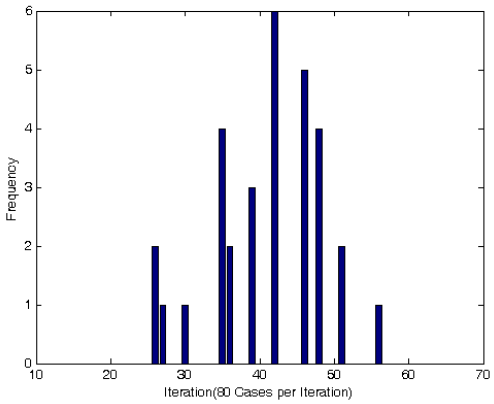


Figure 3. *Runs of proposed and Random tester for Find.c*

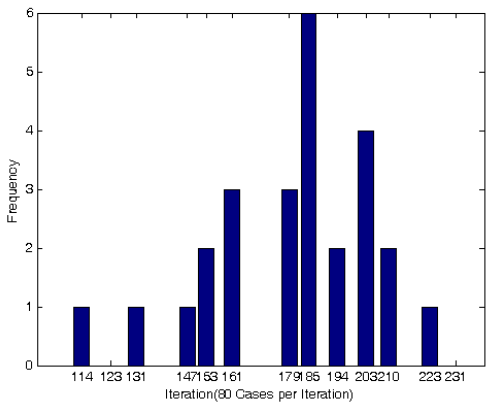
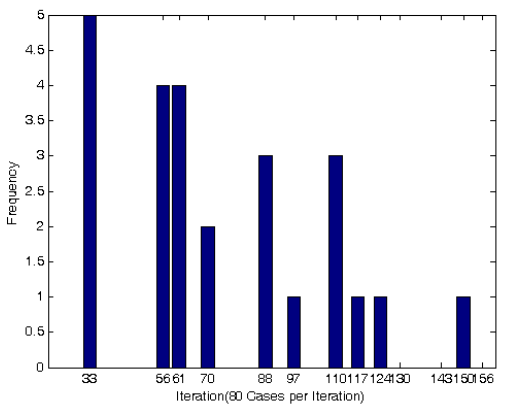
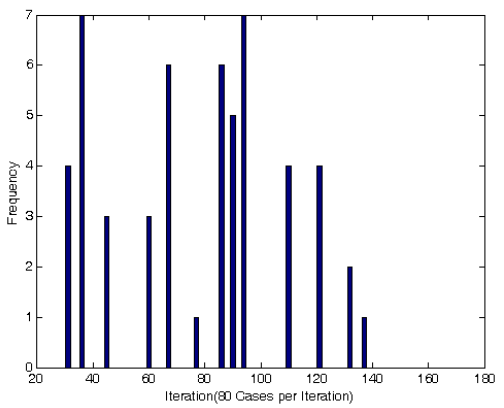


Figure 4. *Runs of proposed and Random tester for Triangles-classider.c*

Fig. 3 summarizes graphically the results of experiment with Find.c-proposed tester on the top and Random is on the bottom. In each diagram, the horizontal axis shows the number of generation required by a run. The left vertical axis gives the frequency that such a random occurred. As it's shown, seven of total runs of proposed tester required 20 iterations to achieve full coverage while this mount is more than 40 iterations for random tester.

Fig. 4 shows that random generator tested all of possible path of execution of program by 185 iterations while this mount was reduced to 33 generations for our tester. Obviously, more complicated program needs more time and iterations to test entirely. This increase grows fewer for our tester rather than random, because proposed tester trains input data to cover possible parts of program.





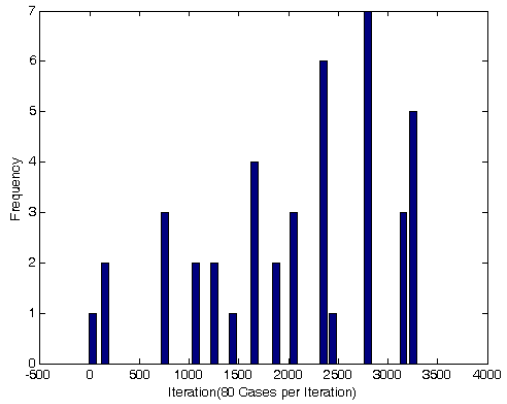


Figure 5. *Runs of proposed and Random tester for Four balls*

These differences for Four balls are significant. By comparison, Random could not achieve 100% coverage with fewer than 2700 iterations. More than half of the Random runs required 2600 or more iterations to achieve full coverage.

The experiments show that our proposed tester outperforms random generator over a number of runs dramatically for source code of program with high complexity. Our tester performs better than random one especially when the source code of program contains nested conditions or loops that are difficult to satisfy.

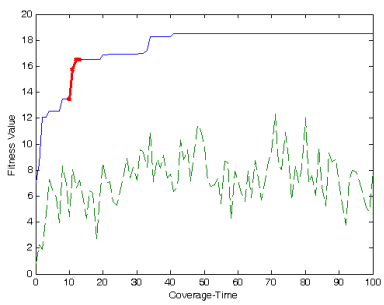


Figure 6. *Total duration coverage time for Random and proposed tester*

Fig. 6 shows the total duration coverage of several mentioned programs measured in second. When the diagram gets straight line, the red points shows that our stopping criterion is satisfied. Consider green line, it shows classical GA-tester. Clearly, the green line makes up disorder lines. It shows that the tester can't lead the generated data, i.e. the data generates entirely stochastically, regardless of kinds of source codes and uncovered parts. While our tester has ascending manner, i.e. our tester can remember whatever has traversed and lead the new generated test data based on previous records.

5. Conclusion

This article investigates the performance of the proposed GA-based tester based on program path coverage criterion. To compare with related work [1, 2, 3, 7, 8, 9], using dynamic repetition frequency concept reduces time-order and number of iteration of tester to achieve full coverage. While this improvement helps programmer to precisely monitor execution trace and high frequented parts of the program. These facts were experimented with three complicated program. Applying different fitness function, this structural-oriented tester is able to identify path which are not executable. The next step is to experiment the proposed tester with more complex program especially instrumentation helps other data types to inspect this algorithm for more real world.

REFERENCES

[1] P. R. Srivastava, P. Gupta, Y. Arrawatia, and S. Yada. "Use of Genetic Algorithm in Generation of Feasible Test Data"; ACM SIGSOFT Software Engineering Notes, 34(2), pp. 1-4, 2009.

[2] R. P. Pargas, M. J. Harrold, R. Pech, "Test Data Generation Using Genetic Algorithm", Journal of Software Testing, Verification and Reliability, John Wiley, 1999.

[3] A. A. Sofokleous, A. S. Andreou. "Automatic, Evolutionary Test Data Generation for Dynamic Software Testing"; the journal of System and Software, 81(11), pp. 1883-1898, 2008.

[4] D. E. Goldberg. "Genetic Algorithm in a Search Optimization and Machine Learning"; Addison Wesley, 1989.

[5] N. Mansour, M. Salame, "Data Generation for Path Testing"; Software Quality Journal, 12, 121-136, 2004.

[6] P. R. Srivastava, and T. Kim. "Applied to Genetic Algorithm in Software Engineering**";** International Journal of Software Engineering and its Applications, 3(4), pp. 87-96, October 2009.

[7] J. Yan, J. Zhang. "An Efficient Method to Generate Feasible Paths for Basis Path Testing"; Information Process Letters, vol.107, pp.87-92, 2008.

[8] T. Manterea, J. T. Alander. "Evolutionary Software Engineering, a review"; Applied Soft Computing, vol.5, pp.315-331, 2005.

[9] J. Miller, M. Reformat, H. Zhang. "Automatic Test Data Generation using Genetic Algorithm and Program Dependence Graphs"; Information and Software Technology, vol.48, pp.586-605, 2006.

[10] G. Myer, "the Art of Software Testing", John Wiley, 2004.