STUDY OF GENETIC ALGORITHM FOR AUTOMATIC SOFTWARE TEST DATA GENERATION

MAHA SALEH AL-ZABIDI*; DR. AJAY KUMAR;
DR. A. D. SHALIGRAM*****

*DEPT. OF COMPUTER SCIENCE
UNIVERSITY OF PUNE

** DIRECTOR, JSPM'S JAYWANT
INSTITUTE OF COMPUTER
APPLICATIONS, PUNE-33

*** DEPT. OF ELECTRONIC SCIENCES
UNIVERSITY OF PUNE

ABSTRACT
Testing is both technically and economically an essential part in high quality software production. General estimations is that testing causes about half of the expenses related to software testing production; much of the testing is done manually or using other labor-intensive methods
Many automatic test case generators have been developed and many classical searching methods have been used to derive test case. However these techniques work only for continuous function (while most program domains are of discontinuous spaces). This is where Evolutionary Algorithms steps in, such as those using Genetic Algorithms GA. GAs are heuristic search techniques which are able to search a discontinuous space. Evolutionary Testing is a promising approach to entirely automate test case design. It can be used to generate test cases for structural testing [5, 21, and 23]
In this study, a GA based test data generation is presented. The proposed GA is applied to different types of software systems small programs with different complexity. Results compared to random testing showed that genetic algorithms can be used effectively in automatic software testing to generate test data for unit testing.

KEY WORDS : software testing - unit testing – evolutionary testing - genetic algorithms - test data generation,

1. Introduction:

Software needs to be tested properly and thoroughly, such that any misbehavior during the run time can be detected and fixed in advance, before its delivery. Generally the goal of software testing is to design a set of minimal number of test cases such that it reveals as many faults as possible. A good test case is one that has a high probability of detecting an as-yet undiscovered error [24].
Software testing is a laborious and time-consuming work; it spends almost 50% of software system development resources [1], [24]. It is vital for the software industry to develop efficient cost effective and automatic means and tools for this task. Even partial automation

of the testing with an effective tool can bring considerable saving. Absolutely an automated software testing can significantly reduce the cost of developing software. However, software testing is not a straightforward process. For years, many researchers have proposed different methods to generate test data automatically [5], [14], [21], [23].

2. Problem definition

This research was motivated by the raped growth of software industry, where new measurement techniques and metrics are needed, for assessing the quality and reliability of software. In the early age of automation of software testing, most data generators were using gradient descent algorithms [10], however these algorithms were inefficient and time consuming and it could not escape from local optima in the space of the domain of possible input data [8]. GA where chosen as testing method because in recent decades they have grown in popularity in optimization of engineering problems [23], [24]. The aim of the work is to investigate the effectiveness of genetic algorithms over random testing and to automatically generate test data to traverse all paths of software. In our study we are presenting an enhancement to the SIMILARITY fitness function introduced by Lin and Yeh [11]. Results obtained are compared with Random testing to assess the effectiveness and efficiency of the proposed algorithm.

3. Testing methods:

3.1. Random Testing:

Random testing selects arbitrary test data from the input domain and then theses data are applied to the program under test. The automatic production of random test data drawn from a uniform distribution should be the default method by which other systems should be judged [9]. Random testing is useful since it can be performed without manual intervention and computer time is much less expensive than human time.

3.2 Evolutionary testing:

Evolutionary testing is characterized by the use of meta-heuristic search for test case generation. To achieve this; the considered test aim is transformed into an optimization problem [20,21]. The input domain of the test object forms the search space for test data that fulfills the respected search aim. Due to the non-linearity of a software (if statements, loops, ..etc) neighborhood search methods are not suitable in such case.
Therefore meta-heuristic search methods are employed e.g. Genetic Algorithms (GA), Simulated Annealing (SA) [22] or Taboo search. Evolutionary algorithm is universally applicable because it adopts itself to the system under test. In this work GA is used to generate test data because their robustness and suitability for solutions of different test tasks has already been proven in previous work, e.g. [5], [7], [19], [21] and [23].
The main idea behind GA is to evolve a population of individuals (candidate solutions for the problem) through competition, mating and mutation, so that the average quality of the population is systematically increased in the direction of the solution of the problem at hand.
The most common operations of a Genetic Algorithm include:

(a) *Reproduction:* this operation assigns the reproduction probability to each individual based on the output of the fitness function. The individual with a higher ranking is given a greater probability for reproduction. As a result, the fitter individuals are allowed a better survival chance from generation to the next. The selection requires that the solution be evaluated for their fitness as parents: solution that is closer to an optimal solution is judged higher, or more *fit*, than others. After solutions have been evaluated, several are selected in a manner that is biased towards the solutions with higher fitness values. The reason for the bias is that a good solution is assumed to be composed of good component (*genes*). Selecting such solutions as parents increases the chance that their offspring will inherit theses genes and will be at least as fit. Although the selection is biased towards the better solutions, the worst members of the population still have chance of being selected as parents- even a poor solution may have a few good genes that may benefit the population.

(b) *Crossover*: this operation is used to produce the descendants that make up the next generation. This operation involves the following procedures:

(i) select two individuals as a couple from the parent generation.
(ii) randomly select a position of the genes, corresponding to his couple, as a crossover point, thus each individual is divided into two parts.
(iii) exchange the first part of both genes corresponding to the couple.
(iv) add the two resulted individuals to the next generation.

(c) *Mutation*: this operation picks a gene at random and changing its state according to the mutation probability. The purpose of the mutation operation is to maintain the diversity in a generation to prevent premature convergence to local optimal solution. The mutation probability is given intuitively since there is no definite way to determine the mutation probability.

Upon completion of the above procedures, a fitness function should be devised to determine which of these parents and offsprings can survive into the next generation. These operations are iterated until the goal is achieved. Genetic algorithms guarantee high probability of improving the quality of individuals over several generations according to the Schema Theorem [8].

4. Structural testing:

Structural testing/Unit testing is a white-box test that considers the internal structure of software when generating test data. It is the verification of a single module, usually in an isolated environment (i.e. isolated from all other modules). White-box testing is also called the structural testing because use requires thorough knowledge of the internal structure of software [23] [16], where test cases are selected so that the structural components are covered. Example of such techniques are statement testing, branch testing, path testing, data flow testing and domain testing [6].

4.1. Path testing:

Many GA based test data generators adopted statement or branch coverage as their objectives, however, by nature, path coverage criterion covers statement and branch coverage. In other words, if testing can be designed to force execution of all paths, every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed in its true and false sides [14], which make it the utmost coverage. Thus an effective software structural testing should have path coverage as its objective.

Path testing searches the program domain for suitable test cases that covers every possible path in the software under test (SUT). However it is impossible to achieve this goal, for several reasons: (1) a program may contain an infinite number of paths when the program has loops [19]. (2) The number of paths in a program is exponential to the number of branches in it. (3) the number of test cases is too large, since each path can be covered by several test cases, making the coverage, of all possible paths computationally impractical [5][13]. It is only practical to select a specific subset path to perform path testing. Since it is impossible to cover all paths in software, the problem of path testing selects a subset of paths to execute and find test data to cover it.
Genetic algorithms could be applied to path testing if the target paths are clearly defined and an appropriate fitness function related to this goal is built.
Test case generation for path testing consists of four basic steps:

1) *Control Flow Graph construction*: A CFG is a representation of a program where contiguous regions of code without branches, known as basic blocks, are represented as nodes in a graph and edges between nodes indicate the possible flow of the program. A cycle in CFG may imply that there is a loop in the code. In this step, the source program is transferred to a graph that represents the control flow of the program. Each branch of the graph is denoted by a label and different branches correspond with different labels.
2) *Target path selection*: In path testing, paths are extracted from the CFG, and some paths might be very meaningful and need to be selected as target path for testing (e.g. the path for Equilateral triangle).
3) Test case generation and execution: in this step the algorithm automatically creates new test cases to execute new path and leads the control flow to the target path. Finally, a suitable test case that executes the target paths could be generated.
4) Test result evaluation: this step is to execute the selected path and to determine the test criteria is satisfied.

5. The proposed algorithm

Our approach is to develop GA algorithm to generate test data for path coverage. The fitness function proposed to evaluate each test data is a modification of *SIMILARITY*, presented by Lin-Yeh [11] which had extended the Hamming Distance to quantify the distance between two given paths. The fitness function in this work will be named Shifted-Modified-SIMILARITY (SMS)
According to Lin-Yeh, given a target path and a current one, the similarity is calculated from n-order sets of ordered and cascaded branches for each of the paths being compared using symmetric differences (the symmetric difference between set A and set B is ($A \oplus B$)). Similarities are then normalized and summed associated with a weighting factor which is usually found by experience.
For path testing, two different paths may contain the same branches but in different sequences. The simple Hamming distance is no longer suitable [11]. To increase path coverage within limited time and computational efforts, we proposed the (SMS). Where we

replaced the comparison of M^{th} ordered pairs of cascaded branches $(1 \leq M \leq n)$ for calculating the distance by *shift* step for the paths being compared to right and left for calculating that distance.
Given a target path A and a current path B the similarity between path A and B (i.e. SMS) is the sum of the distances with shifting to right and left *RS* and *LS* respectively.
RS is expressed as:

$$RS = n_i \times (A \oplus B)$$

Where n_i is number of nodes compared at step i of shifting to right. Illustration of *RS* calculation between path A and path B is shown in Fig (1)..

```
A : Target path.

B: Current path.

l: length of shorter path of paths A, B.

for i=1 to l do

{          n_i=no of nodes to be compared at step i;
```

Fig (1): Shift-Right similarity between path A and path B.

In the same way, *LS* is expressed as:

$$LS = n_i \times (A \oplus B)$$

Where n_i is number of nodes compared at step i of shifting to left.
The object fitness function SMS between two paths A and B is:

$$SMS = \sum_{i=l}^{1} (RS + LS)$$

The algorithm starts with creating initial random solution (i.e. list of test data). The second step is to evaluate initial solutions in order to be selected according to fitness value. The algorithm evaluates test data by executing the program with the test data as input, and recording the path that was covered in the program. The current path is compared to the target path. Test data's fitness evaluation depends on the number of common nodes current path has with target. A test case has the greatest number of common edges with the target path is given the highest fitness evaluation. The algorithm biases its selection of parents for the next generation of test cases using the fitness evaluation, and uses crossover and mutation to generate new population of test cases.

6. Discussion of Results:

To determine the potential effectiveness of the SMS over Random approach, a case study was performed on five C programs. Table (1) gives information about these programs, for each subject program the table lists the number of lines of code LOC, the cyclometric complexity, and a short description of the program.

Table (1): subject programs of the study

Programs	Lines of code	Cyclometric complexity	Description of program
Mid	34	3	Given 3 integers, determine the middle value.
Quad-Equ	40	3	Given 3 integers as the factors of quadratic equation, find the roots if any.
P-lie	35	4	Given a point(x,y) determine if it lies on X-axis, Y-axis or the Origin.
Scale	52	5	Given a numeric value as student score, determine the rank (letter grade) of the student.
Tri-class	65	7	Given 3 integers representing the side of a triangle, determine the triangle type if any.

For each subject program 10 SMS runs and 10 Random runs were performed. The results are shown in table (2) below.

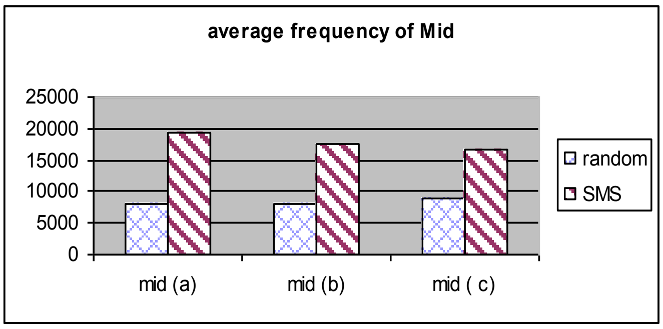
Although the proposed GA algorithm along with random testing was applied with different sizes of test suits (100 to 1000 individuals and 10 to 100 generations/iterations), the best results was with *pop size* 500, and 50 generations/iterations. The probability of crossover was set to 1.0 meaning that selected individuals always cross this helped to achieve coverage sooner.

Table (2): results of SMS and random tests of the subject programs

Program	Genetic or Random	Runs	Mean	Median	Std-Dev	Min	Max
Mid	Genetic	10	17787	17871	390	16942	18255
	Random	10	8357	8347	32	8319	8408
Quad-Equ	Genetic	10	19070	19044	252	18644	19447
	Random	10	8324	8328	16	8243	8417
P-lie	Genetic	10	7199	5842	2939	5120	13115
	Random	10	162	160	10	142	182
Scale	Genetic	10	14619	14600	186	14364	14977
	Random	10	8266	8270	47	8177	8339
Tri-class	Genetic	10	11091	11040	253	10818	11557
	random	10	2649	6258	89	6127	6413

The values in table (2) were obtained from calculation of coverage frequency of the subject programs.
The small value for standard deviation (Std-Dev) in random testing means that it took approximately similar number of generations to cover the selected goal paths. The large value of (Std-Dev) indicate that different runs of the algorithm has significantly different coverage frequency of the selected paths which means there is a significant difference in the coverage frequency of paths and this is due to the fact that some paths are straight codes and easer to reach coverage using GA. The random testing couldn't distinguish simple paths from difficult paths since there is no feed back from the program under test to the search algorithm.

We discus below the effectiveness (indicated by the average frequency of coverage of the selected goal paths) and efficiency (indicated by the number of generations took to cover selected goal path) of the proposed algorithm.
The following figures (*see* Fig. (2) to Fig (6)) summarize the effectiveness of the proposed GA algorithm over 10 runs on the average.



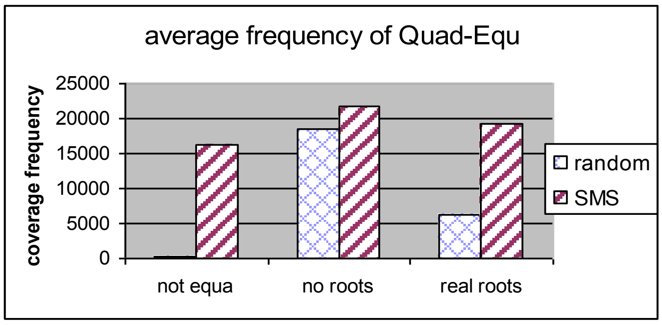
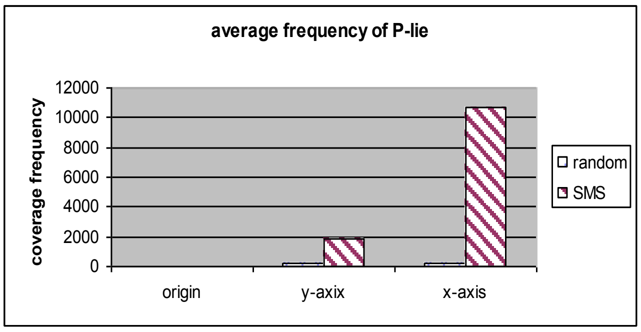


Fig (2): Effectiveness of Mid program

Fig (3): Effectiveness of Quad-Equ program



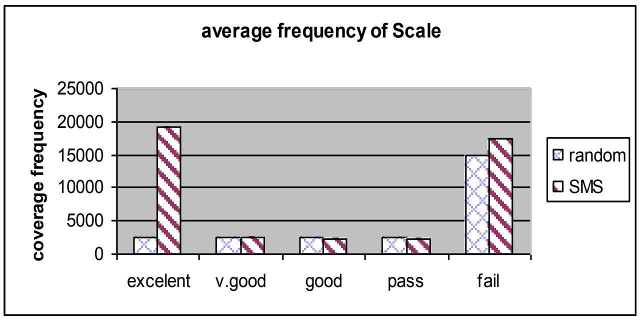


Fig (4): Effectiveness of P-lie program

Fig (5): Effectiveness of Scale program

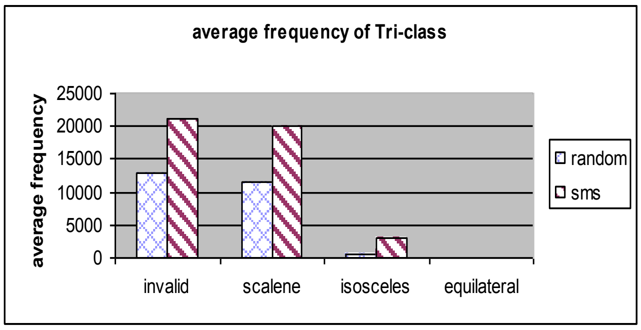


Fig (6): Effectiveness of Tri-class program

In the path *Origin* of P-lie and paths *V.good*, *Good* and *Pass* the in program Scale difference between the two algorithms performance was insignificant. No algorithm can be claimed as an absolute best or an absolute worse.
The average coverage frequency of the Equilateral triangle path in *Tri-class* program was *5.2* while Random testing achieved average of *0.2* only. *Tri-class* has three nested selection statements in which all decisions are compound predicates. GA search SMS visibly outperformed random test generations in our study by finding test cases to cover all the four triangle types.
In the remaining paths of subject programs, there was a considerable difference in the average of coverage frequency between SMS and random testing.

To study the efficiency of proposed algorithm, tables (3) to (7) list the performance of program suit. It shows the generation at which the goal path was covered over 10 runs average, with *pop-size* 500 individuals for *max-gen* 20 iterations.

Table (3) Efficiency of Mid program

Paths	Test method	
	random	SMS
mid (a)	1	1
mid (b)	1	1
mid ©	1	1

Table (4) Efficiency of Quad-Equ program

Paths	Test method	
	random	SMS
not equ	1	1
no roots	1	1
real roots	1	1

Table (5) Efficiency of P-lie program

Paths	Test method	
	random	SMS
origin	N/A	N/A
x-axis	1	1.1
y-axis	1.1	1

Table (6) Efficiency of Scale program

Paths	Test method	
	random	SMS
excellent	1	1
v.good	1	1
good	1	1
pass	1	1
fail	1	1

Table (7) Efficiency of Tri-class

Paths	Test method	
	random	SMS
invalid	1	1
scalene	1	1
isosceles	1	1
equilateral	N/A	10.3

As shown in tables (3) to (6), since code of these programs did not involve nested conditions, they were easily satisfied and most of the code in programs is straight line. SMS achieved 100% path coverage in the initial population. Random achieved similar results.
It is worth mention that *N/A* in program *P-lie* (see table (5)) indicate that both SMS and random testing failed to generate test data that satisfy the *Origin* path since the input (0,0) was not generated within *max-gen.*
Also, Table (7) shows that random testing failed to cover *equilateral triangle* path where input of three equal integers is required whereas SMS achieved coverage of that path in average of approximately 10 generations only. As mentioned earlier, *Tri-class* program contains nested compound condition statements that increased the logical complexity of the program which increases difficulty to achieve path coverage.
A random test data may create many test data; however, because information about the test requirement is not incorporated into the generation process, the test data generator may fail to find test data to satisfy the requirement.
As a summary, GA search outperformed random test data generation by a considerable margin in most of our experiments and always performed at least as well.

7. Conclusion:

We have presented in this article an algorithm for test data generation using GA and a tool implementing the approach for unit-test-level. The algorithm, based on SMS fitness function, is an enhancement of the SIMILARITY fitness function introduced by Lin-Yeh [11].
We used our test data generation algorithm on a suit of small C programs that have frequently been used as benchmark for test data generation techniques.
We applied random test data generation to the same programs. We permitted the same number of runs as we used by genetic search. In this way we allowed random test generation the same computational resources that genetic search used.
The results have shown that the proposed GA algorithm requires les iterations and works more effectively than random testing. We attribute the poor performance of random generation to the complexity of the program code and the increased difficulty of path coverage. Although some results suggests that random test generation might be valuable, our results indicate that this value may decrease considerably for complex programs.
The proposed GA accepts as input an instrumented version of the program to be tested. The list of paths to be covered, and GA parameters: population size, maximum number of generations, probabilities of crossover and mutation.
In the next step we are going to experiment with more complex programs code and more complicated structures (loops, arrays, … etc).
Finally, another possibility is to compare the algorithm with other exhaustive, search techniques to see if co-operation among different search algorithms is possible.

References:

1. Beizer, B., “Software Testing Techniques”, Van Nosterland Reinhold, 1990.
2. Berndt, D. et al, “Breeding Software Test Cases with Genetic Algorithms”, IEEE Proceedings of the Hawaii International Conference on System Science, Hawai’I,2003
3. Borgelt, K., “Software Test Data Generation from a Genetic Algorithm”, Industrial Applications of Genetic Algorithms, CRC Press, 1999.
4. “*Code Coverage*” , www.en.wikipedia.org/wiki/code-coverage

5. B. T. de Abreu, E. Martins, F, de Sousa, "*Automatic Test Data Generation for Path Testing Using A New Stochastic Algorithm"*, http://www.sbbd-sbes2005.ufu.br/arquivos/16-%209523.pdf
6. J. Edverdson, "*A Survey on Automatic Test Data Generation*", proceedings of the 2nd Conference on Computer Science and Engineering, ECSEL, October 1999.
7. M.R .Girgis, "Automatic Test Data Generation for Data Flow Testing Using Genetic Algorithm", Journal of Universal Computer Science, vol. 11, no.6, 2005.
8. D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989.
9. Ince. D.C. and Hekmatpour, S.."An Empirical Evaluation of Random Testing"' the Computer Journal, vol.29, no 4, pp 380, 1986.
10. Korel, B. "Automatic Software Test Data Generation" IEEE Trans of Software Engineering, 16(8):870-879.
11. J. Lin, P.L.Yeh, "*Automatic Test Data Generation of Path Testing Using GAs*", Information Sciences, 131(1-4):47-64.
12. Mall,R. "Fundamentals of Software Engineering" ,Prentice-Hall of India, 6th printing, 1999.
13. Mansour, N., Salame, M.; "Data Generation for Path Testing". Software Quality Journal, 12,121-136-2004.
14. Michael, et. Al. "genetic algorithm for dynamic Test Data Generation" Proceedings of the 12th International Conference of Automated Software Engineering, vol.1, issue 5, Nov.1997, pp: 307-308.
15. Myer. G. "The Art of Software Testing", John Wiley, 1979.
16. Myers, G. "Software Reliability, Principle and Practices", John Wiley, 1981.
17. Pargas, et al, "Test Data Generation Using Genetic Algorithms", Journal of Software Testing, Verification and Reliability, John Wiley, 1999.
18. Pressman, R, S. "Software Engineering, A Practitioner's Approach", McGraw-hill, 5th Ed.2002.
19. D. Sandler, W. Tisthammer, "*A Survey of Testing Tools*", www.users.umn.edu/~dliang/5802reports/06/sandler/surveytesting.pdf
20. Sthamer, H. "The Automatic Generation of Software Test Data Using Genetic Algorithms", PhD. Thesis, University of Glamorgan, Wales, Great Britain, 1996.
21. Sthamer H.,A. Baresel, J. Wegener, "*Evolutionary Testing of Embedded Systems*" 14th International Internet Quality week 2001.
22. N. Ttracy,J. Clark, K. Mander, "*Automated Program Flow Finding With Simulation Annealing*", Proceedings of ASSTA , Florida ,USA,1998.
23. Wegener J., K. Buhr, H. Pohlheim. *"Automatic Test Data Generation for Structural Testing of Embedded Software System by Evolutionary Testing", http://www.lri.fr/~marc/articls/testwegener2002.pdf*
24. Whittaker J.A., "What is software testing? And Why is it so hard?" IEEE Software, 2000.