International Journal of Software Engineering and Its Applications
Vol. 3, No.4, October 2009

Application of Genetic Algorithm in Software Testing

Praveen Ranjan Srivastava[1] and Tai-hoon Kim[2]

[1,] *Computer Science & Information System Group, BITS PILANI – 333031 (INDIA)*
praveenrsrivastava@gmail.com
[2]*Dept. of Multimedia Engineering, Hannam University, Korea*
taihoonn@hnu.kr

Abstract

This paper presents a method for optimizing software testing efficiency by identifying the most critical path clusters in a program. We do this by developing variable length Genetic Algorithms that optimize and select the software path clusters which are weighted in accordance with the criticality of the path. Exhaustive software testing is rarely possible because it becomes intractable for even medium sized software. Typically only parts of a program can be tested, but these parts are not necessarily the most error prone. Therefore, we are developing a more selective approach to testing by focusing on those parts that are most critical so that these paths can be tested first. By identifying the most critical paths, the testing efficiency can be increased.

Keywords: *Software Testing, Genetic Algorithm, Test Data*

1. Introduction

The verification and validation of software through dynamic testing is an area of software engineering where progress towards automation has been slow. In particular the automatic design and generation of test data remains, by and large, a manual activity. Software testing remains the primary technique used to gain consumers' confidence in the software. The process of testing any software system is an enormous task which is time consuming and costly [1] [2]. Software testing is laborious and time-consuming work; it spends almost 50% of software system development resources [1] [2]. 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. As mentioned earlier, software testing is a lengthy and time-consuming work [3]. Absolutely, an automated software testing can significantly reduce the cost of developing software. Other benefits include: the test preparation can be done in advance, the test runs would be considerably fast, and the confidence of the testing result can be increased. However, software testing automation is not a straight forward process. For years, many researchers have proposed different methods to generate test data automatically, i.e. different methods for developing test data/case generators [4, 5, 6, 7, 8, 9]. The development of techniques that will also support the automation of software testing will result in significant cost savings. The application of artificial intelligence (AI) techniques in Software Engineering (SE) is an emerging area of research that brings about the cross fertilization of ideas across two domains. A number of researchers did the work on software testing using artificial inelegance; they examine the effective use of AI for SE related activities which are inherently

knowledge intensive and human-centered. These issues necessitate the need to investigate the suitability of search algorithms, e.g. simulated annealing, genetic algorithms, and ant colony optimization as a better alternative for developing test data generators [4, 5].Using evolutionary computations, researchers have done some work in developing genetic algorithms (GA)-based test data generators [6, 7, 8, 9, 10]. A variety of techniques for test data generation have been developed previously [12, 13, 14, 15, 16] and these can be categorized as structural and functional testing.

In this paper, we present the results of our research into the application of GA search approach, to identify the most error prone paths in a software construct. The paper is structured in the following way: section 2 describe basic structure of genetic algorithm, in section 3 we discussed our proposed algorithm for test data generator, while section 4 represents the case study of proposed approach using an example and finally in section 5.describe the conclusions part.

2. Genetic algorithm

A GA [10] starts with guesses and attempts to improve the guesses by evolution. A GA will typically have five parts: (1) a representation of a guess called a chromosome, (2) an initial pool of chromosomes, (3) a fitness function, (4) a selection function and (5) a crossover operator and a mutation operator. A chromosome can be a binary string or a more elaborate data structure. The initial pool of chromosomes can be randomly produced or manually created. The fitness function measures the suitability of a chromosome to meet a specified objective: for coverage based ATG, a chromosome is fitter if it corresponds to greater coverage. The selection function decides which chromosomes will participate in the evolution stage of the genetic algorithm made up by the crossover and mutation operators. The crossover operator exchanges genes from two chromosomes and creates two new chromosomes. The mutation operator changes a gene in a chromosome and creates one new chromosome. GA has well-defined steps:

A basic algorithm for a GA is as follows [1]
The pseudo code for GA is:

```
Initialize (population)
Evaluate (population)
While (stopping condition not satisfied) do
{
    Selection (population)
    Crossover (population)
    Mutate (population)
    Evaluate (population)
}
```

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

3. Proposed Approach

This section describes details of our proposed approach, to test data generation using GA; more precisely, it describes our fitness function. Our approach uses a weighted CFG. Path testing searches the program domain for suitable test cases that covers every possible path in the software under test (SUT). However, it is generally impossible to achieve this goal, for several reasons. First, a program may contain an infinite number of paths when the program has loops. Second, the number of paths in a program is exponential to the number of branches in it and many of them may be unfeasible. Third, the number of test cases is too large, since each path can be covered by several test cases. For these reasons, the problem of path testing can become a NP complete problem making the covering of all possible paths computationally impractical. 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.

Our algorithm works on control flow graph (CFG). CFG is a simple notation for the representation of control flow. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph an independent path must move along at least edge that has not been traversed before the path is defined.

3.1 Procedure

Input: CFG of the code

Assigning weights to edges of CFG – The first step of algorithm is assigning weights to CFG. More weights are assigned to edges which are critical so to say, that are part of paths which are more error prone. An initial credit is taken (100 or 10), if CFG is dense i.e. large numbers of edges are there than initial credit should be taken as 100 and if CFG is sparse (small codes) then it can be taken as 10.

At each node of CFG the incoming credit (sum of the weights of all the incoming edges) is divided and distributed to all the outgoing edges of the node.

Distribution of weights is done as follows:

Take ‘n’ to be the number of outgoing edges.

We have considered an 80-20 rule. 80 percentage of weight of the incoming credit is given to loops and branches and the remaining 20 percentage of the incoming credit is given to the edges in sequential path. From each node if n1 is the number of edges in sequential path and n2 is the number of edges in looping and branching paths, then n1 edges are given 20 percentage of incoming weight and then divided equally amongst them and the remaining 80 percentage is given to n2 edges.

If there is only one outgoing edge from a particular node than the incoming weight is assigned to the outgoing edge.

In figure 1 of illustration it can be seen that a similar procedure has been followed to assign weights to the edges.

3.2 Selection

The selection of parents for reproduction is done according to a probability distribution based on the individual’s fitness values. First the fitness value is calculated

using the Fitness function proposed in the algorithm. Weights are used to determine the relative contribution of a path to the fitness calculation. Thus, more weight is assigned to a path which is more "critical". Criticality of the path to test data generation is based on the fact that predicate, loop and branch nodes are given preference over sequential nodes during software testing. The fitness function we are using here is

$$F = \sum_{i=1}^{n} wi$$

Where, wi = weight assigned to i-th edge on the path under consideration

The algorithm works by assigning weights to the edges (depicting flow) of CFG on the basis of the importance of path in which the edge lies. Higher weights are assigned to the edges of path corresponding to the critical section of the code for example loops, branch statements, control statements etc. for which testing is essential. After all the fitness function values are calculated, the probability of selection pj for each path j, so that

$pj = Fj / \sum . Fj$
Where, j=1 to n
n= initial population size

Then cumulative probability ck is calculated for each path k with equation:

$$ck = \sum_{j=1}^{k} pj$$

3.3 Reproduction (crossover)

In one-point (or single) crossover, two input data selected as potential parents by selection process exchange substring information at a random position in the data to produce two new data. Crossover happens according to a crossover probability pc, which is an adjustable parameter. For each parent selected, generate a random real number r in the range [0, 1]; if r < pc then select the parent for crossover. After that, the selected data are formatted randomly. Each pair of parents generates two new paths, called offspring.

The crossover technique used is one point crossover done at the midpoint of the input bit string. In this technique, right half of the bits of one parent are swapped with the corresponding right half of the other parent.

3.4 Mutation

Mutation is performed on a bit-by-bit basis. Every bit of every chromosome in the offspring has an equal chance to mutate (change from '0' to '1' or from '1' to '0'), and the mutation occurs according to a mutation probability pm, which is also an adjustable parameter. To perform mutation, for each chromosome in the offspring and for each bit within the chromosome, generate a random real number r in the range [0, 1]; if r < pm then mutate the bit.

These major components including the fitness function will evolve test data to better ones, trying to find a candidate that covers the target path. The crossover process tries to create better test data from fitter ones, while mutation introduces diversity into population, avoiding getting stuck at local optima solutions.

4. Case Study

I have to check my procedure under case study.

```
0. gcd (int m, int n)
   {
    int r;
1. if (n>m) {
2. r=m;
3. m=n;
4. n=r;
    }
5. r=m%n;
6. While(r! =0)
    {
7. m=n;
8. n=r;
9. r=m%n;
10. }
11. return n;
12.}
```

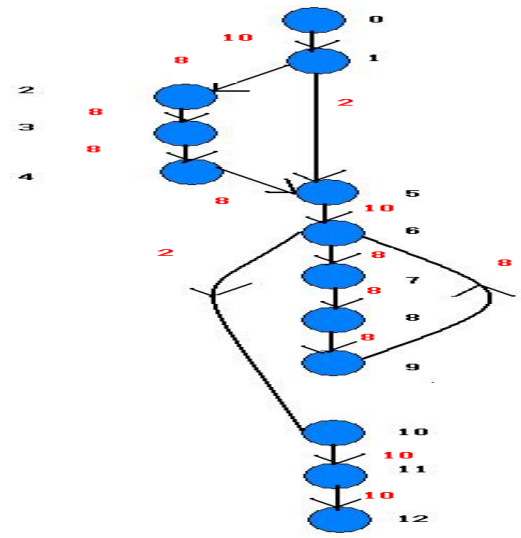

Figure1: Code with weight CFG

This is the CFG for the above code. The numbers of the nodes correspond to line of the code. Since line 1 of the code is an IF statement, node 1 becomes a predicate node and two outgoing edges are made to account for the two possible outcomes of the IF statement. Line 6 is a while statement and correspondingly node 6 has two outgoing edges in the CFG. Also can be seen is an edge from node 9 to node 6 depicting the loop. This figure shows the assignments of weights to the CFG following the procedure described in section 3.1. Here an initial credit of 10 is taken since it is a small piece of code. Then as we traverse CFG from top to bottom, we keep dividing the incoming credit at each node following the 80 – 20 rule.

4.1 Assigning Weights

In the given example we started with initial credit of 10 and then distributed it to the various edges on the basis of their importance. Values of weights are shown in figure 1 and are those which are closer to the edges.

4.2 Solving Case study Using Genetic Algorithms

The table depicts the procedure and key to the table is:

X: denotes our test data set

F(x): corresponding fitness value calculated for each test data, by adding the weights of the path followed by it in the CFG.

Pi: probability for the corresponding data

$$Pi = F(Xi) / (\sum (F(Xi))$$

Ci: cumulative probability

Ran: Random number generated for the test data

Ns: Test data number that has cumulative probability just greater than the corresponding random number.

Mating pool: This column contains the number of times a test data appears in the Ns column.

Steps for carrying out cross over and mutation:

The data obtained from the Ns values is written in binary representation

Pair-wise crossover is done by interchanging the second half off the binary representation of the data, for data satisfying the condition that Ran<0.8. This is because crossover probability is 80% and crossover is carried out only if its corresponding random number is less than this probability.

Mutation: For each entry in the new data set, bit-wise random number are generated. And for random number values less than 0.3, that corresponding bit is flipped to obtain a new data entry.

Same procedure is carried out for the new data set obtained for further crossover and mutation until we start getting better values of the fitness function F(x).

Example 1:

Initial population: (n, m)

(15, 4), (5, 6), (6, 2), (4, 12)

Fitness function used:

Summation of weights of path traversed by a given input data in CFG

For example (15, 4) will travel the path 0-1-2-3-4-5-6-7-8-9-6-7-8-9-6-10-11-12 and therefore its fitness value is 108

Since the mating pool consists of only (15, 4) therefore this is the test data that should be used for the testing of the code during execution. This is because the mating pool depicts the population that will mate in the next iteration. Here since the only value in mating pool is (15, 4), this shows that no further improvement in the fitness function value can be achieved with further reproduction and mutation. Thus (15, 4) is the test data that should be used as input for software testing.

S.N	X	F(x)	Pi	Ci	Ran	Ns	*Mating Pool*
1.	(15,4)	108	0.325	0.325	0.312	1	4
2.	(5,6)	106	0.319	0.644	0.098	1	0
3.	(6,2)	44	0.133	0.777	0.267	1	0
4.	(4,12)	74	0.223	1.000	0.141	1	0

Example 2

Initial population: (n, m)

(12, 8), (2, 3), (6, 2), (15, 4)

Iteration: Table 1, 2, 3, 4, 5, 6and Table 7 on the next page show iteration of the procedure followed. Fitness function values of input population is calculated in coloumn3,then probability is calculated using the formula in section 3.Coloumn5 shown the cumulative probability .Random number are generated to simulate the GA process.

Iteration 1: Table 1 and Table 2

Iteration 2: Table 3 and Table 4

Iteration 3: Table 5 and Table6

Final Result : Table7

Table 1

S. No.	X	F(x)	Pi	Ci	Ran	Ns	*Mating Pool*
1.	(12,8)	76	0.228	0.228	0.934	4	0
2.	(2,3)	106	0.317	0.545	0.474	2	2
3.	(6,2)	44	0.132	0.677	0.374	2	1
4.	(15,4)	108	0.323	1.000	0.618	3	1

Table 2

S. No.	Ns	Mating Pool	Crossover	Mutation
1	4	(15,4) 11110100	(15,4) 11110100	(15,5) 11110101
2	2	(2,3) 00100011	(2,3) 00100011	(3,3) 00110011
3	2	(2,3) 00100011	(2,2) 00100010	(2,2) 00100010
4	3	(6,2) 01100010	(6,3) 01100011	(10,3) 10100011

Table 3

S. No.	X	F(x)	Pi	Ci	Ran	Ns	*Mating Pool*
1.	(15,5)	44	0.212	0.212	0.217	2	0
2.	(3,3)	44	0.212	0.424	0.999	4	1
3.	(2,2)	44	0.212	0.636	0.979	4	1
4.	(10,3)	76	0.364	1.000	0.533	3	2

Table 4

S. No.	Ns	Mating Pool	Crossover	Mutation
1	2	(3,3) 00110011	(3,2) 00110010	(5,10) 01011010
2	4	(10,3) 10100011	(10,3) 10100011	(9,10) 10001010
3	4	(10,3) 10100011	(10,3) 10100011	(11,6) 10110110
4	3	(2,2) 00100010	(2,3) 00100011	(2,2) 00100010

Table 5

S. No.	X	F(x)	Pi	Ci	Ran	Ns	*Mating Pool*
1.	(5,10)	74	0.223	0.223	0.934	4	0
2.	(9,10)	106	0.319	0.542	0.474	2	2
3.	(11,6)	108	0.325	0.867	0.374	2	1
4.	(2,2)	44	0.133	1.000	0.618	3	1

Table 6

S. No.	Ns	Mating Pool	Crossover	Mutation
1	4	(2,2) 00100010	(2,2) 00100010	(4,3) 01000011
2	2	(9,10) 10011010	(9,10) 10011010	(7,12) 01111100
3	2	(9,10) 10010010	(9,10) 10011010	(13,10) 11011010
4	3	(11,6) 10110110	(11,6) 10110110	(2,6) 00100110

Table 7

S. No.	X	F(x)	Pi	Ci	Ran	Ns	*Mating Pool*
1.	(4,3)	76	0.178	0.178	0.098	1	1
2.	(7,12)	170	0.399	0.577	0.275	2	3
3.	(13,10)	106	0.249	0.826	0.325	2	0
4.	(2,6)	74	0.174	1.000	0.487	2	0

** Description to these tables is on the previous page.*

5. Conclusion

Genetic algorithms are often used for optimization problems in which the evolution of a population is a search for a satisfactory solution given a set of constraints. We have reported preliminary results from an experiment comparing random test data generation with a new approach using genetic search. In this paper we have demonstrated that it is possible to apply Genetic Algorithm techniques for finding the most critical paths for improving software testing efficiency. The Genetic Algorithms also outperforms the exhaustive search and local search techniques. In conclusion, by examining the most critical paths first, we obtain a more effective way to approach testing which in turn helps to refine effort and cost estimation in the testing phase. Our experiments conducted so far are based on relatively small examples and more research needs to be conducted with larger commercial examples. Future research will involve comparing GA selected paths in larger test data and further refining the method presented. This research would help in generating various software test cases. Also, since GA can be used independently for any problem and it is an emerging field so it has tremendous importance for users.

References

[1] Somerville, I., "Soft ware engineering," 7th Ed. Addison-Wesley,

[2] Aditya P mathur,"Foundation of Software Testing", 1st edition Pearson Education 2008.

[3] Alander, J.T., Mantere, T., and Turunen, P, "Genetic Algorithm Based Software Testing," http://citeseer.ist.psu.edu/40769.html, 1997.

[4] Nashat Mansour, Miran Salame," Data Generation for Path Testing", Software Quality Journal, 12, 121–136, 2004,Kluwer Academic Publishers.

[5] Praveen Ranjan Srivastava et al, "Generation of test data using Meta heuristic approach" IEEE TENCON (19-21 NOV 2008), India available in IEEEXPLORE.

[6] Wegener, J., Baresel, A., and Sthamer, H, "Suitability of Evolutionary Algorithms for Evolutionary Testing," In Proceedings of the 26th Annual International Computer Software and Applications Conference, Oxford, England, August 26-29, 2002.

[7] Berndt, D.J. and Watkins A, "Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation," In Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04), pp. 261-262, University of South Florida, March 25-26, 2004.

[8] B. Korel. Automated software test data generation. IEEE Transactions on Software Engineering, 16(8), August 1990.

[9] B.F. Jones, H.-H. Sthamer and D.E. Eyres. Automatic structural testing using genetic algorithms. Software Engineering Journal, pages 299-306, September, 1996.

[10] Goldberg, D.E, "Genetic Algorithms: in Search, Optimization & Machine Learning," Addison Wesley, MA. 1989.

[11] Horgan, J., London, S., and Lyu, M., "Achieving Software Quality with Testing Coverage Measures", IEEE Computer, Vol. 27 No.9 pp. 60-69, 1994.

[12] Berndt, D.J., Fisher, J., Johnson, L., Pinglikar, J., and Watkins, A., "Breeding Software Test Cases with Genetic Algorithms," In Proceedings of the Thirty-Sixth Hawaii International Conference on System Sciences (HICSS-36), Hawaii, January 2003.

[13]Mark Last, Shay Eyal1, and Abraham Kandel, "Effective Black-Box Testing with Genetic Algorithms," IBM conference.

[14] Lin, J.C. and Yeh, P.L, "Using Genetic Algorithms for Test Case Generation in Path Testing," In Proceedings of the 9th Asian Test Symposium (ATS'00). Taipei, Taiwan, December 4-6, 2000.

[15] andré baresel, harmen sthamer and michael schmidt, "fitness function design to improve evolutionary structural testing," proceedings of the genetic and evolutionary computation conference, 2002.

[16] Dr. Velur Rajappa, Arun Biradar, Satanik Panda, "Efficient Software Test Case Generation Using Genetic Algorithm Based Graph Theory," First International Conference on Emerging Trends in Engineering and Technology, ICETET '08, pp.298-303, 2008.

[17] Christoph C. Michael, Gary E. McGraw, Michael A. Schatz, and Curtis C. Walton, "Genetic Algorithms for Dynamic Test Data Generation," Proceedings of the 1997 International Conference on Automated Software Engineering (ASE'97) (formerly: KBSE) 0-8186-7961-1/97 © 1997 IEEE.

[18] C. Darwin. On the Origin of Species: A facsimile of the first edition. Harvard University Press, July 1975.

Authors

Praveen Ranjan Srivastava is working in computer science and information systems group at Birla Institute of Technology and Science (BITS) Pilani India. He is currently doing research in the area of Software Testing. His research areas are software testing, quality assurance, testing effort, software release, test data generation, agent oriented software testing, soft computing techniques. He has a number of publications in the area of software testing. He has been actively involved in reviewing various research papers submitted in his field to different leading Journals and various International and National level conferences. Contact him at praveenrsrivastava@gmail.com