Comparison of Software Test Data for Automatic Path Coverage Using Genetic Algorithm

Premal B. Nirpal and K. V. Kale
Department of CS & IT, Dr. B. A. M. University, Aurangabad, India
e-mail: premal.nirpal@gmail.com

ABSTRACT
This paper discusses genetic algorithms that can automatically generate test cases to test selected path. This algorithm takes a selected path as a target and executes sequences of operators iteratively for test cases to evolve. The evolved test case can lead the program execution to achieve the target path. An automatic path-oriented test data generation is not only a crucial problem but also a hot issue in the research area of software testing today.

KEYWORDS
Software Testing, Test case generation, Path Coverage, Genetic Algorithms.

1. Introduction

Software being utilized in various situations and software quality becomes more important than ever. Being main means of software quality assurance, software testing is very laborious and costly due to the act that it is accounts for approximately 50 percent of the elapsed time and more than 50 percent of the total cost in software development [4, 5].
Automatic test data generation is a key problem in software testing and its implementation can not only significantly improve the effectiveness and efficiency but also reduce the high cost of software testing[3, 4]. In particular, it is notable that various structural test data generation problem can be transformed into a path oriented test data generation problem. Moreover, path testing strategy can detect almost 65 ercent of errors in program under test [8].
Although path-oriented test data generation is an undesirable problem [6], researchers still attempt to develop various methods and have made some progress. These means can be classified into two types: static methods and dynamic methods. Static methods include domain reduction [10, 11] and symbolic execution [12] etc. These means suffer from a number of problems when they handle indefinite array, loops, pointer references and procedure calls [13].
Dynamic methods include random testing, local search approach [14], goal-oriented approach [15], chaining approach [16] and evolutionary approach [13, 14-16]. As values of input variables are determined when programs execute, dynamic test data generation can avoid those problems with that static methods are confronted. Being a robust search method in complex spaces, genetic algorithm was applied to test data generation in 1992 [14] and evolutionary approach has been a burgeoning interest since then. Related works [17], [16] and [18] indicate that GA-based test data generation outperforms other dynamic approaches e.g. random testing and local search.
The structure of this paper is organized as follows. Section 2 gives a brief introduction to Genetic Algorithms. Section 3 Basic process flow of path-oriented test data generation using GA. Section 4 describes experimental settings and gives experimental results based on a triangle classification program. Finally, section 5 summarizes the paper with conclusions and directions for future work.

2. Genetic Algorithms

Genetic Algorithms begins with a set of initial individuals as the first generation, which are sampled at random from the problem domain. The algorithms are developed to perform a series of operations that transform the present generation into a new, fitter generation [22].
Each individual in each generation is evaluated with a fitness function. Based on the evaluation, the evolution of the individuals may approach the optimal solution.
The most common operations of genetic algorithms are designed to produce efficient solution for the target problem [15]. These primary operations 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 one generation to the next.
b) Crossover: This operation is used to produce the descendants that make up the next generation. This operation involves the following crossbreeding procedures:

i) Randomly select two individuals as a couple from the parent generation.
ii) Randomly select a position of the genes, corresponding to this couple, as the crossover point. Thus, each gene is divided into two parts.
iii) Exchange the first parts 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 a local optimal solution. The mutation probability is given intuitively since there is no definite way to determine the mutation probability [22].
Upon completion of crossover processing and mutation operations, there will be an original parent population and a new offspring population. A fitness function should be devised to determine which of these parents and offspring's can be survived into the next generation. After performing the fitness function, these parents, and offspring's are filtered and a new generation is formed. These operations are iterated until the expected goal is achieved. Genetic algorithms guarantee high probability of improving the quality of the individuals over several generations according to the Schema Theorem [5].

3. Basic process flow of path-oriented test data generation using genetic algorithm

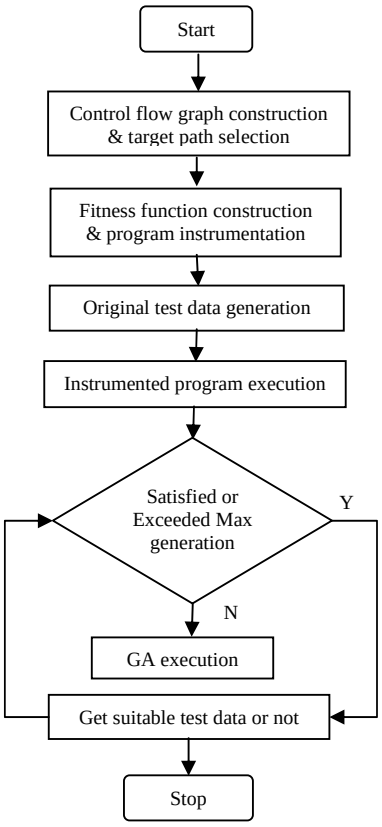

Figure 1. Basic process flow

A selected target path is the goal for GA to achieve, and an input vector X (a test data) is regarded as an individual. To generate path-oriented test data for the program under test using GA, there are five steps and Figure 1 depicts the basic process flow [6, 7].
(1) ***Control flow graph construction***. Control flow graph of the program under test may be constructed manually or automatically with related tools. It helps testers to select representative target paths.

(2) ***Target path selection***. In general, a program under test has too many paths to test completely. Thus, testers have to select meaningful paths as target paths.
(3) ***Fitness function construction***. In order to evaluate distance between the executed path and the target path, fitness function has to be constructed.
(4) ***Program instrumentation.*** This means inserting probes at the beginning of every block of source code to monitor program execution and collect related information (e.g. fitness values of individuals).
(5) ***Test data generation and the instrumented program execution.*** Original test data are chosen from their domain at random and GA generates new test data in order to achieve the target path. Finally, suitable test data that executes along the target paths may be generated or no suitable test data may be found because of exceeding max generation [22].

No	Predicate	Distance if path taken is different
1	A=B	ABS(A-B)
2	A≠B	K
3	A<B	(A-B)+K
4	A≤B	(A-B)
5	A>B	(B-A)+K
6	A≥B	(B-A)
7	X OR Y	MIN(Distance(X), Distance(Y))
8	X AND Y	(Distance(X) + Distance(Y)

Table 1: Korel's distance function

4. Experimental studies

Triangle classification program

Triangle classification program has been widely used in the research area of software testing [22, 24]. It aims to determine if three input edges can form a triangle and so what type of triangle can be formed by them. Figure 2 gives source code of the program.

```
TraversedPath= [];
 TriangleType='Not a Triangle';
 if ((SideA+SideB>SideC)&&(SideB+SideC>SideA)&&(SideC+SideA>SideB))
     TraversedPath =[TraversedPath 'a'];
     if ((SideA~=SideB)&&(SideB~=SideC)&&(SideC~=SideA))
         TraversedPath=[TraversedPath 'e'];
         TriangleType='Scalene';
     else
         TraversedPath =[TraversedPath 'b'];
         if (((SideA==SideB)&&(SideB~=SideC))||((SideB==SideC)&& ...
            (SideC~=SideA))||((SideC==SideA)&&(SideA~=SideB)))
             TraversedPath =[TraversedPath 'f'];
             TriangleType='Isosceles';
         else
             TraversedPath=[TraversedPath 'c'];
             TriangleType='Equilateral';
         end
     end
 else
     TraversedPath =[TraversedPath 'd'];
 end
```

Figure 2. An example program

1. Control flow graph construction: The tested program (Fig. 2 Triangle classification program) determines what kind of triangle can be formed by any three input lengths. The programs control flow diagram, which contains four paths, is shown in fig. 3.

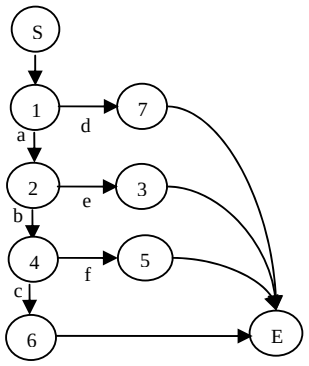

Figure 3. Control flow graph of the example Program

2. Target path selection:

Figure 3 is control flow graph of the triangle classification program, which consists of four paths:

- Path l: <d> //Not-a-triangle
- Path 2: <ae> //Scalene
- Path 3: <abf> //Isosceles
- Path 4: <abc> //Equilateral

According to probability theory, the path <abc> is the most difficult path to be covered in path testing. Therefore, the path <abc> is selected as the target path.

3. Test case generation and execution:

Experimental settings

Settings of standard genetic algorithm (SGA) are as following:

(1) Coding: binary string

(2) Length of chromosome: 3Nbits (N=8, 10……..,15), and each edge are range from 1 to 2^N

(3) Population size: from 1 to 1000

(4) Stochastic universal sampling

(5) Two-point crossover probability = 0.9

(6) Mutation probability = 0.01

(7) Generation gap = 0.96

(8) Max generation = 1000

Table 2 shows that the average number of test cases on the path of each generation. In this experiment we have used Genetic Algorithm for 100 generations with n=15, initial population with 34000 test cases. The size of the chromosome is 36. Mutation rate is 0.01. Selection rate 0.5. Figure 2 shows the average number of test cases on the path of each generation.

	<abc>	**<d>**	**<ae>**	**<abf>**	**time**	**total test cases**
Generation	Equilateral	Not a Triangle	Scalene	Isosceles		
1	157	16918	12100	4825	2.7275	34000
2	74	9173	5450	2303	1.6894	17000
3	79	9438	5253	2230	1.5997	17000
4	77	9831	4960	2132	1.8760	17000
5	86	9960	4879	2075	1.9791	17000
6	95	9988	4821	2096	2.3626	17000
7	87	10240	4626	2047	1.4046	17000
8	76	10527	4416	1981	1.4381	17000
9	81	10607	4355	1957	1.6466	17000
10	67	10636	4395	1902	1.8333	17000

Table 2. Average number of test cases on the path of Fig . 3 of each generation

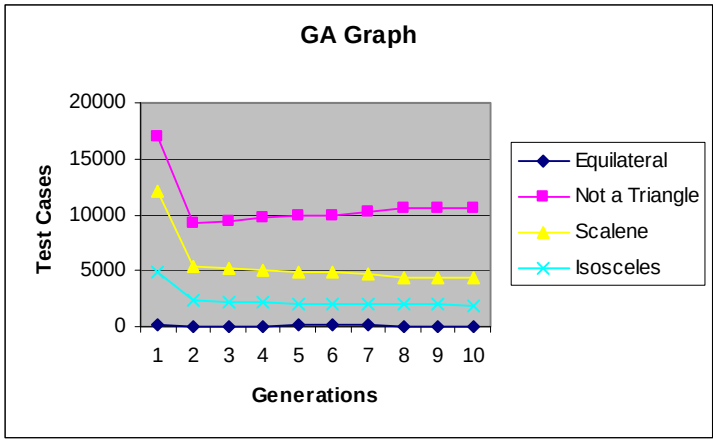

Fig. 4 Average number of test cases on the path of Fig . 3 of each generation

Using Yong Chen [28] approach, Table 3 shows that the average number of test cases on the path of each generation. In this experiment we have used GA for 100 generations with n=15, initial population with 34000 test cases. The size of the chromosome is 36. Mutation rate is 0.01. Selection rate 0.5. Figure 3 shows the average number of test cases on the path of each generation.

	<abc>	**<d>**	**<ae>**	**<abf>**	**time**	**total test cases**
Generation	Equilateral	Not a Triangle	Scalene	Isosceles		
1	0	17181	16805	14	9.2291	34000
2	0	9169	7818	13	5.1290	17000
3	0	9301	7689	10	5.2338	17000
4	0	9545	7439	16	5.6058	17000
5	0	9863	7121	16	5.6808	17000
6	0	9848	7136	16	6.2996	17000
7	0	10024	6962	14	5.9840	17000
8	0	10159	6832	9	6.0668	17000
9	0	10335	6653	12	6.2867	17000
10	0	10620	6361	19	6.1355	17000

Table 3. Average number of test cases on the path of Fig . 3 of each generation

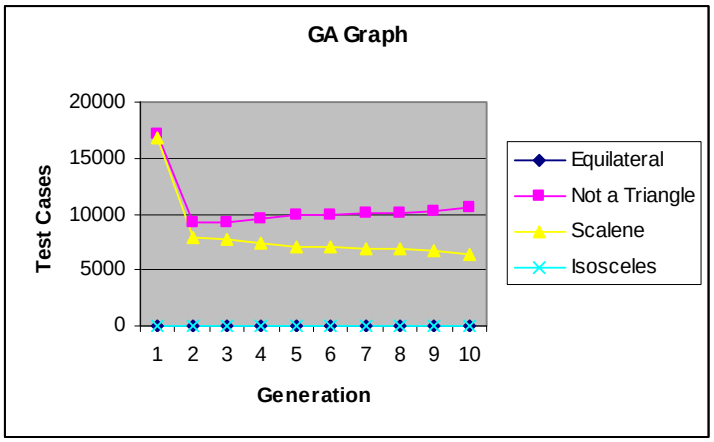

Fig. 5 Average number of test cases on the path of Fig . 3 of each generation

From the experimental results in Figure 4 and Figure 5, it is found that our genetic algorithms based test data always outperforms Young Chen based test data. With increasing length of chromosome, Young Chen based approach requires more and more number of test data and time to achieve target path. Figure 4 and Figure 5 shows average number of test data and average time.

5. Conclusion

In this paper, the genetic algorithms are used to automatically generate test cases for path testing. Using a triangle classification program as an example, experiment results show that Our Genetic Algorithm based test data can more effectively and efficiently than Young Chen method does.
The quality of test cases produces by genetic algorithms is higher than the quality of test cases produced by random way because the algorithm can direct the generation of test cases to the desirable range fast. This paper shows that genetic algorithms are useful in reducing the time required for lengthy testing meaningfully by generating test cases for path testing. Furthermore, we build our Genetic Algorithm for structural testing for reduce execution time & generate more suitable test cases.

Acknowledgment

The authors wish to acknowledge UGC for the award of Research Fellowship under Fellowship in Sciences to Meritorious Students (RFSMS) scheme for carrying out this research.

References

[1] Roger S. Pressman: “Software Engineering”, A Practitioner’s Approach 5th Edition, McGraw Hill, 1997.

[2] B. Beizer, Software Testing Techniques 2nd Edition, International Thomson Computer Press, 1990.

[3] Srinivasan Desikan, Gopalaswamy Ramesh “Software Testing Principles & Practices” PEARSON Education, 2006.

[4] G. J. Myers, The Art of Software Testing.2nd ed.: John Wiley & Sons Inc, 2004.

[5] B. Antonia, "Software Testing Research: Achievements, Challenges, Dreams," in 2007 Future of Software Engineering: IEEE Computer Society, 2007.

[6] Chen Yong and Zhong Yong, "Automatic Path-Oriented Test Data Generation Using a Multi-population Genetic algorithm,"in Proceedings of Fourth International Conference on Natural Computation (ICNC '08), Jinan, China, 2008.

[7] Chen Yong, Zhong Yong, Bao Shengli, and He Famei, "Structural Test Data Generation Using Immune Genetic Algorithm," in The International Conference 2007 on Information Computing and Automation, Chengdu, China, 2008.

[8] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style: McGraw-Hill, Inc. New York, NY, USA, 1982.

[9] E. J. Weyuker, "The applicability of program schema results to programs," International Journal of Parallel Programming, vol. 8, 1979,pp. 387-403.

[10] T. Y. Chen, T. H. Tse, and Z. Zhiquan, "Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing," in Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis Roma, Italy: ACM, 2002.

[11] S. Nguyen Tran and D. Yves, "Consistency techniques for interprocedural test data generation," ACM SIGSOFT Software Engineering Notes, vol. 28, 2003, pp. 108-117.

[12] C. K. James, "A new approach to program testing," in Proceedings of the international conference on Reliable software Los Angeles, California: ACM, 1975.

[13] G. M. C C Michael, M Schatz "Generating software test data by evolution," IEEE Transactions on Software Engineering, vol. 27, 2001, pp. 1085-1110.

[14] B. Korel, "Automated software test data generation," IEEE Transactions on Software Engineering, vol. 16, 1990,pp. 870- 879.

[15] B. Korel, "Dynamic method for software test data generation," Software Testing, Verification & Reliability, vol. 2, 1992, pp. 203-213.

[16] J. Wegener, B. Kerstin, and P. Hartmut, "Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing," in Proceedings of the Genetic and Evolutionary Computation Conference: Morgan Kaufmann Publishers Inc., 2002.

[17] W. Joachim, Andr, Baresel, and S. Harmen, "Suitability of Evolutionary Algorithms for Evolutionary Testing," in Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment: IEEE Computer Society, 2002.

[18] Christoph C. Michael, Gary McGraw and Michael A. Schatz, “Generating Software Test Data by Evolution”, IEEE Transactions On Software Engineering, Vol. 27, No. 12, December 2001.

[19] Roy P Pargas, Mary Jean Harrold, Robert R Peck, “ Test Data Generation Using Genetic Algorithms”, Journal of Software Testing, Verification and Reliability, 1999,

[20] Alan C. Schultz, John J. Grefenstette, aid Kenneth A. De Jong, “Test And Evaluation by Genetic Algorithms”, IEEE, 1993.

[21] Joachim Wegener, Kerstin Buhr, Hartmut Pohlheim, “Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing”.

[22] Yong Chen1, Yong Zhong, Tingting Shi1 and Jingyong Liu, “Comparison of Two Fitness Functions for GA-based Path-Oriented Test Data Generation”, 2009 Fifth International Conference on Natural Computation, IEEE, 2009.

[23] Richard A. DeMillo and A. Jefferson Offutt, “Constraint-Based Automatic Test Data Generation”, 1EEE Transactions On Software Engineering, Vol. 17, No. 9, September 1991.

[24] Jin-Cherng Lin and Pu-Lin Yeh, “Using Genetic Algorithms for Test Case Generation in Path Testing”, IEEE, 2000.

[25] Debasis Mohapatra, Prachet Bhuyan and Durga P. Mohapatra, "Automated Test Case Generation and Its Optimization for Path Testing Using Genetic Algorithm and Sampling", WASE International Conference on Information Engineering, 2009.

[26] Donald J. Berndt and Alison Watkins, "Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation", Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04) 2004.

[27] Xiajiong Shen, Qian Wang, Peipei Wang and Bo Zhou, "Automatic Generation of Test Case based on GATS Algorithm", AA04Z148, 2007.

[28] Yong Chen1, Yong Zhong, Tingting Shi1, Jingyong Liu, "Comparison of Two Fitness Functions for GA-based Path-Oriented Test Data Generation" IEEE 2009 Fifth International Conference on Natural Computation

Authors Profile

Mr. Premal B. Nirpal
UGC Research Fellow,
Department of Computer Science and Information Technology,
Dr. B. A. M. University, Aurangabad.

Dr. K. V. Kale
Professor and Head,
Department of Computer Science and Information Technology,
Dr. B. A. M. University, Aurangabad.