AUTOMATED TEST DATA GENERATION USING GENETIC ALGORITHMS

***Pankaj Saxena, #Vinay Singh**

**Reader, Department of Computer Application, FMCA, R.B.S.College, Agra*

#Asstt. Professor, Department of CS/IT, Anand Engineering College, Agra

ABSTRACT

In software testing, it is often desirable to find test inputs that can handle specific features of the program. To find these inputs by hand is extremely time-consuming, especially when the software is complex. Therefore, many attempts have been made to automate the process. Random test data generation consists of generating test inputs at random, in the hope that they will exercise the desired software features. Often, the desired inputs must satisfy complex constraints, and this makes a random approach seem unlikely to succeed. In contrast, combinatorial optimization techniques, such as those using genetic algorithms, are meant to solve difficult problems involving the simultaneous satisfaction of many constraints. This paper presents a technique that uses a genetic algorithm for automatic test-data generation. A genetic algorithm is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. In test data generation application, the solution sought by the genetic algorithm is test data that causes execution of a given statement, branch, path, or definition-use pair in the program under test.

Keywords: *Software Testing, Chromosome, Allele, Test Data*

INTRODUCTION

In software testing, one is often interested in judging how well a series of test inputs tests a piece of code — good testing means uncovering as many faults as possible with a set of tests. Thus, a test series that has the potential to uncover many faults is better than one that can only uncover a few. Unfortunately, it is almost impossible to say quantitatively how many faults are potentially uncovered by a given test set. This is not only because of the diversity of the faults themselves, but because the very concept of a ‑fault‖ is only vaguely defined. This has lead to the development of test adequacy criteria —criteria that are believed to distinguish good test sets from bad ones. Once a test adequacy criterion has been selected, the question that arises next is how one should go about creating a test set that is ‑good‖ with respect to that criterion [1]. Since this can be difficult to do by hand, there is an obvious need for automatic test data generation. Test data must be generated for feasible paths. A path is feasible if there exists some input that will cause the path to be traversed during execution [2].

A system fails when it does not meet its specification [3]. The purpose of testing a system is to discover faults that cause the system to fail rather than proving the code correctness, which is often an impossible task [4]. In the software testing process, each test case has an identity and is associated with a set of inputs and a list of expected outputs [5]. Functional (black-box) test cases are based solely on functional requirements of the tested system, while structural (white-box) tests are based on the code itself. According to [5], black-box tests have the following two distinct advantages: they are independent of the software implementation and they can be developed in parallel with the implementation. The number of combinatorial black-box tests for any non-trivial program is extremely large, since it is proportional to the number of possible combinations of all input values. On the other hand, testing resources are always limited, which means that the testers have to choose the tests carefully from the following two perspectives:

1. Generate good test cases. A good test case is one that has a high probability of detecting an as-yet undiscovered error [6]. Moreover, several test cases causing the same bug may show a pattern that might lead the programmer to the real cause of the bug.
2. Prioritize test cases according to a rate of fault detection – a measure of how quickly those test cases detect faults during the testing process [7].

This paper is concerned with the problem of generating test data for a particular program automatically that works on the principle of Genetic algorithms. There have been a lot of attempts over the years to develop a tool to automatically generate the test data. Research has highlighted the approach to generate test data automatically using genetic algorithm.

GENETIC ALGORITHMS: AN OVERVIEW

Genetic Algorithms (GAs) are general-purpose search algorithms, which use principles inspired by natural genetics to evolve solutions to problems. As one can guess, genetic algorithms are inspired by Darwin's theory about evolution [8]. They have been successfully applied to a large number of scientific and engineering problems, such as optimization, machine learning, automatic programming, transportation problems, adaptive control, etc. GA starts off with population of randomly generated chromosomes, each representing a candidate solution to the concrete problem being solved, and advances towards better chromosomes by applying genetic operators based on the genetic processes occurring in nature. So far, GAs have had a great measure of success in search and optimization problems due to their robust ability to exploit the information accumulated about an initially unknown search space. Particularly GAs specialize in large, complex and poorly understood search spaces where classic tools are inappropriate, inefficient or time consuming [9]. As mentioned, the GA's basic idea is to maintain a population of chromosomes. This population evolves over time through a successive iteration process of competition and controlled variation. Each state of population is called generation. Associated with each chromosome at every generation is a fitness value, which indicates the quality of the solution, represented by the chromosome values. Based upon these fitness values, the selection of the chromosomes, which form the new generation, takes place. Like in nature, the new

chromosomes are created using genetic operators such as crossover and mutation. The fundamental mechanism consists of the following stages:

1. Generate randomly the initial population.
2. Select the chromosomes with the best fitness values.
3. Recombine selected chromosomes using crossover and mutation operators.
4. Insert offsprings into the population.
5. If a stop criterion is satisfied, return the chromosome(s) with the best fitness.
6. Otherwise, go to Step 2.

AUTOMATED TEST DATA GENERATION SYSTEM

The genetic algorithm is used to generate the test data automatically.
To apply genetic algorithm to a particular problem, such as test case generation, we need to determine the following elements [10]:

1. Genetic representation for potential solutions to the problem (e.g., test cases).
2. Method to create an initial population of potential solutions.
3. Evaluation function, which scores the solution quality (also called objective function).
4. Genetic operators that alter the composition of the off-springs.
5. Values of various parameters used by the genetic algorithm (population size, probabilities of applying genetic operators, etc.).

Each element is briefly discussed below.
Representation refers to the modeling of chromosomes into data structures. Once again terminology is inspired by the biological terms, though the entities genetic algorithm refers to are much simpler than the real biological ones. *Chromosome* typically refers to a candidate data solution to a problem, often encoded as a bit string.
Each element of the chromosome is called *allele*. In other words, chromosome is a sequence of alleles. For example, consider 1-dimension binary representation: each allele is 0 or 1, and a chromosome is a specific sequence of 0 and 1's.
Initialization. This genetic operator creates an initial population of chromosomes, at the beginning of the genetic algorithm execution.
The *selection* operator is used to choose chromosomes from a population for mating. This mechanism defines how these chromosomes will be selected, and how many offsprings each will create. The expectation is that, like in the natural process, chromosomes with higher fitness will produce better offsprings. Therefore, selecting such chromosomes at higher probability will eventually produce better population at each iteration of the algorithm. Classic selection methods are *Roulette-Wheel*, *Rank based*, *Tournament*, *Uniform*, and *Elitism*. A tournament selection operator is used in this implementation.
The *crossover* operator is practically a method for sharing information between two chromosomes: it defines the procedure for generating an offspring from two parents. The crossover operator is considered the most important feature in the GA, especially where building

blocks (i.e. schemas) exchange is necessary. The crossover is data type specific, meaning that its implementation is strongly related to the chosen representation. It is also problem-dependent, since it should create only feasible offspring's. A *Two-point crossover* is used: a two random points are selected and all the bits between these two points get exchanged between the two parents.

The *mutation* operator alters one or more values of the allele in the chromosome in order to increase the structural variability. This operator is the major instrument of the genetic algorithm to protect the population against premature convergence to any particular area of the entire search space [11]. Unlike crossover, mutation works with only one chromosome at a time. In most cases, mutation takes place right after the crossover, so it practically works on the offspring's, which resembles the natural process. The most common mutation methods are:

1. Bit-flip mutation: given chromosome, every bit value changes with a mutation probability.
2. Uniform mutation: choose one bit randomly and change its value.

IMPLEMENTATION

Evaluation function, also called objective function, rates the candidate solutions quality. This is the only single measure of how good a single chromosome is compared to the rest of the population. The fitness function calculates the fitness of each chromosome according to given fitness function[12,13]. It may be possible to get some or all elements optimal in the first generation as first generation get produced randomly. In such cases the system preserves them and performs GA [14] on remaining chromosomes.

The list of common GA parameters is given below:

1. *Population size* – this parameter defines the size of the population, which may be critical in many applications: If *N* is too small, GA may converge quickly, whereas if it is too large the GA may waste computational resources.
2. *Chromosome length* – defines the number of allele within each chromosome. This number is influenced by the chosen representation and the problem being issued.
3. *Number of generations* – defines the number of generations the algorithm will run. It is frequently used as a stopping criterion.
4. *Crossover probability (Pc)* – the probability of crossover between two parents. The crossover probability has another trade-off: if *Pc* is too low, then the sharing of information between high fitness chromosomes will not take place, hence reducing their capability to produce better offsprings.
5. *Mutation probability (Pm)* – the probability of mutation in a given chromosome. As mentioned earlier, this method element helps to prevent the population from falling into local extremes, but a too high value of *Pm* will slow down the convergence of the algorithm.

RESULTS

The system is tested with the following program:

```
main()
{
int x, y, p=1;
do      {
                scanf("%d%d",&x,&y);
        }while(y<0);
for(i=1;i<=y;i++)
        {
                p=p*x;
        }
if(p>7000)
        {
                printf("\nx^y is %d",p);
        }
}
```

The code is written in MATLAB and the conditions given in the program are:

1. The value of x^y >7000
2. x^y should be positive

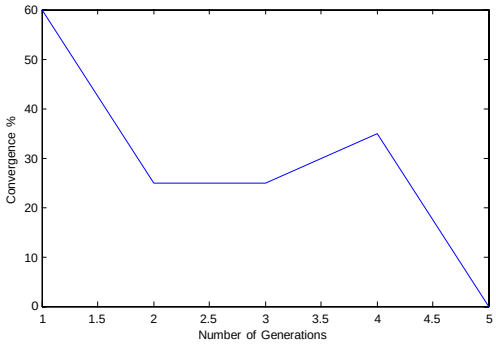
The crossover probability Pc is 0.7

The Mutation Probability Pm is 0.3

	Gen 1		
x	**y**	**Probabilit**	**Operation**
2	5	<0.3	Mutation
5	3	<0.3	Mutation
2	12	<0.7	Crossover
8	0	<0.3	Mutation
	Gen 2		
66	5	1	Correct
5	67	1	Correct
2	12	0.7	Crossover
9	0	0.3	Mutation
	Gen 3		
66	5	1	Correct
5	67	1	Correct
2	12	<0.7	Crossover
1	0	<0.3	Mutation
	Gen 4		
66	5	1	Correct
5	67	1	Correct
2	5	<0.3	Mutation
1	64	<0.3	Mutation
	Gen 5		
66	5	1	Correct
5	67	1	Correct
18	5	1	Correct
3	64	1	Correct

Table 1: Results after 5 generations

After 5 generations we get the correct test cases for the given program.



CONCLUSION

The results show that the system can be satisfactorily used to find feasible test cases. It identifies the feasible paths and tries to find the solutions for those paths. Results indicate that the technique is effective at producing potential test cases automatically. Thus it finds optimal test data automatically and much less time. The testing becomes easier as the test data gets generated automatically.

REFERENCES

1. Korel B., Automated Software Test Data Generation. IEEE transaction on Software Engineering (16)8:870-879, August, 1990.
2. Jasper R.,Brennan M.,Williamson K., Currier B., Test Data Generation and Feasible path analisys. 1994
3. Pfleeger, S. L.: Software Engineering: Theory and Practice. 2nd Edition, Prentice-Hall, 2001.
4. DeMillo, R.A. & Offlut, A.J.: Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering 17, 9 (1991) 900-910.
5. Jorgensen, P. C.: Software Testing: A Craftsman's Approach. Second Edition, CRC Press, 2002.
6. Schroeder P. J., and Korel, B.: Black-Box Test Reduction Using Input-Output Analysis. In Proc. of ISSTA '00 (2000). 173-177.
7. Elbaum, S., Malishevsky, A. G., Rothermel, G.: Prioritizing Test Cases for Regression Testing, in Proc. of ISSTA '00 (2000). 102-112.
8. Holland, J. H.: Genetic Algorithms, Scientific American, 267(1) (1992) 44-150.

9. Herrera F., and Magdalena, L.: Genetic Fuzzy Systems: A Tutorial. Tatra Mt. Math.
10. Publ. (Slovakia), 13, (1997) 93-121
11. Michalewicz, Z.: Genetic Algorithms + Data Structures - Evolution Programs, Verlag, Heidelberg, Berlin, Third Revised and Extended Edition, 1999.
12. Mitchell, M.: An Introduction to Genetic Algorithms, MIT Press, 1996.
13. Srivastava P. R., Kim T., Application of Genetic Algorithm in Software Testing, International Journal of Software Engineering and its Applications, Vol.3, No.4, October 2009.[13].Malhotra, R. and Garg, M., 2011. An adequate based test data generation techniques usinggenetic algorithm. Journal of Information Processing Systems,Vol. 7, Issue 2, June 2011. [14].Andrew.,J.H..2011.Genetic algorithm for randomized unit testing. Software Engineering, IEE jan-feb'2011, vol.37, Issue.1,pp 80-94