



International Journal of Emerging Technologies in Computational and Applied Sciences (IJETCAS)

www.iasir.net

Automatic Test Suit generation with Genetic Algorithm

Rakesh Kumar¹, Surjeet Singh², Girdhar Gopal³
Professor/DCSA¹, Assistant Professor², Research Scholar/DCSA³
KUK¹, GMN College Ambala Cantt.², KUK³
Kurukshetra, Haryana,
INDIA

Abstract: Software testing is most effort consuming phase in software development. One would like to minimize the efforts and maximize the number of faults detected. Hence test case generation may be treated as an optimization problem. One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. Generating test cases automatically will reduce cost and efforts significantly. In this paper, test case data is generated automatically using Genetic Algorithms and results are compared with Random Testing. It is observed that Genetic Algorithms outperforms Random Testing.

Keywords: Automatic test data generation, Equivalence Class Partitioning, Evolutionary algorithms, Random Testing, Software testing.

I. Introduction

Software development consists of various phases like Requirement analysis, Design, Coding and Testing. Out of these testing consumes maximum efforts. The software is tested with enough set of test cases, to make a judgment about quality or acceptability and to discover errors. Two fundamental techniques used to identify test cases are functional and structural testing [1]. Testing of software using these two approaches is very time consuming. So to reduce the efforts there should be a mechanism to generate test cases automatically. A number of techniques are available to generate automatic test cases like random testing, anti-random testing etc. Objective of all these techniques is to find minimal number of test cases to test the software fully. This can be considered as an optimization problem. To solve optimization problems there are a number of techniques and one of them is Genetic Algorithms. Genetic algorithms are population based search based on the Darwin's principle of *survival of the fittest*. GA is basically an evolutionary technique inspired by biological evolution. It was developed in 1970's by J. Holland and his colleagues and his students at University of Michigan's [2], [3]. It mimics the process of natural evolution. GA starts with a initial population and then apply genetic operators like selection, crossover, mutation and replacement on that population to evolve better and better individuals. GA can be terminated in either of two cases: maximum number of generations achieved or optimum value found.

II. Problem Statement

The most significant weakness of testing is that the postulated functioning of the tested system can, in principle, only be verified for those input situations which were selected as test data. Testing can only show the existence but not the non-existence of errors, [4]. Proof of correctness can only be produced by a complete test, i. e. a test with all possible input values, input value sequences, and input value combinations under all practically possible constraints. In practice, complete testing is usually impossible because of the vast amount of possible input situations. Testing can therefore only be a sampling method. Accordingly, the selection of an appropriate sample containing the most error-sensitive test data is essential to testing. If test data relevant to the practical deployment of the system are omitted, the probability of detecting errors within the software declines. Of all the testing activities – test case design, test execution, monitoring, test evaluation, test planning, test organization, and test documentation – essential importance is thus attributed to test case design, [5].

Software Testing accounts for approximately 50% of total software cost, [6]. This cost could be reduced if the process of testing is automated. In the past, a number of different methods for generating test data have been presented. These methods are divided in three classes: *Random*, *path-oriented* and *goal oriented* test data generation, [7]. a typical test data generator system consists of three parts: program analyzer, path selector and test data generator. The source code is run through a program analyzer, which produces the necessary data used by the path selector and the test data generator. The selector inspects the program data in order to find suitable paths. Suitable can for instance mean paths leading to high code coverage. The paths are then given as argument to the test data generator which derives input values that exercise the given paths. The generator may provide the selector with feedback such as information concerning infeasible paths.

Due to the non-linearity of software (conditional and looping statements etc.), the conversion of test problems into optimization tasks usually results in complex, discontinuous, and non-linear search spaces. Neighborhood search methods such as hill climbing are not suitable in such cases. Therefore, meta-heuristic search methods, such as evolutionary algorithms, are employed. The suitability of evolutionary algorithms for testing is based on their ability to produce effective solutions for complex and poorly understood search spaces with many dimensions. The dimensions of the search spaces are directly related to the number of input parameters of the system under test. The execution of different program paths and the nested structures in software systems lead to multi-model search spaces when testing.

In order to automate software tests using evolutionary algorithms, the test aim must itself be transformed into an optimization task. A numeric representation of the test aim is necessary, from which a suitable fitness function for the evaluation of the generated test data can be derived. Depending on which test aim is pursued, different fitness functions emerge for test data evaluation. If an appropriate fitness function can be defined for the test aim, and evolutionary computation is applied as the search technique, then the Evolutionary Test proceeds as follows. The initial set of test data is generated, usually at random. In principle, if the test data has been obtained by a previous systematic test, this could also be used as an initial population, [8]. The Evolutionary Test could thus benefit from the tester's knowledge of the system under test. Each individual within the population represents a test datum with which the system under test is executed. For each test datum the execution is monitored and the fitness value determined for the corresponding individual.

Next, test data with high fitness values are selected with a higher probability than those with a lower value and are subjected to combination and mutation processes to generate new offspring test data. It is important to ensure that the test data generated are in the input domain of the test object. The main idea behind evolutionary testing is the combination of interesting test data in order to generate offspring test data that truly fulfill the test objectives. The offspring test data are evaluated by executing the system under test. A new population of test data is formed by merging offspring and parent individuals according to the survival procedures laid down. From here on, the process repeats itself, starting with selection until the test objective is fulfilled or another given stopping condition is reached

III. Related Work

Random testing is the simplest technique of test data generation. Actually it could be used to generate data for any type of program since; ultimately, every data is a string of bits. But random testing mostly does not perform well in terms of coverage. Since it merely relies on probability it has quite low chances in finding semantically small faults [9], and thus accomplish high coverage. A fault that is only revealed by small percentage of program input is called semantically small fault. For example, in following code:

```
void foo(int var1, int var2)
{
    if (var1 == var2)      print("EQUAL");           // statement 1
    else                  print("NOT EQUAL");        // statement 2
}
```

The probability of executing *statement 1* is $1/n$, where n is the maximum integer, since to execute *statement 1*, both x and y must be same. So random testing can generate these types of test data at very low probability.

Random testing selects test data randomly from the input domain and then test the program with these test cases. The automatic production of random test data, drawn from an uniform distribution, should be the default method by which other systems should be judged, [10]. Statistical testing is a test case design technique in which the tests are derived according to the expected usage distribution profile.

The distribution of selected input data should have the same probability distribution of inputs which will occur in actual use in order to estimate the operational reliability, [11].

There is not much difference between partition and random testing in terms of finding faults, [12]. Hamlet showed that random testing is superior to partition testing with regard to human effort especially with more partitions and if confidence is required. For a small number of sub-domains partition testing will perform better than random testing.

Random number generators are ineffective in that they rarely provide the necessary coverage of the program, [13].

This comment was strengthened and new opinion that random testing is probably the poorest methodology in testing was given, [14].

However, many errors are easy to find, but the problem is to determine whether a test run failed. Therefore, automatic output checking is essential if large numbers of tests are to be performed, [15], [16]. They also said that partition testing is more expensive than performing an equivalent number of random tests which is more cost effective because it only requires a random number generator and a small amount of software support. The change of range for random testing has a great effect, [15]. Further they mentioned a disadvantage of random testing which is to satisfy equality values which are difficult to generate randomly.

The advantage of random testing is normally that it is more stressing to the program under test than hand selected test data, but on the other hand random inputs may never exercise both branches of a predicate which tests for equality, [17]. Even in the case that random testing is cheaper than partition testing, the slight advantage of random testing could be compensated for by using more random tests and there is no assurance that full coverage can be obtained, e.g. if equality between variables are required. And secondly it may mean examining the output from thousands of tests.

Random testing was especially recommended for the final testing stage of software by Tsoukalas [18] and Girard and Rault [19].

Duran and Ntafos [15] recommended a mixed final testing, starting with random testing, followed by a special value testing method (to handle exceptional cases). Ince [10] reported that random testing is a relatively cheap method of generating initial test data.

IV. Research methodology

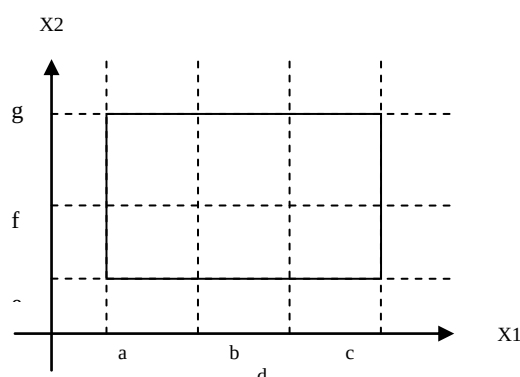
In past, it is observed that test cases lie in different classes. Equivalence classes are to form a partition of set, where partition refers to a collection of mutually disjoint subsets where the union is the entire set. This has two important implications for software testing: the fact that the entire set is represented provides a form of completeness and disjointedness ensures a form of non-redundancy. As the subsets are determined by an equivalence relation, the elements of one subset have something in common. So the idea is to identify test cases by using one element from each equivalence class. If the classes were chosen wisely, it greatly reduces the potential redundancy in test cases. For example for an equilateral triangle test case, if we chose (3, 3, 3) as test case, then we would not expect to learn much from (6, 6, 6) or (50, 50, 50). For the sake of drawings, a function F of two variables x_1 , x_2 will be used [1]. When F is implemented followings are the boundaries & intervals for the values of x_1 and x_2 : -

$$a \leq x_1 \leq d, \text{ with intervals } (a,b), (b,c), (c,d)$$

$$e \leq x_2 \leq g, \text{ with intervals } (e,f), (f,g)$$

Invalid values for x_1 and x_2 are $x_1 < a$, $x_1 > d$ and $x_2 < e$, $x_2 > g$.

Figure 1: Equivalence Class Partitioning for variable boundaries [1]



For a more general example equivalence class partitions for a nextDate module, which return the very next date of the entered current date. Can be done as follows: -

It is a function of three variables and the boundaries are as follows: -

$$M1 = \text{month } (1 \leq \text{month} \leq 12)$$

$$D1 = \text{date } (1 \leq \text{date} \leq 31)$$

$$Y1 = \text{year } (1951 \leq \text{year} \leq 2051)$$

The invalid equivalence classes were: -

M2 = month < 1, M3 = month > 12

D2 = date < 1, D3 = date > 31

Y2 = year < 1951, Y3 = year > 2051

So the robust test cases with equivalence class test may be as follows:

Month	Date	Year	Remarks
5	15	1962	All valid inputs
-1	15	1962	M2 class
15	15	1962	M3 class
5	-1	1962	D2 class
5	45	1962	D3 class
5	15	1900	Y2 class
5	15	2100	Y3 class

In this research, the researcher carried out the identification of these boundaries/intervals automatically through Genetic algorithm and random testing and then compares the results of both techniques. Genetic algorithm and random testing both starts with some random initial population and then GA use the fitness of individuals to progress towards the optimums, whereas random testing works randomly throughout the run. For this experiment, the distance from the boundaries is taken as the fitness of the individual chromosome. The algorithms are coded in MATLAB 2011a.

A. Genetic Algorithm for test case generation

The proposed genetic algorithm for test case generation for Equivalence Class Partitioning is presented here. Firstly, the major components of GA are discussed and then overall algorithm is presented.

Fitness of each chromosome is determined by its difference from the boundaries of the variable. The more a variable is close to the boundaries the more it is declared fit.

Fitness(popsize, chromLength, curpop)

lBound = lower boundary of the variable;

uBound = upper boundary of the variable;

centre = (lbound + ubound) / 2;

diff = centre - lbound;

for I = 1 to popsize

for j = 1 to chromLength

c1 = lbound - diff; c2 = lbound + diff; c3 = ubound + diff;

diff1 = c1 - curpop(I,j); diff2 = c2 - curpop(I,j); diff3 = c3 - curpop(I,j);

moreClose = min(diff1, diff2, diff3);

fitness(i) = moreClose;

end

end

end;

Following parameters are used in experiments:-

- **Population Size:** various population sizes are tried and best ones are taken for comparison i. e. 10, 20, 50 & 100.

- **Generations:** program is executed with different number of generations and analysis of less number of generations and more number of generations is also taken into consideration i. e. 100, 200, 500 & 1000.
- **Encoding:** chromosomes (test cases) are coded in real values, so Value encoding scheme of GA is used.
- **Selection:** Roulette wheel selection is used for Genetic Algorithm, and Random selection is implemented for Random Testing.
- **Crossover:** number of crossovers available for real value coding, out of which arithmetic crossover is applied with 0.7 probability.
- **Mutation:** uniform mutation is applied in experiments with 0.1 probability.
- **Replacement:** Simple genetic algorithm replacement takes place, in which whole new population replaces the old one.

V. Results & Observations:

All inputs are taken from user, so that testing with different parameters can be done easily. User interface while running in MATLAB is as follows: -

INPUTS:

No. of individuals in population : 30, No. of Variables : 2, No. of Generations : 500

limits of 1st variable : Lower limit : 5 & Upper limit : 15

limits of 2nd variable : Lower limit : 6 & Upper limit : 16

OUTPUTS: With Genetic algorithm, test cases generated as follows:

	Genetic Algorithm		Random Testing	
Variables / Runs	Variable 1	Variable 2	Variable 1	Variable 2
1	9.25	9.89	6.52	7.85
2	16.52	2.53	12.86	14.78
3	8.56	12.53	11.23	10.85
4	3.85	22.52	9.86	6.98
5	5.65	15.95	12.56	7.39

Figure 2 to Figure 6 explains these executions

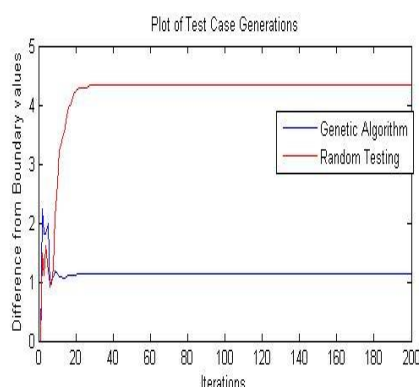


Figure 2: Experimental Results

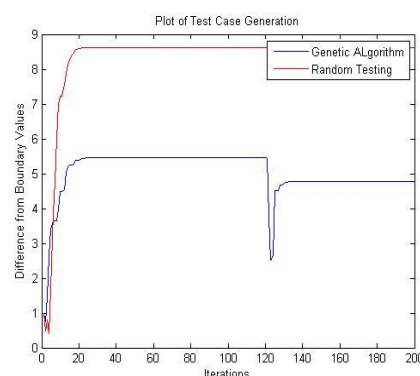


Figure 3: Experimental Results

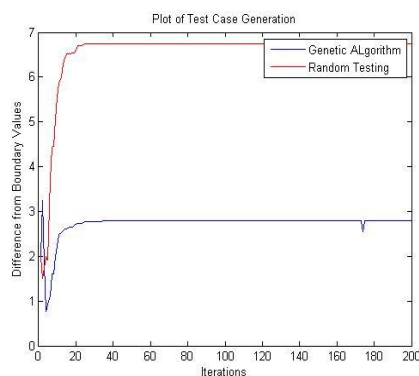


Figure 4: Experimental Results

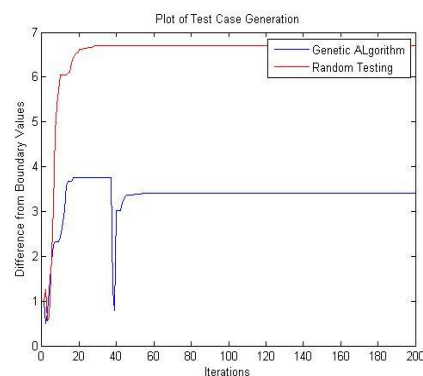


Figure 5: Experimental Results

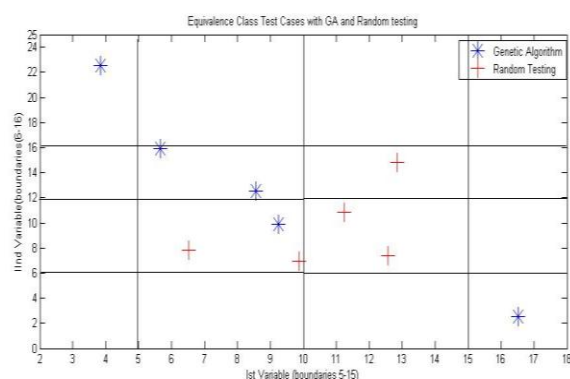


Figure 6: Results showing the final test cases analysis

Another run is carried out with following inputs and outputs:

INPUTS:

No. of individuals in population : 20, No. of Variables : 2, No. of Generations : 200

Limits of 1st variable : Lower limit : 2 & Upper limit : 20

limits of 2nd variable : Lower limit : 10 & Upper limit : 20

OUTPUTS: With Genetic algorithm, test cases generated as follows:

Variables / Runs	Genetic Algorithm		Random Testing	
	Variable 1	Variable 2	Variable 1	Variable 2
1	5.25	12.79	15.25	11.92
2	-2.62	25.63	6.52	16.53
3	16.63	15.88	10.65	19.85
4	26.52	5.52	12.79	16.75
5	10.62	2.53	2.95	11.75

Figure 7 to Figure 11 explains these executions

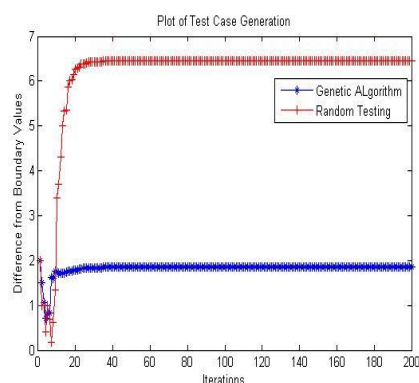


Figure 7: Experimental Results

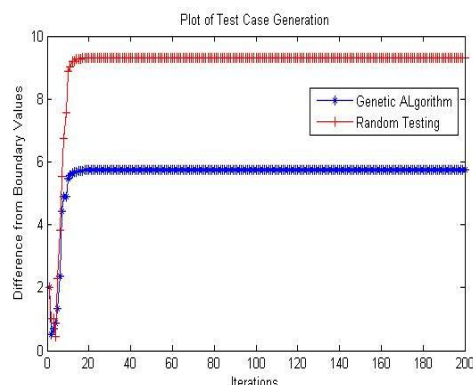


Figure 8: Experimental Results

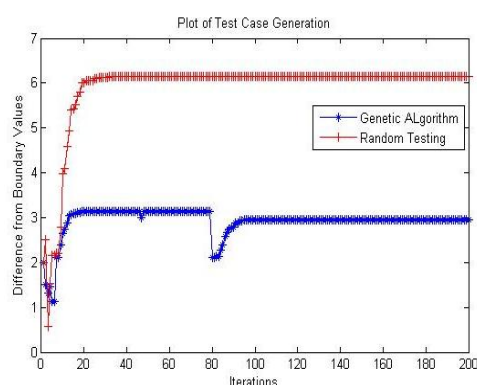


Figure 9: Experimental Results

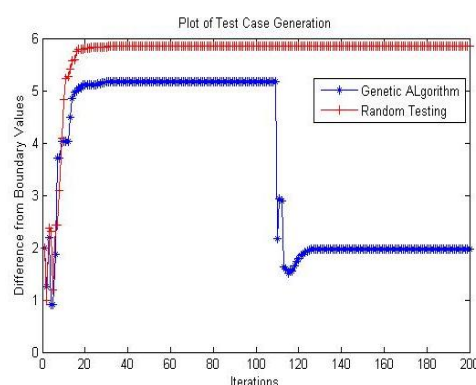


Figure 10: Experimental Results

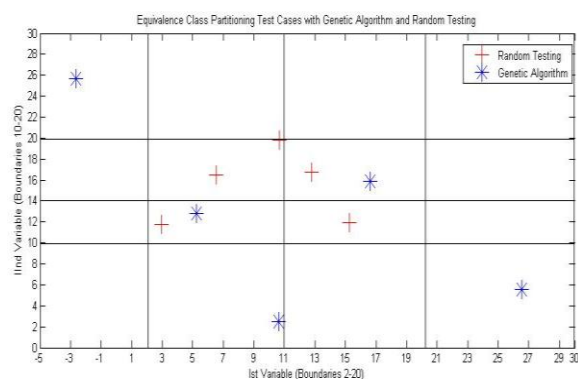


Figure 11: Results showing the final test cases analysis

It is clearly visible from experiments that test cases with genetic algorithms spread over all classes, whereas random testing will generate test cases for small number of equivalence classes. So one can use Genetic algorithm to generate test cases automatically and get better and useful test cases as outputs.

VI. CONCLUSION

The overall results show evolutionary testing to be a promising approach for fully automating test case design for equivalence class partitioning technique of testing. To increase the efficiency and effectiveness, and thus to reduce the overall development cost for software-based systems, a systematic and automatic test case generator is required. Genetic algorithms search for relevant test cases in the input domain of the system under test. Due to the full automation of test case generation, the overall quality of software is also enhanced in comparison of using random testing. The application scope of evolutionary test case generation can go further than the work described above. Additional application fields can be control flow graphs, path testing, stress testing etc. Actually, every technique of testing can be implemented using Genetic Algorithms automatic test case generation.

References

1. Jorgenson C. Paul, "*Software Testing: A Craftsman Approach*", CRC Press, 2nd Edition 2002 pp 97-103.
2. Goldberg D. E., "*Genetic algorithms in search, optimization, and machine learning*", Addison Wesley Longman, Inc., ISBN 0-201-15767-5, 1989.
3. Holland J., "*Adaptation in natural and artificial systems*", University of Michigan Press, Ann Arbor, 1975.
4. Dijkstra, E. W., Dahl, O. J., Hoare, C. A. R.: "*Structured programming*", Academic Press., 1972.
5. Wegener, J. and Pitschinetz, R.: "*TESSY – Yet Another Computer-Aided Software Testing Tool?*" Proceedings of the Second International Conference on Software Testing, Analysis and Review, Bruxelles, Belgium, 1994.
6. Beizer B. "*Software Testing Techniques*". Van Nostrand Reinhold, 2nd edition, 1990.
7. Ferguson R. and Korel B. "*The chaining approach for software test data generation*". IEEE Transactions on Software Engineering, 5(1):63-86, January 1996.
8. Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H. and Jones, B.: "*Systematic Testing of Real-Time Systems*". Proceedings of the Fourth European International Conference on Software Testing, Analysis & Review, Amsterdam, Netherlands, 1996.
9. Offutt J. and Hayes J., "*A semantic model of program faults*". In International Symposium on Software Testing and Analysis (ISSTA 96), pages 195-200. ACM Press, 1996.
10. Ince, D. C.: "*The automatic generation of test data*", The Computer Journal, Vol. 30, No. 1, pp. 63-69, 1987.
11. Taylor R.: '*An example of large scale random testing*', Proc. 7th annual Pacific North West Software Quality Conference, Portland, OR, pp. 339-48, 1989.
12. Hamlet D. & Taylor R., "*Partition testing does not inspire confidence*", IEEE Transactions on Software Engineering, Vol. 16, 1990, pp. 1402-1411.
13. Deason, W. H., Brown, D. B., Chang, K. H., and II, J. H. C. (1991). "*A Rule-Based Software Test Data Generator*". *IEEE Trans. on Knowl. and Data Eng.*, 3(1):108-117.
14. Myers, G. J. (1979). *Art of Software Testing*. John Wiley & Sons.
15. Duran J. W. and Ntafos S. C.: '*An Evaluation of Random Testing*', IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pp. 438-444, July 1984
16. Duran, J. W. and Ntafos S., '*A report on random testing*', Proceedings 5th Int. Conf. on Software Engineering held in San Diego C.A., pp. 179-83, March 1981
17. Bertolino, A.: '*An overview of automated software testing*', Journal Systems Software, Vol. 15, pp. 133-138, 1991
18. Tsoukalas M. Z., Duran J. W. and Ntafos S. C.: '*On some reliability estimation problems in random and partition testing*', IEEE Transactions on Software Engineering, Vol. 19, No. 7, pp. 687-697, July 1993
19. Girard, E. and Rault, F. C.: '*A programming technique for software reliability*', IEEE Symp. Computer Software Reliability, pp. 44-50, 1973