International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878 (Online), Volume-8 Issue-6, March 2020

Software Test Data Generation based on Path Testing using Genetic Algorithms

Tina Sachdeva, Astha Pattanaik

Abstract*: *Test data generation is the task of constructing test cases for predicting the acceptability of novel or updated software. Test data could be the original test suite taken from previous run or imitation data generated afresh specifically for this purpose. The simplest way of generating test data is done randomly but such test cases may not be competent enough in detecting all defects and bugs. In contrast, test cases can also be generated automatically and this has a number of advantages over the conventional manual method. Genetic Algorithms, one of the automation techniques, are iterative algorithms and apply basic operations repeatedly in greed for optimal solutions or in this case, test data. By finding out the most error-prone path using such test cases one can reduce the software development cost and improve the testing efficiency. During the evolution process such algorithms pass on the better traits to the next generations and when applied to generations of software test data they produce test cases that are closer to optimal solutions. Most of the automated test data generators developed so far work well only for continuous functions. In this study, we have used Genetic Algorithms to develop a tool and named it TG-GA (Test Data Generation using Genetic Algorithms) that searches for test data in a discontinuous space. The goal of the work is to analyze the effectiveness of Genetic Algorithms in automated test data generation and to compare its performance over random sampling particularly for discontinuous spaces.

Keywords: Genetic Algorithms, Genetic Operators, Fitness Function, Software Testing, Test Adequacy Criteria, Test Data Generation.

I. INTRODUCTION

Software Testing is a vital and a time extensive process that consumes more than 50% of the software development resources and cost [10, 11]. Also, in the software industry constant novel assessment approaches and metrics are required for predicting the quality and reliability of the software by executing test cases. These test cases could be inputs to variables of the software, execution paths, execution conditions, or testing requirements. Thus, the quality of test cases plays a prominent role in testing and problems usually require them in large number. This calls for high manpower cost, and considerable amount of time for their generation. The most straightforward way to generate test cases is manual and random in nature but such test cases may not be capable enough to highlight all defects and errors. Moreover, manual generation is quite inefficient and software industry has constantly tried to automate this process. Therefore, automated and efficient generation of test cases is a potential and a critical research problem in the domain of testing.

Automation of test cases has a number of other advantages also besides from being fast and accurate. An automated generator may implement algorithms to generate correctly formatted special data like PAN (Permanent Account Number) numbers, email addresses, etc. This eliminates the need to look up algorithms for generating special test data by the tester. It can also create both valid and invalid test data and this quantity can be controlled by a percentage distribution between them. Such generators also provide options to either generate the test data directly in the desired format (e.g. Excel, CSV or SQL) or even export it to a desired format.

Evolutionary Algorithms, particularly, Genetic Algorithms are meta-heuristic search techniques that change the task of building up of test cases into an optimization task [2]. Genetic Algorithm is an adaptive search procedure based on the concept of selection and evolution. It is a computational technique that resembles biological evolution as a problem-solving approach [1]. It enhances a population of separate solutions in a recursive manner and for this at each step it randomly chooses individuals from the present population and uses them as parents to produce off springs for the next generation. The population moves towards an optimized result over consecutive generations. Thereafter, they search for optimal test parameter combinations that fulfill some pre-defined test condition which is represented using a fitness function. They behave well to problems of higher dimensions in short span of time and can be seamlessly modified to the new problem and can be changed for customization as well. Because of these inherent qualities they are a promising technique for test case generation.

In this paper we develop a tool named TG_GA, that looks for optimal solutions in a discontinuous search space and collects inputs for such solutions as test data. A discontinuous search space is one which has solutions that are isolated from each other in a graph and hence, they do not represent a continuous curve. Thus, problems like finding GCD of two numbers, greatest of 'n' numbers, division, etc., have discontinuous search spaces.

Manuscript received on March 15, 2020.
Revised Manuscript received on March 24, 2020.
Manuscript published on March 30, 2020.
* Correspondence Author
Tina Sachdeva*, Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi, Delhi, India. E-mail: sachdeva_tina@yahoo.com
Astha Pattanaik, Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi, Delhi, India. E-mail: asthapattnaik01@gmail.com

During the development process, we make a file *geneticdata.txt* and specify the range of permissible input variables in this. Initial discontinuous values of the individuals during the run are pseudo randomly generated in this specified range and the fitness function is computed for them. On several runs of the tool, observations show the suitability of such an approach using Genetic Algorithms for discontinuous spaces. This work was motivated by the fact that Software Testing consumes for more than 50% of the total software development cost.

Manual and random generation of test cases is a time intensive activity and produce test cases having low error detecting capability. Genetic algorithms were chosen as a testing method because in recent years they have grown in popularity in optimization of engineering problems [19].

II. LITERARTURE REVIEW

Over the years, many researchers and scientists have carried out extensive and in-depth study in the domain of software test data generation using evolutionary algorithms. Sharma et al [18] have suggested methods for increasing performance of Genetic Algorithms in search space exploration and exploitation fields with better convergence rate for test data generation. Al-Zabidi et al [8] have applied Genetic Algorithms successfully to software systems of different complexities for same. Nirpal and Gupta have successfully used Genetic Algorithms to generate data for the triangle classification problem [7]. Yang et. al. [14], presented an approach of generating test data for a specific single path based on Genetic Algorithms. A similarity between the target path and execution path with sub path overlapped is taken as the fitness value to evaluate the individuals of a population and drive Genetic Algorithms to search appropriate solutions. Srivastava and Tai-hoon, applied Genetic Algorithm technique to find the most critical paths in discontinuous spaces [15]. Xie et al [2] have presented dynamic optimization strategies for evolutionary testing, an efficient method for automated test case generation. Ghiduk et al [26] have suggested an approach to overcome inefficiency in covering multiple target paths. Due to full automation of evolutionary testing algorithms, more than several thousand data sets are generated and executed within few minutes therefore improving quality and reducing development cost [19].

III. TEST DATA GENERATION

Test data generation is the process of making test suites for software testing in order to determine the acceptability of a new or an altered system.

A. Static and Dynamic Test Data Generation

Techniques that do not require execution of the software under consideration are known as static test data generation techniques. In this, all the possible paths of the program are examined and traversed without actually running them. The software under test here acts as a passive entity. Static evaluation parameters are considered for evaluation.

In dynamic test data generation, the software under test is actually executed. It is ensured that every path is executed and if not, the control is backtracked to find out which statement diverted the flow of the program wrongly.

B. Random Test Data Generation

In this the test cases are created arbitrarily and software is executed using this data. It is the simplest of all approaches but does not ensures maximum error detection as random data may not cover all functionalities or internal structure or adequacy criteria. It just requires a random number generator and hence is easy and not expensive to carry out but has low chances of finding out semantically small bugs.

C. Symbolic Execution

Some old approaches of test data generation used mnemonic execution for creating test cases where symbols are assigned to variables in place of their actual values. In this the constraints based on inputs are specified which determines the conditions necessary for the traversal of the paths. One looks for paths and then only finds out the conditions that traverse those paths.

D. Automation in Test Data Generation

Automated generation of test cases refers to machine driven approach of creating test suites based on some functionality or inherent structure of the software under consideration [4]. Clearly, it has number of advantages over the conventional manual and random method of writing test cases. Automated test data generators, which are not random in nature, carry out test assessment based on some constraints popularly known as **test adequacy criteria**. In the domain of Genetic Algorithms, this condition is written in the form of a mathematical formula called as the fitness function.

Some popular test adequacy criteria are:

1. **Statement Coverage**: All statements of the software under test must be executed at least once.
2. **Branch Coverage**: All branches of the source code should be executed at least once.
3. **Condition Coverage**: Every conditional statement must be under test at least once.
4. **Path Coverage**: All paths of the software under test must be executed at least once.
5. **Independent Path Coverage**: All independent paths of the source code should be executed at least once.
6. Every point from the Software Requirements Specification must be executed at least once.
7. Every possible output of the program should be verified at least once. Every du (definition use) and dc (definition clear) path must come under test at least once.

IV. GENETIC ALGORITHMS

Genetic Algorithms are based on a heuristic search strategy and compute optimal solutions using operators that are derived from genetics of natural selection [9]. The most popular forms of evolutionary techniques are the Genetic Algorithms in which goal is steered by use of various combinations of input variables in order to satisfy the goal of testing. Such algorithms are based on biological genetic theory and Darwin's Principle of survival of the fittest.

Genetic Operators

A genetic operator is an operator used in biological as well as computational genetics to drive the algorithm towards an optimal solution for a given problem statement.

A. Selection:

It examines the fitness of an individual allowing the fitter one to transfer its genes to the next generation. More weightage is given to better individuals, permitting them to transfer their genes to the upcoming generation.

Fitness is computed using an objective or fitness function and the suitability of an individual is determined by this fitness. Few popular selection techniques are Roulette Wheel Selection, Rank Selection, Tournament Selection, etc. Right selection of initial population is vital for the convergence rate because better parents drive individuals to better and faster optimal values.

B. Crossover:

It is the interchanging of an allele of an individual with another one from a different individual. Any two candidates are selected from the existing population. The formula mentioned below is a proposed implementation of crossover [6]:

off spring1 = cr*p1 + (1-cr) *p2

off spring2 = (1-cr) *p1 + cr*p2

(cr: chromosome; p1: parent1; p2: parent2)

A crossover bit among all the bits is chosen (single point crossover). The instances of the two sub-strings are swapped till these selected bits are reached. If X=000111 and Y=111000 and the crossover point is 3 then X`=111111 and Y`=00000.The novel off springs born become part of the next generation. By recombination of good off-springs, Genetic Algorithms have a tendency to make even better individuals. Another variant of this, namely two-point crossover, involves selecting two random bits in the selected individuals and then swapping bits between these two genes (bits). Crossover is the important differentiating factor of Genetic Algorithms and operates at the individual level.

C. Mutation

Allele of genes is randomly replaced by another possible allele to produce a new individual. Under the binary system, this would imply that the bits of the individuals get complemented. The primary aim of mutating is to introduce variety into the population and avoid early untimely convergence to a local solution. For this, it spans through the entire search space randomly. It inducts individuals into the population which may not be originally present. Low mutation probability leads to insufficient global sampling.

Fitness Function

A fitness function is an objective function used to summarize and compare the closeness of a solution in achieving optimal solution. It accepts as input any candidate solution to the problem and finds out as resultant how much superior the solution is in contrast to the question in consideration. Computation of fitness value is done repeatedly and thus it must be reasonably fast. A slow computation of the fitness value can adversely affect a Genetic Algorithm and make it exceptionally slow. Every point in the search space has a fitness value. Any candidate that is closer to an optimal value possesses a higher value of this objective function as compared to one which is farther.

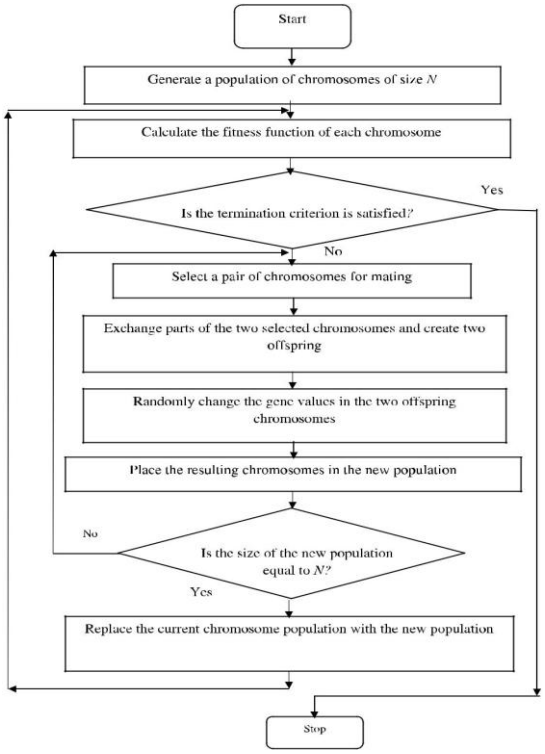

Fig. 1. Generic flowchart of Genetic Algorithm

V. GENETIC ALGORITHMS BASED APPROACH TO TEST DATA GENERATION

CFG (Control Flow Graph)

The Control Flow Graph is a graph that is used for pictorial representation of control structure of software. It shows the structure of flow or the path followed by the program during runtime. It is a directed graph (V, E) comprising of a set of vertices V and a set of directed edges E. It has a start node, end node, connection edges, decision nodes, junction nodes, and bounded regions.

- **Node**: It denotes procedural statements of the program. In the CFG, they are drawn using oval shape. They are either numbered or labeled.
- **Edges or links**: These are directed edges. They are drawn as arrows and show control flow from one node to another.
- **Decision Node**: It is a node with one or more arrows leaving from it corresponding to different outcomes of a decision.
- **Region**: The area bounded by some nodes and edges.

Path Testing Terminologies

Path: A lineage of instructions or statements covered during the execution of a program. In a CFG, it begins from a start node and stops at the end node with some other nodes and edges in between.

Independent Path: It is a path in which there must be at least one new statement or node or edge that is not traversed by any other existing paths.

Path Testing: In this a tester checks whether the given input covers the expected path or not.

VI. COMPARITIVE STUDY OF SIMILAR EXISTING TOOLS

Testing an application is one of the most important and time intensive tasks. Lately, automated test data generation tools have been used for populating the software under test, with test cases. Although, use of automated tools for test data generation is still in its early stages but accuracy is the main advantage that comes with it. Speed is also an important factor that makes this time intensive task faster. Data can be filled in during non-working hours wherein tester interaction is not required at all.

Advantages of automated test data generation tools include considerable savings in time, generation of more accurate data, ensuring that the data in question is high in volume and data in many different formats.

Table- I: Popular test data generation tools

Product Name	Vendor	Platform / Language compatibility	Short Description	Few Remarks
T-VEC Data Generator [30]	TVEC	Java and C++	Model-based functional test data generation, generates variety of HTML reports	Popular for ease of use
SQL Data Generator [31]	Red-Gate	My SQL Server	Creates realistic data based on column and table names	High success rate
DTM Data Generator [32]	GSApps	Support via OLE DB or ADO: MS SQL, DB2, text files, Informix, Oracle, Sybase, Access.	Automatically fills a database with test data for quality assurance testing	Support for stress testing, usability, software marketing
Advanced Data Generator [33]	Upscene Productions	Easy connectivity of Pro Edition to Firebird, MySQL, Inter Base,	Support for data types: large text, binary, numbers, integers, date & time, Boolean and GUIDs.	High compatibility
IBM DB2 Test Database Generator [34]	IBM	DB2	Creates realistic test data generation tools for database application	Generates test data from scratch or from existing data

VII. METHODOLOGY

This section provides a description of the technique and all parameter settings used in the tool developed and named TG_GA.

- Single point crossover operation was used in TG_GA and the crossover probability was computed as a scaled pseudorandom number R8 between 0 and 1. Further, the crossover point was computed as a pseudorandom number I4 between 0 and number of variables in each individual.

- In the mutation sub process the variable to undergo mutation was selected randomly and was replaced with a random value between the upper and lower bound of that variable.

- Crossover and mutation were performed only if their probability of execution was less than their respective initial pre-defined probability.

- The fitness function is problem dependent but for a sample run it was taken as:

  $$f(x,y,z)=x*x*x-(x*y)+z$$

- timestamp () function computed the execution time till specified number of generations were reached. For this it captured current time twice and subtracted them.

- The user defined function evaluation () computed the fitness value of each individual which was taken as the objective function.

During the initial run of TG_GA, we initialized count of individuals in population, maximum number of generations, probability of crossover and probability of mutation in the program. These variables in the program can be easily reinitialized to any value suited to the problem under consideration. The value of the variables used in the initial run of TG_GA are as given in the Table- II. The range of three input variables namely, x, y and z are specified in a file named *geneticdata.txt.* TG_GA opens this text file in input or read mode to get initial values of these variables.

Table II: Initial experimental settings for TG_GA

PARAMETER	INITIAL VALUE USED
Population Size	60
Maximum Generations	200
Number of variables in each individual	3
Crossover Probability	0.80
Mutation Probability	0.15
Initial range of 1st variable	0 -5
Initial range of 2nd variable	0-5
Initial range of 3rd variable	(-5) to (+5)

Although the fitness function is problem and test adequacy dependent but for this trial run of TG_GA it was formulated as a cubic function mentioned below

$$f(x,y,z)=x*x*x-(x*y)+z$$

where x, y and z are the three input variables to TG_GA within their due range specified in *geneticdata.txt*

VIII. RESULTS

Execution of TG_GA

TG_GA was executed with the initial parameter settings and fitness function as mentioned in last section.

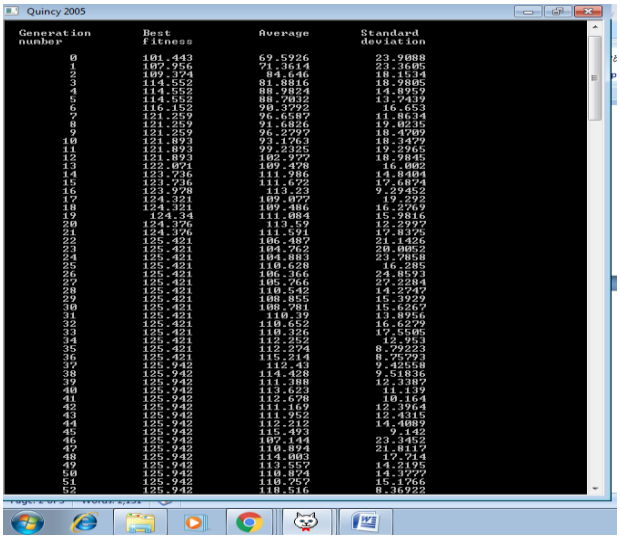

Fig. 2. Execution of TG_GA for computation of fitness function for successive generations

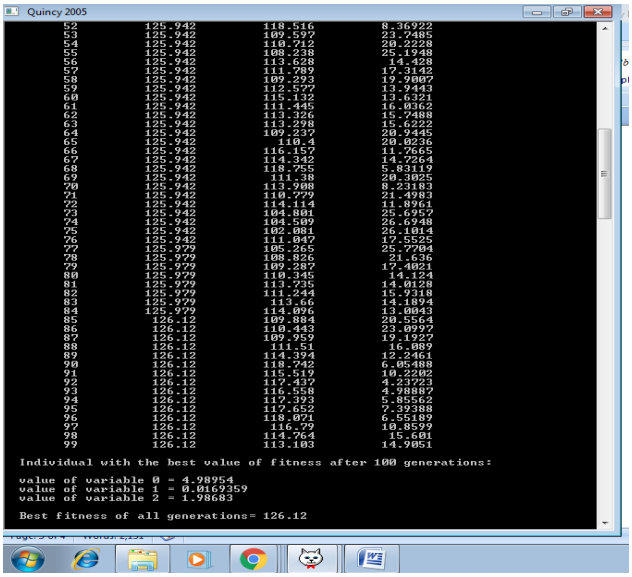

Fig. 3. Execution (continued) of TG_GA for computation of fitness function for successive generations

Fig.2 and Fig.3 gives us the value of the three input variables for the best instance of fitness function of last section as 4.98954, 0.0169359 and 1.98683.The screenshots captured also show the best fitness, average fitness and the standard deviation for first hundred generations of population, numbered from 0 to 99.These values indicate the tendency of Genetic Algorithms to get trapped in local optima, here, 125.421, 125.942 and 126.12. Since, the number of iterations, however large, in any computational procedure cannot be unbounded, thus such local optima prevents TG_GA to converge to the global optimum.

Table-III shows the best fitness values of the first thirty generations of TG_GA captured by screenshots of Fig.2 and Fig.3.

Table III: Best Fitness values of first 30 (numbered as 0 to 29 in screenshot) generations of TG_GA for fitness function

Generation Number	Best Fitness Value
1	101.443
2	107.956
3	109.374
4	114.552
5	114.552
6	114.552
7	116.152
8	121.259
9	121.259
10	121.259
11	121.893
12	121.893
13	121.893
14	122.071
15	123.736
16	123.736
17	123.978
18	124.321
19	124.321
20	124.34
21	124.376
22	124.376
23	125.421
24	125.421
25	125.421
26	125.421
27	125.421
28	125.421
29	125.421
30	125.421

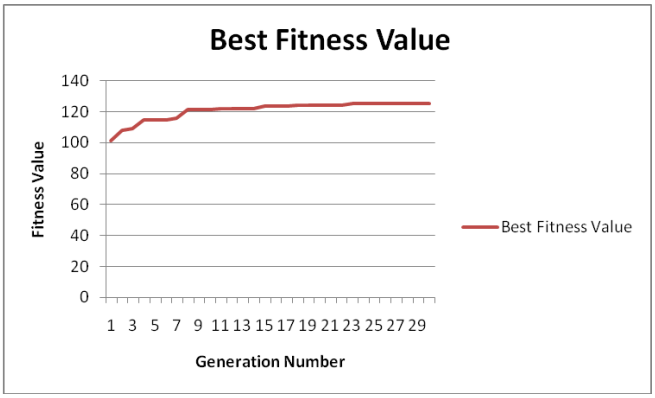

Fig. 4.Generation number vs best fitness value for first 30 generations

Fig.4 shows the variation in the best fitness value plotted against the generation number and more importantly it shows the convergence of Genetic Algorithms for the data of Table-III and also trapped in local optima.This is a critical problem of Genetic Algorithms irrespective of the fitness function used.

Impact of change in range (max-min) of input variable to the number of generations required for convergence

Table-IV shows that the number of genrations required has no dependency on the range of value of variables.

Table-IV: Variation in number of generations required for convergence with change in range of three variables for the fitness function

Range of Variables (MAX-MIN)	No. of generations before convergence
1	175
2	152
3	155
4	83
5	150
6	85
7	117
8	166

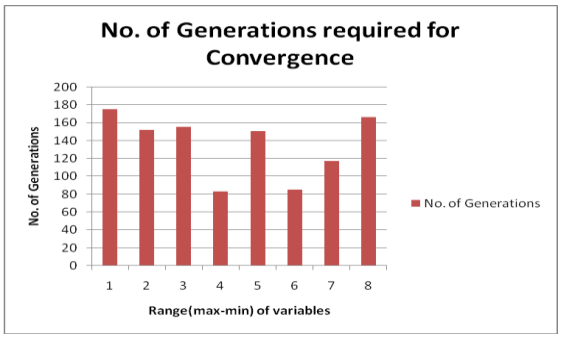

Fig. 5. Graph showing variation in number of generations required for convergence with change in range of three variables for the fitness function.

It is apparent from graph of Fig. 5 that change in the range of values of different individuals in Genetic Algorithms has no effect on the count of generations and hence the running time required to converge to a global optimum (or local optimum if trapped)

IX. CASE STUDY WORKING OF TG_GA FOR A SAMPLE PROGRAM

To determine potential effectiveness of TG_GA, a case study comprising of a GCD program as shown in Fig.6 was carried out [29]. This case study was also carried out by the first author in [29] for comparison of time taken by Genetic Algorithms against that taken by Ant Colony Optimization.

```
1.  void main ( int x, int y)  { int z;
2.  if( y > x)  {
3.  z = x;
4.  x = y;
5.  y =z; }
6.  z = x % y;
7.  while ( z !=  0) {
8.  x = y;
9.  y = z;
10. z = x  %  y;
11. }
12. return y;
13. }
```

Fig. 6. Program to find GCD of two numbers

Fig.6. shows an instrumented GCD program that accepts two integer parameters namely x and y and computes their greatest common divisor or highest common factor and outputs it as z.

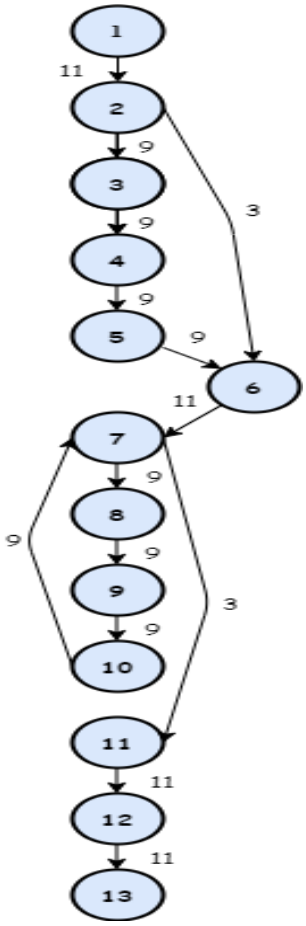

Fig. 7. Control flow graph of the GCD program

Four of the simple independent paths for the program of Fig.6 can be easily inferred from the CFG of Fig.7 as:

1. **P1**: 1-2-6-7-11-12-13
2. **P2**:1-2-6-7-8-9-10-7-11-12-13
3. **P3**:1-2-3-4-5-6-7-11-12-13
4. **P4**:1-2-3-4-5-6-7-8-9-10-7-11-12-13

Fitness function for the problem based on path dependency is given as:

$$f(x)=\sum W_i \text{ for all } i = 0 \text{ to } n,$$

where W_i denotes the weights assigned to respective paths.

TG_GA was executed for the GCD program of Fig.6 and it was also executed with randomly generated test data. The results summarized in Table V show that for the same parameter settings of input, random testing took execution time which was on average five times or more when compared to TG_GA to reach the same fitness value for which the associated value of variables can serve as competent enough test data for the purpose of error detection.

Table-V: Time taken by TG_GA and random testing for GCD program of Fig.6

Testing Number	Time taken by TG_GA (in msec)	Time taken by Random Testing (in msec)
1	2.1	11.5
2	1.06	8.8
3	3.4	13.99
4	2.86	16.07
5	1.1	8
6	3.1	19.66

IX. CONCLUSIONS AND FUTURE SCOPE

In this paper are presented the overview and possibilities of applying Genetic Algorithms to automated buildup of test data and for this a tool named TG_GA was developed. The methodology presented tries to detect a collection of test cases that escort to fulfilling a given condition or a constraint. It is represented in the form of a fitness function, for the software under test.

The tool TG_GA was applied on the program of finding GCD of two numbers (case study). Such small C programs are used as basis in test data generation approach. Using the same experimental settings, the GCD program was executed again but using randomly generated test cases this time. It was inferred that for the same parameter settings of input, random testing took execution time which was on average **five** times or more when compared to TG_GA to reach the same fitness value for which the associated values of variables can serve as competent enough test data for the purpose of error detection. In random testing, since data points do not have dependence with time, it becomes inefficient as the quantity of test data to be generated becomes large.

Also, the competence of test cases produced by Genetic Algorithms is far better than the quality of test cases produced randomly because they can direct the constructing of test data to a sensible range fast. Thus, the important merits of Genetic Algorithms have been its simplicity, speed and accuracy.

However, during this experimental study it was observed that within few numbers of generations, solutions derived by Genetic Algorithms might get trapped around local optimum because of unwanted paths and consequently, fail to detect the global optimum. In real sense, the execution time cannot be limitless, so the repetitions in the algorithm also have to be bounded.

In future, there is a possibility to compare Genetic Algorithms with other exhaustive search techniques to see if cooperation among them has the potential to eliminate the problem of Genetic Algorithms being trapped in local optimum. Exploring effect of multiple crossover points instead of a single one and a lower value of mutation probability might also bring more diversity into the population and might be able to reduce the likeliness of Genetic Algorithms being trapped in such local optima.

REFERENCES

1. Nagori, M., Kale, J.(2010). Genetic Algorithms and Evolutionary Computation. IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.12, 126-133.
2. Xie, X., Xu, B., Shi, L., Nie, C., & He, Y. (2005, December). A dynamic optimization strategy for evolutionary testing. *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific* (pp. 8-pp),121-130
3. Miller, J., Reformat, M., & Zhang, H. (2006). Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, *48*(7), 586-605.
4. Korel, B. (1996, May). Automated test data generation for programs with procedures. In *ACM SIGSOFT Software Engineering Notes*, 21(3), 209-215.
5. Kaur, A., & Goyal, S. (2011). A genetic algorithm for regression test case prioritization using code coverage. *International journal on computer science and engineering*, *3*(5), 1839-1847.
6. Umbarkar, A. J., & Sheth, P. D. (2015). Crossover Operators In Genetic Algorithms: A Review" Ictact Journal On Soft Computing. *ICTACT journal on soft computing*, *6*(1), 1083-1092.
7. Nirpal, P. B., & Kale, K. V. (2011). Using genetic algorithm for automated efficient software test case generation for path testing. *International Journal of Advanced Networking and Applications*, *2*(6), 911-915.
8. Al-Zabidi, M.S., Kumar, A., & Shaligram, A.D. (2013). Study Of Genetic Algorithm For Automatic Software Test Data Generation, Galaxy International Interdisciplinary Research Journal. 1(2), 65-74.
9. Ansari, A., Khan, A., Khan, A., & Mukadam, K. (2016). Optimized regression test using test case prioritization. *Procedia Computer Science*, *79*, 152-160.
10. Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.
11. Bertolino, A. (2007, May). Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering* (pp. 85-103). IEEE Computer Society.
12. Li, Z., Harman, M., & Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, *33*(4). 225-237.
13. Camazine, S. (2003). *Self-organization in biological systems*. Princeton University Press.
14. Cao, Y., Hu, C., & Li, L. (2009, July). An approach to generate software test data for a specific path automatically with genetic algorithm. In *Reliability, Maintainability and Safety, 2009. ICRMS 2009. 8th International Conference on* (pp. 888-892). IEEE.
15. Srivastava, P. R., & Kim, T. H. (2009). Application of genetic algorithm in software testing. *International Journal of software Engineering and its Applications*, *3*(4), 87-96.
16. Mitras, B., & Aboo, A. K. (2014). Hybrid of Genetic Algorithm and Continuous Ant Colony Optimization for Optimum Solution. International Journal of Computer Networks and Communications Security, 2(1), 1-6.
17. McMinn, P., & Holcombe, M. (2003). The state problem for evolutionary testing. In *Genetic and Evolutionary Computation—GECCO 2003* (pp. 214-214). Springer Berlin/Heidelberg.
18. Sharma, A., Rishon, P., & Aggarwal, A. (2016). Software testing using genetic algorithms. *Int. J. Comput. Sci. Eng. Surv.(IJCSES)*, *7*(2), 21-33.
19. Wegener, J., Buhr, K., & Pohlheim, H. (2002, July). Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation* (pp. 1233-1240). Morgan Kaufman.
20. You, L., & Lu, Y. (2012, May). A genetic algorithm for the time-aware regression testing reduction problem. In *Natural Computation (ICNC), 2012 Eighth International Conference on* (pp. 596-599). IEEE.
21. Alzabidi, M., Kumar, A., & Shaligram, A. D. (2009). Automatic Software structural testing by using Evolutionary Algorithms for test data generations. *International Journal of Computer Science and Network Security*, *9*(4), 390-395.

22. Rajappa, V., Biradar, A., & Panda, S. (2008, July). Efficient software test case generation using genetic algorithm based graph theory. In *Emerging Trends in Engineering and Technology, 2008. ICETET'08. First International Conference on* (pp. 298-303). IEEE.
23. Peng, X., & Lu, L. (2011, May). A new approach for session-based test case generation by GA. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on* (pp. 91-96). IEEE.
24. Girgis, M. R. (2005). Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. *J. UCS*, *11*(6), 898-915.
25. Ahmed, M. A., & Hermadi, I. (2008). GA-based multiple paths test data generator. *Computers & Operations Research*, *35*(10), 3107-3124.
26. Ghiduk, A. S., Harrold, M. J., & Girgis, M. R. (2007, December). Using genetic algorithms to aid test-data generation for data-flow coverage. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific* (pp. 41-48). IEEE.
27. N Meghna, KJyoti,"Genetic Algorithms and Evolutionary Computation" *IJCSNS International Journal of Computer Science and Network Security*, VOL.10 No.12, December 2010.
28. Xiaoyuan Xie, Baowen Xu, Liang Shi, Changhai Nie, Yanxiang He" A dynamic optimization strategy for evolutionary testing, *IEEEXplore, 2006*.
29. Sachdeva, T.(2020). "Swarm Intelligence Techniques and Genetic Algorithms for Test Case Prioritization". *IJEAT International Journal of Engineering and advanced Technology*, Volume-9 Issue-4(in press).
30. www.t-vec.com/solutions/tvec.php
31. www.red-gate.com/products/sql-development/sql-data-generator/
32. download.cnet.com/GS-DataGenerator/3000-2092_4-10303373.html
33. www.upscene.com/advanced_data_generator/editions
34. www.ibm.com/developerworks/data/library/techarticle/dm-0706salkosuo/

AUTHORS PROFILE

Tina Sachdeva is currently working as an Assistant Professor in the Department of Computer Science of Shaheed Rajguru College of Applied Sciences for Women, University of Delhi with about 12 years of teaching experience. She has authored several National and International research publications.

Astha Pattanaik is a student of B.Sc(H) Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi .