Project report

on

"**Software Test Data Generation Using Evolutionary Techniques**"

submitted under Major Project-II

in

Master of Technology in Computer Science and Engineering

(M.Tech CSE)

By

Tina Arora

2K14/CSE/503

Under the Guidance of:

Dr. Ruchika Malhotra
(Assistant Professor, Department of Computer Science and Engineering)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

July 2017

CERTIFICATE

This is to certify that Project Report entitled "**Software Test Data Generation Using Evolutionary Techniques**" submitted by Tina Arora **(Roll No: 2K14/CSE/503)** for partial fulfilment of the requirement for the award of Degree in Master of Technology (Computer Science and Engineering) is a record of the candidate work carried out by her under my supervision.

Dr. Ruchika Malhotra

Project Guide
Assistant Professor
Department of Computer Science & Engineering
Delhi Technological University

DECLARATION

I hereby declare that the Major Project-II work entitled "**Software Test Data Generation Using Evolutionary Techniques**" which is being submitted to Delhi Technological University, in partial fulfilment of requirements for the award of degree of Master of Technology (Computer Science and Engineering) is a bonafide report of Major Project-II carried out by me. The material contained in the report has not been submitted to any university or institution for the award of any degree.

Tina Arora

2K14/CSE/503

ACKNOWLEDGEMENT

I owe my profound gratitude to my project guide, Assistant Professor, Dr. Ruchika Malhotra who has been a constant source of inspiration to me throughout the period of this project. It was her competent guidance, constant encouragement and critical evaluation that helped me to develop a new insight into my project. Her calm, collected and professionally impeccable style of handling situations not only steered me through every problem, but also helped me to grow as a matured person.

I am also thankful to her for trusting my capabilities to develop this project under her guidance.

Date : Tina Arora

TABLE OF CONTENTS

List of Figures

List of Tables

List of Abbreviations

S.No	Acronym	Full Form
1	**CFG**	Control Flow Graph
2	**du paths**	Definition Use Paths
3	**dc paths**	Definition Clear Paths
4	**PAN**	Permanent Account Number
5	**SQL**	Structured Query Language
6	**CSV**	Comma Separated Values
7	**GA**	Genetic Algorithms
8	**TG_GA**	Test Data Generator using Genetic Algorithms

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. One of the automation techniques is using Genetic Algorithms (GA). They are iterative algorithms that 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 an optimal solution. 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 Case Generation using Genetic Algorithms) that searches for test data in a discontinuous space.

1. Chapter 1 : 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, test cases play 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 competent enough to reveal all defects and bugs. 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) card numbers, email addresses, etc. This eliminates the need to look up the algorithms for generating the 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 export it to a desired format.

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. Thus, the population moves towards an optimized result over consecutive generations.

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]. 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 our paper we develop a tool, TG_GA, that looks for optimal solution in a discontinuous search space and collect 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. During the development process, we make a file *geneticdata.txt* and specify the range of the 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.

1.1 Motivation Of The Work

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 the recent years they have grown in popularity in optimization of engineering problems [19].

1.2 Objective of The Project

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.

1.3 Organization of the Thesis

The Thesis is divided into chapters and each chapter is organized as follows:

Chapter 2 gives a summary of Literature Review done and works done concisely in each article**.** This Literature Survey was done before the implementation of this Project was started.

Chapter 3 describes basic concepts related to Genetic Algorithms that are used in the rest of the chapters. Concepts like Genetic Operators (Selection, Crossover and Mutation); Fitness Function; Flowchart and Pseudocode for a Genetic Algorithm are given.

Chapter 4 discusses Test Data Generation, advantages associated with its automation and Test Adequacy Criteria.

Chapter 5 talks of some popular Test Data Generation Tools available. It presents a concise summary in a tabular form with respect to basic functionality, platform or languages supported, etc.

Chapter 6 discusses Control Flow Graphs (CFG), various path testing terminologies and TG_GA the tool developed.

Chapter 7 presents the working of TG_GA, its initial experimental settings and various tables and graphs analyzing data obtained from its working.

Chapter 8 presents a Case Study of Greatest Common Divisor (GCD) Problem, Fitness Function associated with it and the Control Flow Diagram for it.

Chapter 9 gives the conclusions from the working of TG_GA. It shows the comparison between generation of test cases by TG_GA as compared to generation done using random values.
Finally, we give the list of references.

2. Chapter 2: LITERATURE SURVEY

Over the years, many researchers and scientists have carried out extensive and in-depth study in the domain of software test data generating using evolutionary algorithms. Sharma et al [18] have suggested methods in 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 GAs 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 GA to search the appropriate solutions. Srivastava and Tai-hoon, applied the Genetic Algorithm technique to find the most critical paths discontinuous spaces [15].

3. Chapter 3: EVOLUTIONARY ALGORITHMS

3.1 What are Evolutionary Algorithms?

Evolutionary Algorithms are based on a heuristic search strategy and compute an optimal solution 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 the 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 the continuation of the fittest. Each organism has a set of features, encoded and represented in the form of bits for purpose of computer programming.

3.2 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.

3.2.1 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 oncoming generation.
- The suitability of an individual is determined by its fitness.
- Fitness is computed using an objective or fitness function
- Popular selection techniques are Roulette Wheel Selection, Tournament Selection, Rank selection

- Right selection of initial population is vital for the convergence rate because better parents drive individuals to better and faster optimal values.

3.2.2 Crossover

It is the interchanging an allele of an individual with another from a different individual. 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)

Two point crossover involves selecting two random bits in the selected individuals and then swapping bits between these two genes (bits)

- It is the important differentiating factor of Genetic Algorithms from other search techniques.
- It operates at the individual level.
- Ant two candidates are selected from the existing population
- A crossover bit among all the bits is chosen
- The instances of the two sub-strings are swapped till the selected bit
- If X=000111 and Y=111000 and the crossover point is 3 then X'=111111 and Y=00000
- The novels 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.

3.2.3 Mutation

Allele of genes is randomly replaced by another to produce a new individual. The primary aim of mutating is to maintain introduce variety in the population and avoid early untimely convergence.

- Some of the bits of the individuals get complemented.
- Its goal is to sustain variety in the population and thus make untimely convergence to a local solution
- It spans through the entire search space randomly
- It inducts individuals into the population that may not be originally present
- Low mutation probability leads to insufficient global sampling and prevents convergence to a local optimum

3.3 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 the objective function as compared to one who is farther.

3.4 Flowchart

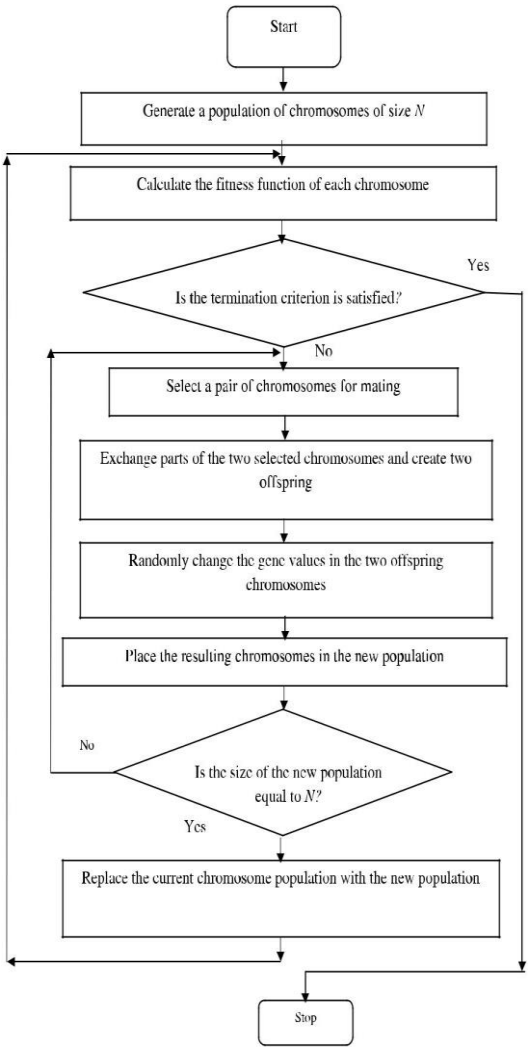


Fig 1: Flowchart for a Genetic Algorithm

Figure 1 shows the basic steps involved in a Genetic Algorithm in the form of a flowchart. It consists of the following steps:

1. Randomly generate an initial population.

2. Then compute the fitness value of all candidates in the population using Fitness Function.

3. Evaluate the individuals using their respective fitness values.

4. Check the condition satisfying the stopping criteria.

    i. Select the best solutions from the population.

    ii. Apply crossover to the solutions at random points.

    iii. Apply mutation to the new solution according to mutation probability.

    iv. Goto step 2.

5. Return the current generation of solutions.

3.5 Pseudocode of Genetic Algorithms

```
// Initialize generation
  x = 0
GP =population of randomly-generated organisms;
Calculate fitness value for each individual 'x'
belonging to the generation 'GP_x'
do {
      create_Nextgeneration();
      //Selection
      best_pop=elit_rate*p_size
      send the best_pop to the next generation
      GP_x+1 rem _pop=p_size- best_pop;
      //Crossover
         for( x=1;x<=rem_pop/2;x++)
                  randomly select 2 organisms
                  Q1 and Q2 from the remaining population
                  crossover(Q1,Q2)
         end for
    //Mutation
    for(y=1;y<=no_crossover;y++)
     select an individual from the crossover population
     and now mutate its each bit using μ
    end for
    best_soln=eval_pop();//returns the best solution
    by comparing the fitness values of each individual
 }while fitness of fittest individual in
GPy is not up to the mark
```

Fig 2: Pseudocode for a typical Genetic Algorithm

Pseudocode of a Genetic Algorithm can be given as in Fig 2 and the variables have the following meanings:

i. **p_size** :Population size
ii. **elit_rate**: Percentage of individuals passed on to next generation
iii. **best_pop:** Number of individuals passed on to next generation
iv. **rem_pop:** Number of remaining individuals in the current population after fit individuals passed onto next generation.

4. Chapter 4: 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.

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. 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.

During dynamic test data generation the software under test is actually executed are called dynamic test data generation. 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.

APPROACHES TO TEST DATA GENERATION

Random Testing: In this the test cases are created arbitrarily and executes the software using this data. It is the most simplest of all approaches but does not ensures maximum error detection as the 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. It has low chances of finding out semantically small bugs.

SYMBOLIC EXECUTION

Some old approaches of test data generation used neumonic 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. We look for paths and then only find out the conditions that traverse those paths.

4.1 Automation in Test Data Generation

Automated generation of test cases generation 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 a 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 constraint popularly known as test adequacy criteria. This condition is written in the form of a mathematical formula called as the fitness function.

4.2 Test Adequacy Criteria

Automated test data generators, which are not random in nature, carry out test assessment based on some constraint popularly known as test adequacy criteria. This condition is written in the form of a mathematical formula called as the fitness function.

Some popular test adequacy criteria are as:

1. **Statement Coverage**: All statements of the software under test must be executed at least one time.
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.

5. Chapter 5: TEST DATA GENERATION 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. The data can be filled in during non-working hours, where tester interaction is not required at all.

Advantages of automated test data generating tools are:

- Massive savings in time
- Generation of more accurate data
- Ensuring that the data in question is high in volume

Table 1 : Popular test data generation tools

Product Name	Vendor/Company	Paltform/Language compatibility	Description	Remarks
T-VEC Data Generator	TVEC	Java and C++	Uses branch coverage,generates variety of HTML reports	Popular for ease of use
SQL Data generator	Red-Gate	My SQL Server	Creates realistic data based on column and table names	High success rate
DTM Data Generator	GSApps	ERP,CRM and data warehouse development. SQL Server,DB2,Oracle	Automatically fills a database with test data for acceptance Testing	Focuses on version control and RDBMS repository
Advanced Data Genarator	Upscene Productions	InterBase,Firebird,MySQL	Generates realistic data into database,CSV files or SQL scripts	High compatibility
IBM DB2 Test Database generator	IBM	Only for DB2	Creates realistic test data generation tools for database application	Not compatible with DBMS other than DB2

6. Chapter 6: GENETIC ALGORITHM BASED APPROACH TO TEST DATA GENERATION

6.1 Basic Concepts and Definitions

6.1.1 CFG(Control Flow Graph

The Control Flow Graph (CFG) 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 set of vertices V and a set of directed edges E. It has a start node, end nodes, connection edges, decision nodes, junction nodes, and bounded regions.

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

6.1.2 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 tester checks whether the given input covers the expected path or not.

6.2 Implementation

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 it was replaced with a random value between the upper and lower bund of that variable.
- Crossover and mutation are performed only if their probability of execution is less than the 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 computes the run time for execution till specified number of generations by capturing current time twice and subtracting them.
- The user defined function evaluation computes the fitness value of each individual which is the same as the objective function

7. Chapter 7: EXPERIMENTING WITH TG_GA AND RESULTS

7.1 Initial Experimental Settings

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 values of the variables used in the initial run of TG_GA are as given in the Table 2. 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 the initial values of these variables.

Table 2: 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)

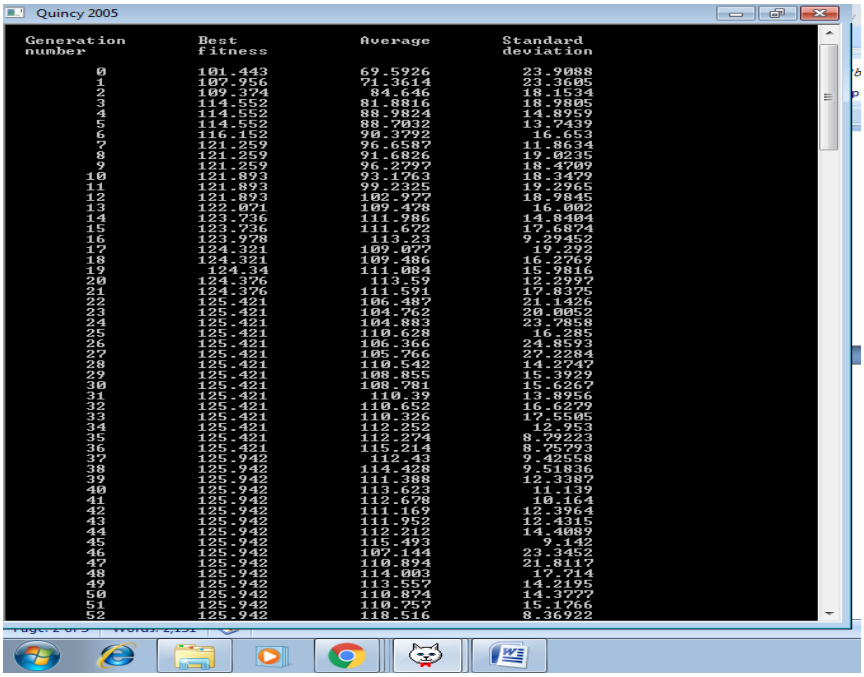
Although the fitness function is problem and test adequacy dependent and for this trial run og TG_GA it was formulated as a cubic function mentioned below

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*

7.2 Execution of TG_GA

TG_GA was executed with the initial parameter settings and fitness function as mentioned in Section 7.1.



Generation number	Best fitness	Average	Standard deviation
0	101.443	69.5926	23.9088
1	107.956	71.3614	23.3605
2	109.374	84.646	18.1534
3	114.552	81.8816	18.9805
4	114.552	88.9824	14.8959
5	114.552	88.7032	13.7439
6	116.152	90.3792	16.653
7	121.259	96.6587	11.8634
8	121.259	91.6826	19.0235
9	121.259	96.2797	18.4709
10	121.893	93.1763	18.3479
11	121.893	99.2325	19.2965
12	121.893	102.977	18.9845
13	122.071	109.478	16.002
14	123.736	111.986	14.8404
15	123.736	111.692	17.6874
16	123.978	113.23	9.29452
17	124.321	109.077	19.292
18	124.321	109.486	16.2769
19	124.34	111.084	15.9816
20	124.376	113.59	12.2997
21	124.376	111.591	17.8375
22	125.421	106.487	21.1426
23	125.421	104.762	20.0052
24	125.421	104.883	23.7858
25	125.421	110.628	16.285
26	125.421	106.366	24.8593
27	125.421	105.766	27.2284
28	125.421	110.542	14.2747
29	125.421	108.855	15.3929
30	125.421	108.781	15.6267
31	125.421	110.39	13.8956
32	125.421	110.652	16.6279
33	125.421	110.326	17.5505
34	125.421	112.252	12.953
35	125.421	112.274	8.79223
36	125.421	115.214	8.75793
37	125.942	112.43	9.42558
38	125.942	114.428	9.51836
39	125.942	111.388	12.3387
40	125.942	113.623	11.139
41	125.942	112.678	10.164
42	125.942	111.169	12.3964
43	125.942	111.952	12.4315
44	125.942	112.212	14.4089
45	125.942	115.493	9.142
46	125.942	107.144	23.3452
47	125.942	110.894	21.8117
48	125.942	114.003	17.714
49	125.942	113.557	14.2195
50	125.942	110.874	14.3777
51	125.942	110.757	15.1766
52	125.942	118.516	8.36922

Fig 3: Execution of TG_GA for fitness function as in Section 7.1

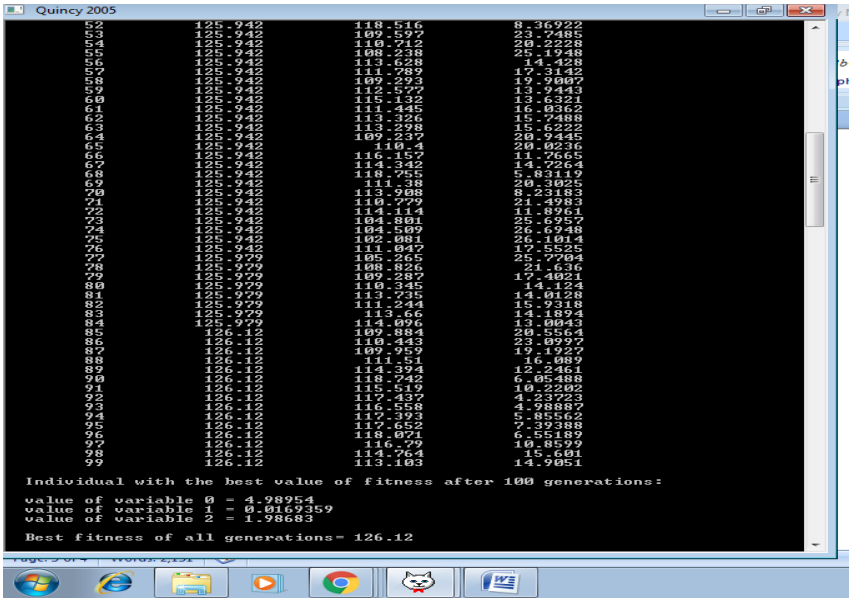


Fig 4: Execution of TG_GA for fitness function as in Section 7.1(contd.)

Fig 3 and 4 give us the values of the three input variables for the best instance of fitness function of Section 7.1 as 4.98954, 0.0169359 and 1.98683.The screenshots captured also show the best fitness, average fitness and the standard deviation for the first hundred generations of population, numbered from 0 to 100.These values indicate the tendency of Genetic Algorithms to get trapped in local optima, here, 125.42, 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 3 shows the best fitness values of the first thirty generations of TG_GA captured by screenshots of Fig.3 and Fig.4.

Table 3:Best Fitness values of first 30 generations of TG_DA for fitness function as in 7.1

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

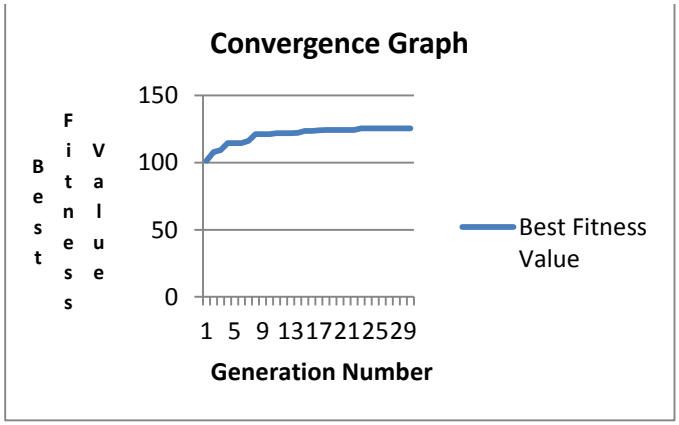


Fig 5: Generation number vs best fitness value for first 30 generations

Fig.5 shows the variation in the best fitness value computed against the generation number and more importantly it shows the convergence of GA for the data of Table 3 in local optima.This is a critical problem of GA irrespective of the fitness function used.

7.3 Impact of change in range of input variable to the number of generations required for convergence

Table 4 shows that the number of genrations required has no dependency on the range of values of variables.

Table 4: Variation in number of generations required for convergence with change in range of 3 variables for the fitness function as in (i)

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

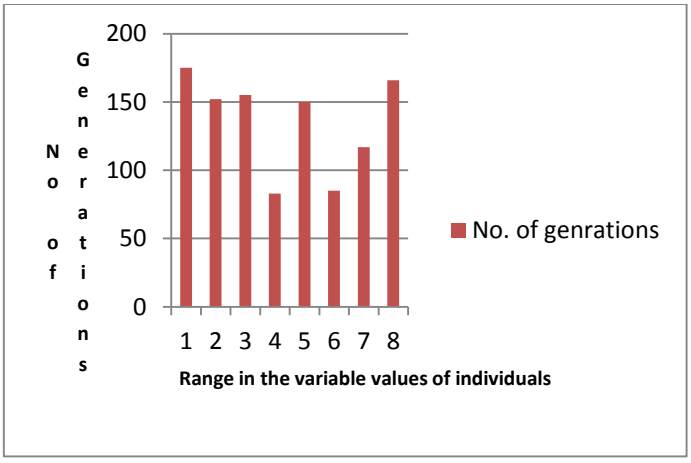


Fig.6: Graph showing variation in number of generations required for convergence with change in range of 3 variables for the fitness function as given in Section 7.1

It is apparent from graph in Fig. 6 that change in the range of values of different individuals in GA 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)

8. Chapter 8: CASE STUDY:WORKING OF TG_GA FOR A SAMPLE PROGRAM

To determine the potential effectiveness of TG_GA, a case study comprising of a GCD program as shown in Fig.7 was carried out.

8.1 GCD Program

```
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.7: Program to find GCD of two numbers

Fig.7. 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.

8.2 CFG(Control Flow Graph) of the GCD Program

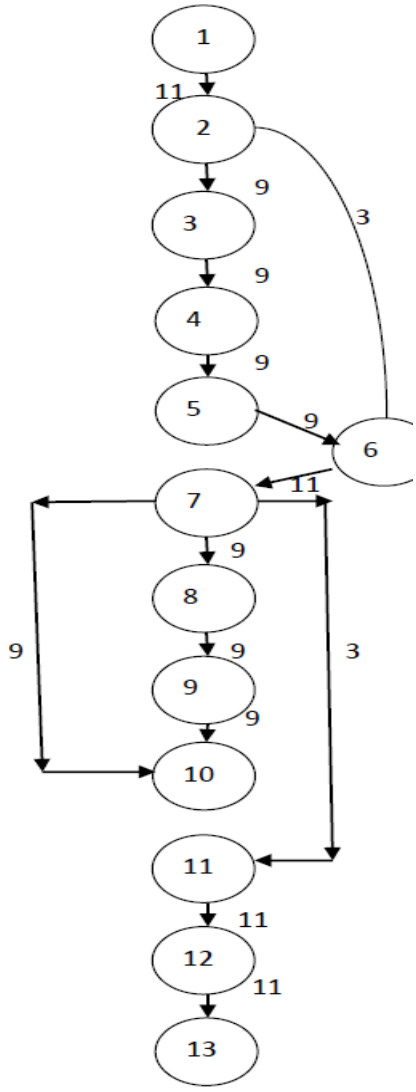


Fig 8: Control flow graph for the GCD program

The simple independent paths for the program of Fig.7 can be easily inferred from the CFG of Fig.8 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:

$f(x)=\sum W_i$ for all i =0 to n

where W_i denotes the weights assigned to the respective paths.

TG_GA was executed for the GCD program of Fig.7 and it was also executed with randomly generated test data. The results summarized in Table 4 show that for the same para\meter settings of input, random testing took execution time which was more than five times 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.

Table 5: Time taken by TG_GA and random testing for GCD program of Fig.7

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

Chapter 9: CONCLUSIONS AND FUTURE WORK

In this project, are presented the overview and possibilities of applying Genetic Algorithms to automated build up 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, 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 more than **5 (five)** times 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 GAs is far better than the quality of test cases produced randomly because they can direct the constructing of test data to the 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 GAs might get trapped around local optimum because of unwanted paths and consequently, fails to detect the global optimum. In real sense, the execution time cannot be limitless, so the repititions in the algorithm also have to be bounded.

In future, there is a possibility to compare GAs with other exhaustive search techniques to see if cooperation among them has the potential to eliminate the problem of GAs 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 GA 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.