Empirical Validation of Test Case Generation based on All-edge Coverage Criteria

Shveta Parnami
The IIS University
Jaipur, India

K.S. Sharma
The IIS University
Jaipur, India

Swati V. Chande
IIIM
Jaipur, India

ABSTRACT
Software testing assesses the functionality and correctness of the software through analysis and execution. It is done by exercising appropriate number of test cases so that no part of the program is left untested. Presence of multiple loops in a program makes it unlikely or impossible to test all paths. Therefore researchers try to find the subsets of the test cases, which when tested give confidence of complete testing. However, the subsets of paths are based on some testing criteria. In this research paper GA approach has been used to find out the subset of paths of the test program that fulfills all edge coverage criteria. The Genetic Algorithm for Test Case Generation (GATCG) proposed in this work generates reduced number of paths for a test program. These paths are termed as prime paths. The proposed GATCG technique makes use of the concept of prime paths to reduce the cost of testing. The efficiency of proposed algorithm is established from the results, in terms of number of iterations and time consumed in generating the prime paths for test programs.

Keywords
Prime paths, Test case generation, Testing cost, Genetic algorithm.

1. INTRODUCTION
Software testing is done by executing the adequate test cases with appropriate test inputs and comparing the results obtained with the expected output. It is practically impossible to manually test the larger programs. However, the automation of the testing process reduces the time and resources required to test the complex and large programs. On the basis of test design, testing is broadly categorized as: Statement testing, Path testing [1-5] and Branch testing [6-9]. Path testing is based on the basic control structure of the program and are more challenging in nature [10] in comparison to other testing methods. Most of the researches on search based software engineering have focused on branch coverage or statement coverage; very few of them consider the path coverage [11]. Path coverage allows deeper logical error(s) to be found that may not be detected if branch or statement coverage is used. It is unlikely to achieve complete path coverage for a looping program because a loop can go infinitely. Generating a sufficient amount of test paths set is a crucial task. The numbers of test paths in a no-loop program are equal to its cyclomatic complexity (CC). The presence of loops, especially nested loops, increases the number of test paths tremendously [11]. As such, it is desirable to design a mechanism through which the number of paths could be kept within definite limits.

This paper presents GA based GATCG approach to generate the subset of test paths that adequately represents the complete set of all paths of a program. The approach uses all-edge coverage criteria for the generation of test paths. For a loop based program, the number of executions for a loop is limited to, zero or one time. The generated subset of paths is termed as 'prime paths'. Prime paths set ensures that all statements of the program are covered atleast once.

The paper is organized as follows: Section 2 explains some important concepts used in the present study to generate test cases. Section 3 describes the proposed GATCG technique for test case generation. Section 4 presents the results of applying this algorithm to a sample program so as to evaluate the effectiveness of the proposed GATCG technique. Section 5 presents the conclusions drawn by the authors and future scope of the work.

2. BASIC CONCEPTS
This section explains some basic concepts used in this work to generate test cases.

2.1 Software Testing
Software testing is an important element of software quality assurance and represents the definitive analysis of specification, design and coding. It is laborious, costly and time consuming task: it spends almost 50% of software system development resources [12]. Software testing is performed for defect detection and reliability estimation. The goal of software testing is to design a set of minimal number of test cases to reveal any existing faults [12-13] and promising the complete coverage of the program under test.

Testing or inspecting for the coverage of the software is termed as coverage testing. Coverage testing is done through use of statement coverage or branch coverage or path coverage. Testing conducted to ensure that each statement is covered is called statement coverage testing. In branch testing each branch is checked during testing. Out of all, most powerful testing is path testing [11, 14-17]. In path testing each path of the test program is tested. Path testing assures complete coverage of the program. Path testing is realized through control flow graph of the test program. Identifying the test paths for testing process is a challenging task and there is need to explore these aspects of test case generation in order to increase the degree of automation and efficiency of software testing.

2.2 Control Flow Graph
A control flow graph (CFG) of the test program is build to identify paths in it. CFG is a representation, using graphical notation, of all paths that might be traversed through a program during its execution. The CFG of a program is represented by a directed graph G = (N, E), consisting of finite set of nodes (N) and a set of edges (E), where the set containing N nodes represents N statements and E is a set of edges that represents directed edges between the two nodes. A directed edge 'e' represents an ordered pair (n, m), where n

and m are adjacent nodes of CFG. A path is a sequence of adjacent nodes that starts from the starting node and ends at the exit node of CFG. A closed path passing through the start and exits node is called a cycle [18].

2.3 Path Testing

In testing, a foremost challenge is to find a good starting set of test cases that removes redundant testing, provide adequate test coverage, allows more effective testing and make limited use of the most of the limited testing resources. If the set of paths are properly chosen then some measure of test thoroughness is said to be achieved.

There are numerous paths between the start and exit of a software program. Every condition or decision statement in the program doubles the number of paths in it. Each case statement multiplies the number of paths by the number of cases it has. And also every loop duplicates the number of paths by the number of times the loop iterates[19].In the path selection, every path is exercised from start to exit, every statement, branch and case statement must be exercised at least once. All branch statements must be exercised in both directions i.e., true or false.

2.4 Prime Paths

As the number of loops and complexity of the program increases, the number of paths also increases. In that case it becomes almost impossible to test all paths of a program. The testing objectives could be achieved by testing only a subset of the paths. These subsets are made on the basis of testing criteria [15], selecting only those paths which are difficult to reach i.e. the probability of executing these paths are very low. Path subset proposed by Singh [20] has included those paths which have high probability of execution. Independent paths were considered for path subset by Faezeh et.al.[21]. The study conducted by them in the field of generating subset of the test paths manually selected these paths. However, in the existing manual selection method there are chances that the paths left during testing might have errors, which could propagate to higher levels of software development. The problem is overcome by automatically generating the subset of paths based on some coverage criteria. This subset of paths is called Prime Paths. A path from one node to other node is a prime path, if it is a simple path and does not appear as a proper subpath of any other simple path. Prime path coverage is a set of test requirements containing each prime path. The paper aims at generating a subset of paths, i.e., a set of prime paths to meet all-edge coverage criteria. All-edge coverage means that all the edges of the CFG must be exercised atleast once in the generated test cases. Studies made by them have empirically proved that the evolutionary search techniques pave the way for an effective and efficient approach for finding test cases. Next section provides a brief introduction to the Genetic algorithm technique, which is a popular and best suited method for generating test cases.

2.5 Genetic Algorithm

In many fields in the engineering worlds, Genetic Algorithms (GA) have been widely studied [1-2, 8-9, 11-15, 17, 20] and experimented. GA is based on the ideology of the evolution via natural selection, employing a population of individuals that undergo selection in presence of operators, such as mutation and recombination, which are responsible for providing variation in the population. GA is useful and work efficiently in very large and complex search space.

Simple Genetic Algorithm

A simple genetic algorithm is given below:

```
{
Population initialization;
Population evaluation;
        while (Termination Criteria Not Met)
        {
                Parents selected for reproduction;
                Perform recombination and mutation on
                selected parents;
                Re-evaluate population to pass the best
                individuals to next generation;
        }
}
```

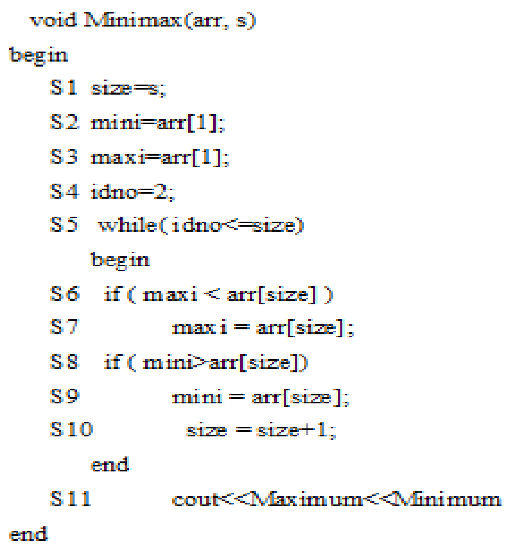
Genetic Algorithms begin with a set of initial individuals sampled from the problem domain. Individuals in each generation are evaluated with a fitness function. The algorithm performs a series of operations to transform the present generation into a new, fitter generation.

3. PROPOSED GATCG APPROACH FOR TEST CASE GENERATION

GATCG is GA based test case generation algorithm. It searches for test paths (test cases) which satisfy the all-edge coverage criterion. GATCG methodology proposes generation of test cases (in the form of test paths) by means of genetic algorithm approach. The proposed GATCG algorithm considers only test requirements that do not pose a constraint on the amount of time that a loop takes for its execution. Only two cases are relevant: whether the loop is executed or not, i.e. either the path traverse the loop zero time or traverse for one time.

The approach works by converting a program under test (PUT) to its corresponding CFG, then generating paths in the form of test cases such that every edge of the CFG are covered. GATCG works in two phases.

First phase is the set up phase where PUT is converted into CFG and then in to Optimized Control Flow Graph (OCFG). OCFG is obtained by removing unnecessary nodes from CFG, without changing the control flow semantics of the test program. The statement of a program is mapped to nodes in a control flow graph. The nodes are connected by walking through the program code and pointing a statement's node to the statement's child. The graph is then optimized by removing the unnecessary nodes in order to keep the expenses of the prime path calculation algorithm to a minimum. As explained by Gerritsen [22] it is quite costly to calculate all prime paths in a control flow graph. The more nodes a CFG has, the costlier it is. Generally, there are many nodes that will not influence the amount of prime paths; they are the nodes with only one child. Removing unnecessary nodes can optimize the prime path calculation. The optimization process must not alter the prime paths in any way, apart from the fact that they will be shortened. The set up phase also lists the set of successor for each node of OCFG. Second phase of GATCG uses GA to generate test cases to cover all edges of OCFG by using the information (list of set of successors) provided by the first phase. The end result of GATCG is the set of prime paths that include all edges of the OCFG. The approach is explained below with the help of an example. Consider the Minimax program given in Figure1. The corresponding CFG and OCFG of the program given in Figure 1 are illustrated, respectively, in Figure 2 and 3.



```
    void Minimax(arr, s)
  begin
      S1  size=s;
      S2  mini=arr[1];
      S3  maxi=arr[1];
      S4  idno=2;
      S5  while(idno<=size)
           begin
      S6   if ( maxi < arr[size] )
      S7            maxi = arr[size];
      S8   if ( mini>arr[size])
      S9            mini = arr[size];
      S10             size = size+1;
           end
      S11           cout<<Maximum<<Minimum
  end
```

Figure 1: Minimax Program

The major components of GATCG for prime path set generation are discussed below.

Search Space: It is the set of all solutions among which the result lies. It is, therefore, a set of paths in the OCFG of the program under test. A path is represented by sequence of connected nodes. Search space for the example (program in Figure 1) is the set of paths P = {P1, P2, P3, …, Pn} where Pi is a path from starting node to ending node. The search space of given example is {{0,1,2,9,10}, {0,1,2,3,5,6,7,8,9,10}, {0,1,2,3,4,5,6,7,8,9,10}…}.

Encoding: It is a process of representing individual genes (i.e nodes in the present case). GATCG uses binary vectors as chromosomes to represent nodes in the OCFG. Each gene (cell) in the binary vectored chromosome is mapped to its corresponding node in OCFG, using the map function. M_{fun} : i ϵ chromosome -> i ϵ nodes of OCFG, where i=1 represent the presence of node in the path defined by chromosome and i=0 represents absence of node from the chromosome path.

Length of the binary vector is equal to the total number of nodes in the OCFG plus three extra bits. The extra three bits represent start, loop and end node. First bit of the binary vector chromosome represents start node, last bit of binary vector chromosome represents end node. An extra bit is used to represent the node initiating loop in the program. For example the paths from the prime path set PP P1 {0, 1, 2, 9, 10} which is represented as {11100000011} in binary vector chromosome.

Initial Population: Each chromosome represents a potential prime path which is shown as binary vector. Number of chromosomes in each generation is equal to a predefined value POPSIZE. Initial population of the chromosome has the bit value 1 for start, first and end nodes. For example the initial population of chromosome C for program in Figure 1 is

Nodes	0	1	2	3	4	5	6	7	8	9	10
C =	1	1	0	0	0	0	0	0	0	0	1

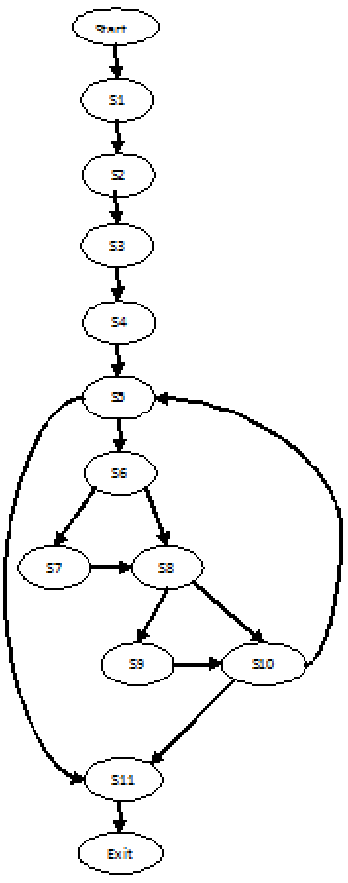


Figure 2: CFG of Minimax

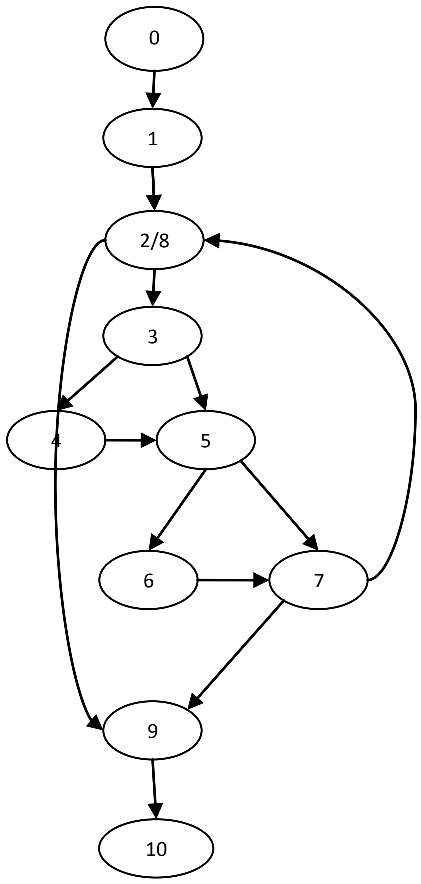


Figure 3: OCFG of Minimax

Fitness Function: The fitness function depends on the concept of adjacent nodes in OCFG. It is the probability of the adjacent nodes included in the path. The fitness function defined is as follows.

$$FFN\ (C_i) = \frac{Total\ number\ of\ adjacent\ nodes\ covered\ by\ Ci+1}{Number\ of\ total\ nodes\ present\ in\ Ci}$$

where C_i is the chromosome for i=1,2,3,… POPSIZE. The path P_i represented by chromosome C_i is an optimal solution of the problem if its fitness value is 1. For example, let chromosomes: C_1 = (11000110111) and the corresponding path is PC_1 = (0, 1, 5, 6, 8, 9, 10). In PC_1 there are 4 pairs of adjacent nodes connected with a directed edge. There are 7 nodes present in PC_1 hence, the fitness value of C_1 is (4+1)/7 i.e. 0.71.

Selection: The selection of parent chromosomes is done on the basis of their fitness values. After computing the fitness of each chromosome (using fitness function) in the current population, the algorithm uses the roulette wheel selection method [23][30] to select fit chromosomes (test paths) from the effective members of the current population that will behave as parents for the new population.

Reproduction: The algorithm uses three operators: crossover, mutation and expansion (a new operator to extend the chromosomes to represent the complete path), which are the key to the power of GAs. Crossover, mutation and expansion operators create new individuals from the selected parent chromosomes to form a new population.

(a) Crossover: Crossover operates at the individual or chromosome level with a predetermined probability CP. In one point crossover, two parent chromosomes exchange substring information at a random position in the parent chromosomes to produce two new offspring.

(b) Mutation: It operates after crossover operator and works at cell level. In mutation each cell is changed with predetermined mutation probability MP. If the cell value is 1 and the node representing that position has sibling, then the present cell value is turned to zero and its sibling's cell value is made binary one irrespective of sibling's previous value. Cell value of nodes without sibling(s) is not altered.

(c) Expansion: Expansion operates after mutation operator and works at cell level just like mutation. This operator introduces the nodes and there successor in the existing population to bring assortment in the current population. It is applied periodically in all iterations of the proposed genetic algorithm. For each chromosome in the population breeding generates a random number *pos* in the range [2,…, L-1],where L is the length of the chromosome. The node is expanded at position *pos* by altering its bit value to one and also by randomly selecting its successors and altering its bit value to 1.

Elitism: Best chromosome from the old population with a survival probability SP is retained which replaces the worst member of the current population.

Stopping Condition: GATCG algorithm stops under two circumstances; once when the generated test paths (test cases) satisfy all edge coverage condition, i.e., it covers all the edges of the OCFG and second when the number of iterations reaches the maximum number of generations.

4. EXPERIMENTAL RESULTS AND DISCUSSION

To analyze the proposed GATCG algorithm's performance, the study is conducted using a combination of iterative generation number and time required for generating test cases. The iterative generation number illustrates the convergence of the algorithm and the time consumed during the generation process, describes the generation efficiency. The experiments were conducted on a system with configuration setup where the hardware platform is a PC with 2.4 GHz Intel Core i3 CPU and 2 GB (1.86 GB usable) physical memory. The experiments were conducted using Visual C++ 10 environment. The 5 test programs as PUT (Program Under Test) that were used in the experiments are taken from the existing literature. These 5 PUT are called Triangle Classification (TC)[2,8,24-27], Minimax (MM)[2,28], Insertion Sort (IS)[2,26], Binary Search (BNS)[2,24] and Minimax Triangle Classification (MMT)[2]. The programs under test (PUT) were selected because of representativeness of their logical characteristics.

The convergence analysis of GATCG algorithm to find the optimum solution is done by recording the number of iterations required to generate prime paths and execution time in seconds. The time is calculated from the execution of proposed algorithm till the algorithm ends. The average number of iterations and execution time in 20 runs of each program are presented in Table 1 and 2 respectively.

Table 1 Number of Iterations for each program through 20 runs

Run#	TC	MM	IS	BNS	MMT
1	5	5	5	8	20
2	5	5	9	10	16
3	6	3	6	12	13
4	6	3	6	8	15
5	8	4	6	7	18
6	6	5	7	10	14
7	6	6	4	9	24
8	5	5	7	9	21
9	7	4	6	7	15
10	5	7	8	9	16
11	14	7	5	8	20
12	5	6	6	7	15
13	6	7	7	14	15
14	3	4	6	7	16
15	6	5	7	13	20
16	4	5	5	10	12
17	5	5	5	9	34
18	6	5	6	12	17
19	12	4	5	7	18
20	5	4	6	7	16
Average	6.25	4.95	6.1	9.15	17.75

Table 2 Execution time for each program through 20 runs

Run#\P#	TC	MM	IS	BNS	MMT
1	0.0381	0.0396	0.0305	0.0630	0.4197
2	0.0285	0.0485	0.0550	0.0803	0.3131
3	0.0336	0.0218	0.0397	0.0968	0.2558
4	0.0398	0.0310	0.0388	0.0659	0.2989
5	0.0452	0.0300	0.0415	0.0713	0.3542
6	0.0305	0.0473	0.0447	0.0831	0.2583
7	0.0380	0.0486	0.0285	0.0609	0.4820
8	0.0374	0.0323	0.0363	0.0739	0.4313
9	0.0461	0.0423	0.0432	0.0696	0.2782
10	0.0174	0.0429	0.0577	0.0660	0.3256
11	0.0713	0.0505	0.0386	0.0706	0.4127
12	0.0275	0.0451	0.0618	0.0643	0.2753
13	0.0350	0.0541	0.0525	0.1216	0.2818
14	0.0191	0.0361	0.0519	0.0675	0.2994
15	0.0304	0.0373	0.0595	0.1103	0.3902
16	0.0290	0.0307	0.0369	0.0823	0.2127
17	0.0327	0.0496	0.0280	0.0631	0.7021
18	0.0411	0.0455	0.0424	0.0986	0.3474
19	0.0534	0.0387	0.0430	0.0603	0.3497
20	0.0321	0.0350	0.0433	0.0599	0.2919
Average	**0.0363**	**0.0403**	**0.0437**	**0.0765**	**0.3490**

The results show that 'MM' program converges to the solution in less than 5 iterations whereas TC, IS, BNS takes more than 5 but less than 10 iterations to converge to the solution in majority of the cases. Since MMT had more edges to cover, therefore, it took on the average 17 iterations to converge to the solution. The average time required to generate test paths for the program with lesser iteration number was also found to be less.

The proposed GATCG not only generates the optimum result but also reduces the cost of testing [29]. Table 3 demonstrates that the cost of testing by generating minimal set of prime paths for all-edge coverage criteria is reduced by proposed GATCG algorithm in comparison to the cost incurred for covering all edges of the program's CFG. The GATCG technique covers a subset of statements in the form of minimal set of prime paths that guarantees the coverage of all edges of the OCFG of the tested program. The goal of covering all edges of the program's optimized control flow graph can be reduced to covering the minimal set of prime paths only. Thus, by applying GATCG technique, the cost of testing is reduced by many folds.

Table 3 Cost Reduction percentage of all edge testing

Program #	**Minimum set of Prime Paths**	**Total Number of Edges in CFG**	**Cost Reduction % =(1- minimum set of prime paths/ Number of Edges) * 100**
1	4	14	71.43%
2	3	16	81.25%
4	3	13	76.92%
5	4	17	76.47%
8	5	25	80.00%

Table 3 shows the cost reduction percentage of all edge testing requirements. Column#2 shows the minimum set of prime paths which fulfils the all-edge coverage criterion, column#3 shows the total edges in the program's Control flow graph and column#4 gives the percentage of cost reduced by using prime paths for covering all edges.

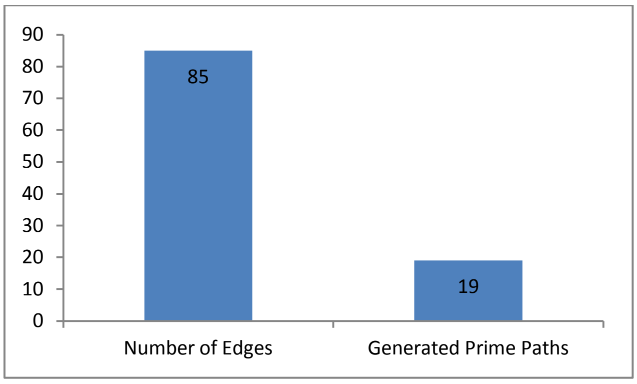


Figure 4: Total Number of Edges v/s Generated Prime Paths

The sum of edges and number of prime paths for test programs are 85 and 19 respectively. Therefore, the cost of all edge testing is reduced by 77.6%. Figure 4 shows the effectiveness of the proposed GATCG technique to reduce the cost of all-edge testing by using the reduced number of the prime paths set instead of all-edges of the OCFG of the test programs.

5. CONCLUSIONS

In software testing, the generation of testing cases is one of the key steps which have a great impact on the automation of software testing. The challenging task of testing is to find a subset of target paths that adequately represents the complete set of all paths. In this paper GATCG algorithm based on the principles of genetic algorithm has been proposed to generate test cases (a subset of all the paths of the program) that provide good coverage for all-edge testing criteria. Experiments and results show that proposed GATCG not only generates prime paths but also reduces the cost of testing. The GATCG technique based on Genetic Algorithms, proposed in the present work is an effective and efficient method of generating test cases. However, some limitations of the proposed technique have been found which are as follows: The test case generation process starts with building CFG of the program. In the present work, the CFG and OCFG of the test program are manually constructed, which consumes more time and reduces the test case generator's proficiency. The time required in the process could be reduced if a technique for generating the CFG and OCFG through an automatic process is developed. The fitness function used in the present work evaluates the chromosomes and promotes the chromosomes with high percentage of connected paths, to pass on to the next generation. In future work, the authors proposes to include some improvements in the fitness function by including the identification and elimination of infinite paths from the subset of prime paths, so as to improve the existing testing criteria. The rundown of future work is also likely to devise a system that generates test data to exercise the prime paths.

6. REFERENCES

[1] Girgis, M. R., Ghiduk, A. S., & Abd-Elkawy, E. H. (2014). Automatic Generation of Data Flow Test Paths using a Genetic Algorithm. International Journal of Computer Applications, 89(12), 29-36.

[2] Ahmed, M. A., & Hermadi, I. (2008). GA-based multiple paths test data generator. Computers & Operations Research, 35(10), 3107-3124.

[3] Groce, A. (2009). (Quickly) testing the tester via path coverage. In Proceedings of the Seventh International Workshop on Dynamic Analysis (pp. 22-28). ACM.

[4] Sy, N. T., & Deville, Y. (2001). Automatic test data generation for programs with integer and float variables. In Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on (pp. 13-21). IEEE.

[5] Gotlieb, A., & Petit, M. (2010). A uniform random test data generator for path testing. Journal of Systems and Software, 83(12), 2618-2626.

[6] Harman, M., & McMinn, P. (2007). A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In Proceedings of the 2007 international symposium on Software testing and analysis (pp. 73-83). ACM.

[7] Gupta, N., Mathur, A. P., & Soffa, M. L. (2000). Generating test data for branch coverage. In Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on (pp. 219-227). IEEE.

[8] Pargas, R. P., Harrold, M. J., & Peck, R. R. (1999). Test-data generation using genetic algorithms. Software Testing Verification and Reliability, 9(4), 263-282.

[9] Jones, B. F., Eyres, D. E., & Sthamer, H. H. (1998). A strategy for using genetic algorithms to automate branch and fault-based testing. The Computer Journal, 41(2), 98-107.

[10] Lu, S., Zhou, P., Liu, W., Zhou, Y., & Torrellas, J. (2006). Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on (pp. 38-52). IEEE.

[11] Hermadi, I., Lokan, C., & Sarker, R. (2010). Genetic algorithm based path testing: challenges and key parameters. In Software Engineering (WCSE), 2010 Second World Congress on (Vol. 2, pp. 241-244). IEEE.

[12] Parnami, S., Sharma, K. S., & Chande, S. V. (2012). A Survey on Generation of Test Cases and Test Data Using Artificial Intelligence Techniques. International Journal of Advances in Computer Networks and its Security, 2(1), 16-18.

[13] Faezeh, S. Babamir, Esmaeil Amini, S. Mehrdad Babamir, Ali Norouzi and Berk Burak Ustundag(2010), Genetic Algorithm and Software Testing based on Independent Path Concept, International Conference on Genetic and Evolutionary Methods-GEM'10, The 2010 World Congress in Computer Science, Computer Engineering and Applied Science, Las Vegas, Nevada, USA, July 2010.

[14] Ghiduk, A. S. (2014). Automatic generation of basis test paths using variable length genetic algorithm. Information Processing Letters, 114(6), 304-316.

[15] Ahmed, M. A., & Hermadi, I. (2008). GA-based multiple paths test data generator. Computers & Operations Research, 35(10), 3107-3124.

[16] Sthamer, H., Wegener, J., & Baresel, A. (2002). Using evolutionary testing to improve efficiency and quality in software testing. In Proc. of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review.

[17] Sthamer, H. H. (1995). The automatic generation of software test data using genetic algorithms (Doctoral dissertation, University of Glamorgan).

[18] Gold, R. (2010). Control flow graphs and code coverage. International Journal of Applied Mathematics and Computer Science, 20(4), 739-749.

[19] Beizer,B. (1990), Software Testing Techniques, Second Edition, Van Nostrand Reinhold Company Limited, ISBN 0-442-20672-0.

[20] Singh, H. (2004). Automatic generation of software test cases using genetic algorithms. a thesis in Thapar University Patiala may2004.

[21] Faezeh, S. Babamir, Esmaeil Amini, S. Mehrdad Babamir, Ali Norouzi and Berk Burak Ustundag(2010), Genetic Algorithm and Software Testing based on Independent Path Concept, International Conference on Genetic and Evolutionary Methods-GEM'10, The 2010 World Congress in Computer Science, Computer Engineering and Applied Science, Las Vegas, Nevada, USA, July 2010.

[22] Gerritsen, M. (2008). Extending T2 with prime path coverage exploration (Doctoral dissertation, Master's thesis, Dept. Inf. and Comp. Sciences, Utrecht Univ., 2008. Available at: http://www. cs. uu. nl/wiki/WP/T2Framework).

[23] Goldberg,D.E. (1989), Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, Mass.

[24] Jones, B. F., Sthamer, H. H., & Eyres, D. E. (1996). Automatic structural testing using genetic algorithms. Software Engineering Journal, 11(5), 299-306.

[25] Lin, J. C., & Yeh, P. L. (2000). Using genetic algorithms for test case generation in path testing. In Test Symposium, 2000.(ATS 2000). Proceedings of the Ninth Asian (pp. 241-246). IEEE.

[26] Alba, E., & Chicano, F. (2008). Observations in using parallel and sequential evolutionary algorithms for automatic software testing. Computers & Operations Research, 35(10), 3161-3183.

[27] Sagarna, R., & Yao, X. (2008). Handling constraints for search based software test data generation. In Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on (pp. 232-240). IEEE.

[28] Pei, M., Goodman, E. D., Gao, Z., & Zhong, K. (1994). Automated software test data generation using a genetic algorithm. Michigan State University, Tech. Rep, (1), 1-15.

[29] Ghiduk, A. S., & Girgis, M. R. (2010). Using genetic algorithms and dominance concepts for generating reduced test data. Informatica, 34(3).

[30] Kumar, R. (2012). Blending roulette wheel selection & rank selection in genetic algorithms. International Journal of Machine Learning and Computing, 2(4), 365.