**An improved method of genetic algorithm for path testing**

Dinh Ngoc Thi, Vo Dinh Hieu, Nguyen Viet Ha

*VNU University of Engineering and Technology, Hanoi, Vietnam*

***Abstract***

Software testing is a process to identify the quality and reliability of software, which can be achieved through the help of proper test data. However, doing this manually is a difficult task due to the presence of　huge number of predicate nodes in themodule. So, this leads towards a problem of NP-complete. Therefore, some heuristic search algorithms have to be used to generate test data. In this paper, a soft computing based approach for generating test data automatically using genetic algorithm based upon the set of given target paths is proposed. This paper cũng đề xuất một phương pháp cải tiến genetic algorithm bằng cách sử dụng static analysis để tìm ra các path có path condition có xác suất sinh data phủ được là thấp, và dựa trên các path condition này điều chỉnh thủ tục sinh quần thể mới. Finally, the proposed approach is applied some test functions. Experimental results show that improved genetic algorithm sinh ra được suitable test data có path coverage cao hơn traditional GA.

**1. Introduction**

Testing is a critical but expensive part of the software development life cycle. There is considerable interest in ways to automate testing, to reduce the cost and to gain more confidence in the result. A major task in software testing is test data generation. Heuristic search-based test data generation aims to automate this task, by searching for input test cases that satisfy chosen testing criteria.

Most research in this area considers “white box” testing, or structural coverage, in which the aim is to ensure that executing a collection of test cases results in all parts of a program being tested. This can be interpreted in various ways, including “statement coverage” (when the program is tested with all of the testcases, somewhere along the line every statement in the programis executed at least once), ‘‘branch coverage’’ (both outcomes at every logical branch in the program are executed at least once), and “path coverage” (every distinct path through the code is executed at least once). Path coverage is the strongest form of structural coverage. This paper considers path coverage.

Many approaches have been used in path testing [1]. Evolutionary path testing, which uses an evolutionary algorithm, e.x. genetic algorithm (GA) as the search engine has been found effective [2]. In this research, GA is used as the search engine.

A challenge for any search-based approach using GA is làm sao để sinh được những input test data mà có thể phủ được những test path chứa các path condition có xác suất phủ được là thấp.

Chúng tôi đề xuất một phương pháp kết hợp static analysis với cải tiến GA để sinh ra input test data nhằm mục đích phủ được các difficult path to be covered in traditional GA.

This paper is organized as follows: Section 2 presents some theoretical background to understanding this research. Section 3 describes related work, and Section 4 describes the proposed approach in detail. Section 5 presents the evaluation. Section 6 concludes the paper.

**2. Background**

*2.1. Path testing*

The objective of path testing is to search for a collection of test cases (inputs to a program) that between them lead to the traversal of all logical paths through the program. In general, path testing process consists of two major steps: target paths generation, and test data generation. Cả hai bước này đều được đề cập đến trong đề xuất của bài báo này.

*2.1.1. Target paths generation*

Target paths generation means identifying a set of logical execution pathways through the program, that we hope should all be exercised during testing. The source code is needed to construct its logical control flow, which can be presented in a control flow graph (CFG). This graph can be automatically generated by using appropriate programming language grammar in which the program is written.

From the CFG, the different logical paths through the program need to be enumerated. A logical path is a particular flow of execution through the program, which is determined by the decisions made at each decision point between the program’s entry point and its exit point.

*2.1.2. Test data generation*

Generating test data that fulfill path coverage is the main task in path testing. It is the process of creating test data, either heuristically or randomly. In a heuristic approach, the process is guided by some rules to search for required test data; the alternative is that random test data is generated.

*2.2. Generic algorithm*

The basic concepts of genetic algorithm (GA) were developed by Holland [3]. GA are commonly applied to a variety of problems involving search andoptimisation. GA search methods are rooted in the mechanisms of evolution andnatural genetics. GA draw inspiration from the natural search and selection processesleading to the survival of the fittest individuals. GA generate a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms.

The principle behind GA is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous tothe chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction.

Each individual in the environment (represented by a chromosome) receives ameasure of its fitness in the environment. Reproduction selects individuals with highfitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given below.

Simple Genetic Algorithm ()

{

initialize population;

evaluate population;

while stopping criteria not reached

{

select solutions for next population;

perform crossover and mutation;

evaluate population;

}

}

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

**3. Related work**

The path coverage literature started with Lin and Yeh [4] in 2000. They extended Jones et al.'s work [5] from branch coverage to path coverage. The ordinary (weighted) Hamming distance was extended to handle different ordering of target paths that have the same branches. The fitness function is called SIMILARITY, which computes similar items with respect to their ordering within two different paths between actual executed path and the target path. Only one program was used to test the approach, i.e. simple triangle classifier. They reported that the approach outperformed random search. Tuy nhiên phương pháp này vẫn mất một số lần gọi đến evaluation function thì mới sinh ra được test data for the most difficult path to be covered.

Bueno et al. [6] proposed an approach that utilizes control and data flow dynamic information to achieve path coverage testing using GA. In addition, the work also tackled the detection of infeasible paths by monitoring the progress of evolutionary search. The fitness function was formulated by number of coincidence branches and the normalized branch predicate value at which the actual executed path starts to deviate from the target path. Six small test programs were used to validate the approach, with 10 repetitions each to minimize random variations. Two execution modes were used, i.e. one with initialized population and the other with a random initial population. The experiment results were promising.

In 2003, Hermadi and Ahmed [7] presented evolutionary test data generation for path testing using multiple paths. Prior to this work, almost all of the evolutionary test data generators only sought to cover a single target path at a time. The fitness function used the number of matching branches and branch predicate values using Korel's fitness function [12]. It also considered path traversal techniques, neighbourhood influence, weighting, and normalization. Three small programs were used to validate the approach: minimum-maximum finder, triangle classifier, and a combination of both of them. Results were more effective and efficient by tackling multiple paths at a time.

In 2008, Ahmed and Hermadi [9] extended Hermadi and Ahmed's work of 2003 [7]. The extensions were adding a rewarding scheme and using a more efficient test data generator. A total of 32 fitness function variations were tested empirically and analysed to determine which was the best. There were 7 test programs used in the experiments. The results demonstrated that the approach was better compared to other existing work.

In the same year, Chen and Zhong [10] developed a multi-population genetic algorithm for path testing. This work has been improving GA-based path testing as described in Section 2.2. The work reported that the proposed approach outperformed a simple genetic algorithm based approach, using the triangle classifier as the test function. Cũng tương tự như cách tiếp cận của chúng tôi, paper này cũng hướng đến việc tìm kiếm được test data để phủ được path khó phủ nhất của test function. Do cách tiếp cận theo xử lý song song nên thời gian sinh test data thì tốt hơn traditional GA, nhưng số lần sinh test data vẫn nhiều (requires 21,073 test data generation by average).

In [11], Srivastava P.R and Kim T have presented a method for optimizing software testing efficiency by identifying the most critical path clusters in a program. The software under test is converted into a CFG. Weights are assigned to the edges of the CFG by applying 80-20 rule. 80 percentage of weight of incoming credit is given to loops and branches and the remaining 20 percentage of incoming credit is given to the edges in sequential path. The summation of weights along the edges comprising a path determines criticality of path. Higher the summation more critical is path and therefore must be tested before other paths. In this way by identifying most critical paths that must be tested first, testing efficiency is increased.

**4. Proposed approach**

Consider the function tA2008 (simple triangle classifier) [4, 8] which determines whether three given numbers that represent three lengths on a plane form a scalene, isosceles, equilateral, or not a triangle.

int Tritype(int a, int b, int c)

{

inttrityp = 0;

if ((a + b > c) && (b + c > a) && (c + a > b))

{

if ((a != b) && (b != c) && (c != a))

trityp = 1; // Scalene

else

if (((a == b) && (b != c)) || ((b == c) && (c != a)) ||

((c == a) && (a != b)))

trityp = 2; // Isosceles

else

trityp = 3; // Equilateral

}

else

trityp = -1; // Not a triangle

return trityp;

}

This is the CFG for the above code.



From above CFG we have target paths which have 4 paths as below:

{[1,F]} // path 1 (Not Triangle)

{[1,T], [2,F], [3,F]} // path 2 (Equilateral)

{[1,T], [2,F], [3,T]} // path 3 (Isosceles)

{[1,T], [2,T]} // path 4 (Scalence)

Giả sử rằng các input parameter a, b, c lấy giá trị nguyên dương trong khoảng a, b, c [1, 1000]. Mỗi tham số có thể lấy 1000 giá trị, cho nên không gian của input parameter sẽ là 10003. Có 1000 tam giác có các cạnh bằng nhau (1, 1, 1), (2, 2, 2)…(1000, 1000, 1000). Như vậy, xác suất để tạo ra một test data để có thể phủ được path 2 (Equilateral) là 1000/10003 = 0.000001.

Cũng bằng cách tính tương tự, ta có bảng xác suất sinh data phủ được các path như bảng sau:

|  |  |
| --- | --- |
| Classification | Probability |
| Path 4 (Scalence) | 0.4985015 |
| Path 3 (Isosceles) | 0.001998 |
| Path 2 (Equilateral) | 0.000001 |
| Path 1 (Not Triangle) | 0.4994995 |

Từ thực nghiệm nhận thấy rằng nếu chỉ dùng traditional GA để sinh ra test data cho function tA2008 thì không thể tạo ra được test data for the path 2 (Equilateral) which is the most difficult path to be covered.

GA started....

Path 4: a = 737 b = 241 c = 615

Objective call: 1

Path 1: a = 389 b = 69 c = 133

Objective call: 3

Path 3: a = 764 b = 764 c = 819

Objective call: 81

Objective call: 10000

GA completed./.

Để có thể sinh ra được test data phủ được các path có xác suất được phủ thấp, chúng tôi đưa ra cách tiếp cận gồm có 2 bước theo sơ đồ sau:



Step 1: Tìm các path mà có xác suất phủ được là thấp trong chương trình. Determine the path conditions of difficult paths to be covered.

|  |  |  |
| --- | --- | --- |
| Classification | Probability | Path condition |
| Path 4 (Scalence) | 0.4985015 |  |
| Path 3 (Isosceles) | 0.001998 | a == b || b == c || c == a |
| Path2 (Equilateral) | 0.000001 | a == b == c |
| Path 1 (Not Triangle) | 0.4994995 |  |

Step 2: Execute GA to automatically generate test cases.

***Representation***

Depend on the type of input parameters of test function, GA uses a real or integer vector as a chromosome x = (x1, x2,…, xn) to represent values of the input variables. The length of the vector depends on the required precision and the domain length for each input variable. Ngoài ra cần phải biểu diễn tập target paths là toàn bộ các path được sinh ra từ CFG.

Với function tA2008 ở trên, do input parameter là một bộ ba số a, b, c cho nên mỗi test case sinh ra được biểu diễn bằng một chromosome là một vector 3 chiều x = (x1, x2, x3). Còn tập target paths ban đầu sẽ chứa 4 path của test function.

***Initial population***

Bau đầu cần phải xác định được một số cố định popsize là số lượng chromosome trong một population. Sau đó khởi tạo giá trị ngẫu nhiên cho toàn bộ chromosome trong population ban đầu này.

***Evaluation function***

The algorithm evaluates each test case by executing the program with it as input, and recording the executed path in the program that are covered by this test case. The fitness value *eval(xi)* for each chromosome *xi* (*i* = 1,…, *popsize*) is calculated as follows:

if (executed path tập target paths)

{

eval(xi) = 0;

remove the executed path from the target paths;

}

else

{

eval(xi) = *;*

}

The distance between 2 different paths of given test function is defined as below:

distance(path i, path j) =

Ví dụ, với các path của function Tritype, thì ta có khoảng cách giữa các path như sau:

distance(path 1, path 2) = 3

distance(path 2, path 3) = 0.3333

***Selection***

After computing the fitness of each test case in the current population, the algorithm selects test cases from the effective members of the current population that will be parents of the new population. If none of the members of the current population was effective, all the members of current population are considered the parents of the new population. In the selection process the GA uses random selection method whose the selection of parents is made randomly, so that every effective member of the current population has an equal chance of being selected for recombination.

Assume that *l* members of the current population were effective, where *l* ≤ *popsize*. The parents are selected as follows:

Isolate the effective members and number them from 1 to *l*;

For i = 1 to popsize do

{

Generate an random integer number j from the range [0..*l*];

Select chromosome xj from the effective members;

}

***Recombination***

In the recombination phase, we use two operators, crossover and mutation, which are the key to the power of GA. These operators create new individuals from the selected parents to form a new population.

*Crossover*: It operates at the individual level. During crossover, two parents(chromosomes) exchange sub string information (genetic material) at a random position in the chromosome to produce two new strings (offspring). The objective here is to create better population over time by combining material from pairs of (fitter) members from the parent population. Crossover occurs according to acrossover probability *pc*. We proceed inthe following way:

For each chromosome in the (new) population:

• Generate a random (float) number *r* from the range [0..1];

• If *r* < *pc* then select given chromosome for crossover.

Now we mate selected chromosomes randomly: For each pair of coupled chromosomes we generate a random integer number *pos* from the range [1..*m*-1] (*m* is the number of bit in a chromosome). The number *pos* indicates the position of the crossing point. Two chromosomes (*b1*…*bposbpos+1*…*bm*) and (*c1*…*cposcpos+1*…*cm*) are replaced by a pair of their offspring (*b1*…*bposcpos+1*…*cm*) and (*c1*…*cposbpos+1*…*bm*).

*Mutation*: It is performed on a bit-by-bit basis. Mutation always operates after the crossover operator, and flips each bit with the pre-determined probability. The probability of mutation is *pm*. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa. So we proceed in the following way:

For each chromosome in the current (i.e. after crossover) population and for each bit within the chromosome:

• Generate a random (float) number *r* from the range [0..1];

• If *r* < *pm* then mutate the bit.

***Adjustment***

Sau khi thực hiện xử lý recombination, để có thể sinh được các test case can cover the difficult paths, cần phải điều chỉnh các giá trị của mỗi chromosomes trong population.

Phương pháp điều chỉnh được tiến hành như sau:

• Duyệt danh sách path conditions of the difficult paths to be covered

• Nếu có giá trị nào của chromosome gần thỏa mãn được path condition này thì điều chỉnh để cho nó thỏa mãn.

Quay trở lại function tA2008, với các path condition đã được xác định ở trên, tiến hành điều chỉnh như sau:

for i = 1 to popsize do

{

if ((abs(chromosome[i].x1 - chromosome[i].x2) <= epsilon) &&

(abs(chromosome[i].x1 - chromosome[i].x3) <= epsilon))

{

// Adjustment for path 2 (Equilateral)

chromosome[i].x2 = chromosome[i].x1;

chromosome[i].x3 = chromosome[i].x1;

}

}

Với các điều chỉnh này, thu được như sau kết quả khi chạy GA:

GA started....

Path 4: a = 737 b = 241 c = 615

Objective call: 1

Path 1: a = 389 b = 69 c = 133

Objective call: 3

Path 3: a = 764 b = 764 c = 819

Objective call: 81

Path 2: a = 900 b = 900 c = 900

Objective call: 3198

GA completed./.

**5. Experiments and discussion**

Ngoài test function tA2008 đã đề cập ở trên, bài báo này đã thực hiện thực nghiệm với các test function sau:

- ttB2002(Tritype)[8] accepts three numbers representing sides of a triangle, classifies its type, and computes its area.

- tM2004(Triangle)[8] classifies three numbers representing triangle side lengths into five type triangles: scalene, isosceles, right, iso-right, or equilateral.

- QuadEq2 finds all roots of a quadratic equation with 3 coefficients a, b and c as input parameters.

Kết quả phân tích tĩnh của từng test function này thu được tập target paths và path condition tương ứng như sau.

|  |  |  |  |
| --- | --- | --- | --- |
| Function | PathID | Target paths | Path condition |
| ttB2002 | 1 | {[1,F], [2,F], [3,F], [4,F], [5,F]} | b ==c |
| (Tritype) [8] | 2 | {[1,F], [2,F], [3,F], [4,F], [5,T]} | a == b |
|  | 3 | {[1,F], [2,F], [3,F], [4,T]} | a == b == c |
|  | 4 | {[1,F], [2,F], [3,T], [6,F], [7,F]} |  |
|  | 5 | {[1,F], [2,F], [3,T], [6,F], [7,T]} |  |
|  | 6 | {[1,F], [2,F], [3,T], [6,T]} | a\*a == b\*b + c\*c |
|  | 7 | {[1,F], [2,T]} |  |
|  | 8 | {[1,T]} |  |
| tM2004 | 1 | {[1,F], [2,F], [3,F]} |  |
| (Triangle)[8] | 2 | {[1,F], [2,F], [3,T]} | a\*a == b\*b + c\*c |
|  | 3 | {[1,F], [2,T], [3,F]} | b = c |
|  | 4 | {[1,F], [2,T], [3,T]} | (a\*a == b\*b + c\*c)&&(b == c) |
|  | 5 | {[1,T], [2,F], [3,F]} | a == b |
|  | 6 | {[1,T], [2,F], [3,T]} |  |
|  | 7 | {[1,T], [2,T], [3,F]} | a == b == c |
|  | 8 | {[1,T], [2,T], [3,T]} |  |
| tA2008[4, 8] | 1 | {[1,F]} |  |
|  | 2 | {[1,T], [2,F], [3,F]} | a == b |
|  | 3 | {[1,T], [2,F], [3,T]} | a == b == c |
|  | 4 | {[1,T], [2,T]} |  |
| QuadEq2 | 1 | {[1,F], [2,F], [3,F]} |  |
|  | 2 | {[1,F], [2,F], [3,T]} | b\*b == 4\*a\*c |
|  | 3 | {[1,F], [2,T]} |  |
|  | 4 | {[1,T], [4,F]} | a == 0 |
|  | 5 | {[1,T], [4,T]} | (a == 0)&&(b == 0) |

Sau đó thực hiện traditional GA và improved GA với các thông số chung được thiết lập như sau:

• Length of the chromosome: 3

• Selection method: random

• Two-point crossover probability (pc): 0.5

• Mutation probability (pm): 0.1

• Stopping criteria: target paths list is empty

Ngoài ra tùy thuộc vào mỗi test function, thiết lập các thông số cho cả traditional GA và improved GA như sau:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test function | Type | Range | Max gen | Popsize | Max run | Epsilon | Max aval. func. call |
| ttB2002 | integer | [1, 400] | 150 | 250 | 1 | 10 | 37,500 |
| tM2004 | double | [0, 50] | 100 | 250 | 1 | 1 | 25,000 |
| tA2008 | integer | [1, 1000] | 100 | 100 | 1 | 10 | 10,000 |
| QuadEq2 | double | [-50, 50] | 150 | 250 | 5 | 1 | 187,500 |

• Type: type of input parameters

• Range: range of input parameters

• Max gen: Maximum population generation for each time to run GA

• Popsize: Number of chromosome for each population

• Max run: Maximum runtime of GA

• Epsilon: critical value to adjust chromosome

• Max aval. func. call: The maximum number of times the evaluation call

Kết quả chạy traditional GA và improved GA được thể hiện trong bảng sau:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test function | Target paths | Traditional GA | | Improved GA | |
| Path ID | Eval. function call | Path ID | Eval. function call |
| ttB2002 | 8 | 8 | 1 | 8 | 1 |
| (Tritype) [8] |  | 7 | 4 | 7 | 4 |
|  |  | 5 | 10 | 5 | 10 |
|  |  | 4 | 16 | 4 | 16 |
|  |  | 2 | 266 | 2 | 252 |
|  |  | 1 | 2,747 | 1 | 988 |
|  |  | \* | 37,500 | 3 | 4,218 |
|  |  | \* | 37,500 | 6 | 19,322 |
| tM2004 | 8 | 1 | 1 | 1 | 1 |
| (Triangle)[8] |  | \* | 25,000 | 5 | 266 |
|  |  | \* | 25,000 | 7 | 3,328 |
|  |  | \* | 25,000 | 3 | 4,973 |
|  |  | \* | 25,000 | 2 | 6,968 |
|  |  | \* | 25,000 | 4 | 21,034 |
| tA2008[4, 8] | 4 | 3 | 1 | 3 | 1 |
|  |  | 0 | 3 | 0 | 3 |
|  |  | 2 | 81 | 2 | 81 |
|  |  | \* | 10,000 | 1 | 3,198 |
| QuadEq2 | 5 | 3 | 1 | 3 | 1 |
|  |  | 1 | 3 | 1 | 3 |
|  |  | \* | 187,500 | 5 | 222 |
|  |  | \* | 187,500 | 2 | 43,361 |
|  |  | \* | 187,500 | 4 | 75,508 |

Từ bảng kết quả này có thể thấy được là, mặc dù đã gọi đến số lần tối đa của evaluation function thì có những path mà traditional GA không thể sinh được test data để có thể phủ được, trong khi improved GA có thể sinh được test data mà có thể phủ được chúng.

So sánh với [4, 10], thì số lần gọi evaluation function để có thể sinh ra được test case phủ cho path 2 (Equilateral) là ít hơn (3,198 lần gọi so với 10,000 lần [4] hay 21,073 lần [10]), chứng tỏ rằng thuật toán mà chúng tôi đề xuất có hiệu quả hơn.

**6. Conclusion**

In software development life cycle, software testing is one of the critical phases. So generation of test data automatically is a key step which has a great influence on code coverage in software testing. In this paper, chúng tôi đã cải tiến GA để có thể generate test data automatically for feasible target paths.

Phương pháp tiếp cận của chúng tôi là kết hợp phân tích tĩnh để tìm các path conditions of difficult path to be covered in test fucntion, rồi từ các path condition này, điều chỉnh thủ tục sinh ra quần thể mới trong GA để có thể sinh ra được các test case phủ được các path này.

Kết quả thực nghiệm với các test function cho thấy GA đã cải tiến có thể sinh ra được các test data phủ được các path có path condition mà nếu dùng GA thông thường sẽ không thể sinh test data ra để phủ được. Ngoài ra so với các phương pháp đã có, thì đề xuất của chúng tôi cũng có hiệu quả hơn.

**7. References**

[1] M. Harman, S.A. Mansouri, Y. Zhang, Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications, Technical Report TR-09-03, April 2009.

[2] M. Harman, P. McMinn, A theoretical and empirical study of search-based testing: local, global, and hybrid search, IEEE Transaction Software Engineering, 36(2):226–247, 2010.

[3] J. H. Holland, Adaptation in Nature and Artificial Systems, Addison-Wesley, Reading, MA, 1975.

[4] Jin-Cherng Lin and Pu-Lin Yeh, Using genetic algorithms for test case generation in path testing, In Proceedings of the 9th Asian Test Symposium 2000 (ATS '00), 241-246, December 2000.

[5] Bryan F. Jones, Harmen-Hinrich Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms, Software Engineering, 11(5):299-306, September 1996.

[6] Paulo Marcos Siqueira Bueno and Mario Jino, Automatic test datageneration for program paths using genetic algorithms, International Journal of Software Engineering & Knowledge Engineering (IJSEKE), 12(6):691-709, 2002.

[7] Irman Hermadi and Moataz A. Ahmed, Genetic Algorithm based test data generator, In Proceedings of the 2003 Congress on Evolutionary Computation (CEC), volume 1, pages 85-91, December 2003.

[8] I. Hermadi, C. Lokan, R. Sarker, Dynamic stopping criteria for search-based test data generation for path testing, Information and Software Technology, 56 (4):395-407, April 2014.

[9] Moataz A. Ahmed and Irman Hermadi, GA-based Multiple Paths Test Data Generator, Computers & Operations Research, 35:3107-3124, October 2008.

[10] Yong Chen and Yong Zhong, Automatic path-oriented test data generation using a multi-population genetic algorithm, In Proceedings of the 4th International Conference on Natural Computation, 2008 (ICNC'08), volume 1, pages 566-570, October 2008.

[11] Srivastava P. R and Kim T, Application of Genetic Algorithm in Software Testing, International Journal of Software Engineering and Its Applications, 3(4), 87-96, 2009.

[12] Bogdan Korel, Automated software test data generation, IEEE Transactions on Software Engineering, 16(8):870-879, August 1990.