**A Technique for Generating Test Data using**

**Genetic Algorithms**

1st Author

1st author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

1st author's E-mail address

2nd Author

2nd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

2nd E-mail

3rd Author

3rd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

3rd E-mail

**ABSTRACT**

Search-based testing techniques using genetic algorithm (GA) can automatically generate test data that achieve high coverage on almost any given test program. GA casts the path coverage problem as a search problem and applies efficient algorithms to find test data that can serve as suitable test cases. GA approaches have its strengths and weaknesses: it scales well and can handle any code and test criterion, but degrades when test program has any critical path clusters. This paper presents a method for optimizing GA efficiency by integrating a constraint solver to solve path conditions which GA cannot generate test data for coverage. The proposed approach is also applied some test programs. Experimental results show that improved GA which can generate suitable test data has higher path coverage than the standard GA.

**Categories and Subject Descriptors**

D.2.5 [**Software Engineering**]: Testing and Debugging – Testing Tools

**General Terms**

Algorithms, Reliability, Experimentation

**Keywords**

Generic algorithm, path coverage testing, automatic test data generation

# INTRODUCTION

Software quality becomes more important than ever and software testing is the most significant measure for it. However, software testing is very laborious and costly due to the fact that it is mostly made by manual [1]. In general, software testing accounts for approximately 50 percent of the elapsed time and more than 50 percent of the total cost in software development [2]. Thus, automated software testing is a promising way to cut down time and cost.

Automatic structural test data generation is a crucial problem in software testing automation and its implementation cannot only significantly improve the effectiveness and efficiency but also reduce the high cost of software testing. We focus on path coverage test data generation in respect that various structural test data generation problem can be transformed into a path coverage test data generation problem. Furthermore, path coverage testing strategy can detect almost 65 percent of errors in test program [3].

Although path coverage test data generation is an undecidable problem [4], researchers still attempt to develop various methods and have made some progress. These methods can be classified into two types: static methods and dynamic methods.

Static methods include symbolic execution [5] and domain reduction [6, 7] etc. These methods suffer from a number of problems when it handles indefinite loops, array, procedure calls and pointer references [8].

Dynamic methods include random testing, local search approach [9], goal-oriented approach [10], chaining approach [11] and evolutionary approach [8, 12-14]. Since values of input variables are determined when programs execute, dynamic test data generation can avoid those problems with that static methods are confronted.

As a robust search method in complex spaces, genetic algorithm (GA) was applied to test data generation in 1992 [12] and evolutionary approach has been a burgeoning interest since then. Related works [8, 15, 16] indicate that GA-based test data generation outperforms other dynamic approaches e.g. random testing and local search.

As far as we know, even though GA-based test data generation already proved its efficiency in generating test data for dynamic approaches, it still has to face difficulties when the test program having test paths with low probability in generating coverable test data. For example, consider test program example1() as below [27]:

1 void example1(double x, double y, double z) {

2 if (Math.cos(z)- 0.95 < Math.exp(z)) {

3 if ((x + y == 1024) && (y > 1000))

4 // path 1

5 }

6 else

7 // path 2

8 }

By using constraint solvers, symbolic execution can generate test data for the second condition but not for the first. GA can solve the first condition. However, it has problems with the second condition. This mean that if only either symbolic execution or GA, we cannot have test data for the path 1.

This paper gives the proposal to improve GA in generating test data which can cover all the paths in the above test program. It combines constraint solvers into GA. The static program analysis step is applied to find out paths of the test program which are difficult to be covered. In this paper, the difficult path means the path contains if-else statements which is difficult to generate test data for coverage. For these difficult paths, the constraint solver Z3 [29] is used to generate the mutated individual. After that, mutated individual is used in the procedure of generating new populations in GA.

This paper is organized as follows: Section 2 gives some theoretical background to understanding this research. Section 3 summarizes some related works, and Section 4 presents the proposed approach in detail. Section 5 shows the experimental results and discussion. Section 6 concludes the paper

# BACKGROUND

This section describes to the some content is the theoretical background for the proposed approach of this paper.

## Path coverage test data generation as an optimization problem

To make use of genetic algorithm, a path coverage test data generation problem requires being transformed into an optimization problem.

Firstly, test program should be represented by its control flow graph (CFG). A CFG is a directed graph which can be denoted as G = (N, A, s, e) where N is a set of nodes, A is a set of edges; s and e are unique entry and unique exit node respectively. Each decision node is associated with a branch predicate, which is a logical expression. The edges leaving decision nodes are labeled with true or false values for corresponding branch predicate. To cause a path to be covered during execution, it is necessary to find appropriate values for the input variables that satisfy related branch predicates. A simple way is Korel’s branch distance function [9] based approach. For example, if a branch predicate C is (a == b), then the branch distance function f(C) = abs(a - b). So, to achieve a desired branch is transformed to search input vector that minimize its branch distance function. Table 1 gives some common used branch distance functions. To achieve a desired path P, we can define F(P) as the sum of all related branch distance functions. Consequently, generating path coverage test data can be transformed into searching input vector that can minimize F(P).

Table 1. Korel’s branch distance function

|  |  |  |
| --- | --- | --- |
| **No** | **Branch** | **Branch distance function** |
| 1 | a = b | f(C) = abs(a - b) |
| 2 | a ≠ b | f(C) = k |
| 3 | a < b | f(C) = (a - b) + k |
| 4 | a ≤ b | f(C) = (a - b) |
| 5 | a > b | f(C) = (b - a) + k |
| 6 | a ≥ b | f(C) = (b - a) |
| 7 | C1 ∧ C2 | f(C) = min(f(C1), f(C2)) |
| 8 | C1 ∨ C2 | f(C) = f(C1) + f(C2) |

In Korel function, k is the smallest step between 2 operands in the condition. In this paper, because all operand types are double, so to simplify we assume k = 0.

## Generic algorithm

The basic concepts of genetic algorithm (GA) were developed by Holland [17]. GA is commonly applied to a variety of problems involving search and optimization. GA search methods are rooted in the mechanisms of evolution and natural genetics. GA draw inspiration from the natural search and selection processes leading to the survival of the fittest individuals. GA generates 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 to the 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, crossover and mutation [28].

Each individual in the environment (represented by a chromosome) receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness 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 traditional GA is given below.

1 Genetic Algorithm() {

2 initialize population;

3 evaluate population;

4 while (stopping criteria not reached){

5 select solutions for next population;

6 perform crossover and mutation;

7 evaluate population;

8 }

9 }

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).

## Conditional statements in Java

Java, like all other programming languages, is equipped with specific statements that allow us to check a condition and execute certain parts of code depending on whether the condition is true or false. Such statements are called conditional, and are a form of composite statement [26].

In Java, there are two forms of conditional statements:

* the if-else statement, to choose between two alternatives
* the switch statement, to choose between multiple alternatives

This paper will only focus to the if-else statement.

### The if-else statement

The if-else statement allows us to select between two alternatives.

Syntax:

if (condition)

then-statement

else

else-statement

### Condition in an if-else statement

The condition in an if-else statement can be an arbitrary expression of type Boolean. There are 4 types of if-else statement as below.

1. *a variable of type boolean*

Example:

boolean finished;

// ...

if (finished)

// ...

1. *one of the comparison operators (==, !=, >, <, >=, or <=) applied to variables (or expressions) of a primitive type*

Example:

int a, b, c;

// ...

if (a == b + c)

// ...

1. *a call to a predicate (i.e., a method that returns a value of type boolean)*

Example:

String answer;

// ...

if (answer.equalsIgnoreCase("YES"))

// ...

1. *a complex boolean expression, obtained by applying the boolean operators !, &&, and || to simpler expressions*

Example:

int a, b, c, d;

double e, f;

// ...

if ((a > (b+c)) || (a == d) && !(Math.abs(e-f) > 10))

In this paper we focus on 2 types of if-else statement, they are 2) (one of the comparison operators) and 4) (a complex boolean expression).

# RELATED WORK

The path coverage literature using GA started with Lin and Yeh [18] in 2000. They extended Jones et al.'s work [19] 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. However, in this method, test data generation must be called many many times in order to generate the test data for the most difficult path to be covered. In addition, because their work only used GA so the test program example1() in the section 1 cannot be covered all test paths.

Bueno et al. [20] 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 [21] 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 [9]. It also considered path traversal techniques, neighborhood 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 [23] extended their work of 2003 [21]. 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 analyzed to determine which the best was. 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 [24] 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 traditional genetic algorithm based approach, using the triangle classifier as the test program. Similar to our approach, this paper also targets finding the test data to cover path conditions of the most difficult path to be covered in test program. As it approached the parallel processing, test data generating time is better than traditional GA, however the number of test data generation is still high (requires 21073 test data generation count by average).

In [25], 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.

# PROPOSED APPROACH

This section describes details of our proposed approach for automatic test data generation using improved GA. In order to generate test data which can cover the paths having the lowest coverable probability, we propose 2 step approaches as in the above flow chart:



**Fig. 1. Flow chart of our proposed approach**

## Perform static program analysis

The purpose of this step to create a list of input parameters and their setting value by performing static analysis program and using constraint solver. We called this list by mutated individual list. This list is used as conditions of adjustment procedure for GA in the next step. To create this list we have taken the following steps:

### Solve path conditions

In this paper we analyzed two types of if-else statement Java language as discussed in section 2.3.2, there are "only one comparison operator" and "complex boolean expression". We also use constraint solver Z3 to solve these path conditions.

From experiments we found that, with the equal condition in the if-else statement, without any adjustments in GA, we cannot generate test data to satisfy these condition statements. Therefore, we extract equal condition from the if-else statement, solve them by using widely known constraint solver solve Z3 and stored constraint satisfaction (will be called mutated individual) in a list to adjust in GA.

### Store equal conditions

This paper uses below class to contain each mutated individual (output of constraint solver Z3) of a test program.

class Adjust

{

public int index; // index of input parameteri

public double value; // assigned value of parameteri

}

Back to the example1() test program mentioned in section 1, the second conditions (line 2) will be analyzed and stored in the list of class Adjust as below:

adjust[0].index = 0; // input parameter x

adjust[0].value = 23; // assigned value of x

adjust[1].index = 1; // input parameter y

adjust[1].value = 1001; // assigned value of y

## Execute GA

To automatically generate test cases, using GA with below procedures:

### Representation

Depend on the type of input parameters of test program, GA uses a double or integer vector as a chromosome *chrom* = (*x*1, *x*2… *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.

### Initial population

At first, it needs to identify a fixed *popsize* number is the number of chromosome in a population (called *popsize*) also maximum population generation for each time to run GA (called *maxgen*). Then initialize random values for all chromosomes in the first population.

### Fitness function

Korel’s branch distance function is used as fitness function in improved GA. Để áp dụng được Korel’s branch distance function thì cần phải insert instrumented code vào test program và sử dụng chính test program này như là fitness function of GA. For example, with test program example1() in section 1, instrumented code will be inserted into original code at line 2 and 9 as below:

1 double example1(double x, double y, double z) {

2 double ret = (Math.cos(z) - 0.95) - Math.exp(z);

3 if (Math.cos(z)- 0.95 < Math.exp(z)) {

4 if ((x + y == 1024) && (y > 1000))

5 // path 1

6 }

7 else

8 // path 2

9 return ret;

10}

### Selection

A selection scheme is applied to determine how individuals are chosen for mating based on their fitness. Fitness can be defined as a capability of an individual to survive and reproduce in an environment. Selection generates the new population from the old one, thus starting a new generation. Each chromosome is evaluated in present generation to determine its fitness value. This fitness value is used to select the better chromosomes from the population for the next generation.

### Crossover and mutation

After selection, the crossover operation is applied to the selected chromosomes. It involves swapping of values of vector *x* = (*x*1, *x*2,…, *xn*) between two chromosomes. This process is repeated with different parent chromosomes until the next generation has enough chromosomes. After crossover, the mutation operator is applied to a randomly selected subset of the population. Mutation alters chromosomes in small ways to introduce new good traits. It is applied to bring diversity in the population.

### Adjustment

Mục đích của thủ tục điều chỉnh là giúp cho GA nhanh chóng sinh ra được test data mà có thể cover được toàn bộ test path của test program. So that after executing the mutation, based on list of mutated individual which are contained in list adjust, we need to adjust the values of each chromosome in the population. The adjustment will be executed as follows:

void Adjustment(adjust list){

for each adjust[i] in the adjust list

chrom.x[adjust[i]] = adjust[i].value

}

# EXPERIMENTAL RESULTS

Ở phần này chúng tôi sẽ trình bày về kết quả thực hiện thực nghiệm với các test program của improved GA so với traditional GA.

## Programs under test

Ngoài test program example1() đã trình bày ở section 1, để minh họa cho kết quả cải tiến GA chúng tôi thực hiện thêm trên 2 test program example2() và example3() như sau:

### Example2 test program

This function sử dụng các hàm thư viện của ngôn ngữ Java, với mục đích để xác định xem từ 3 số input x, y, z có phải là thể hiện cho 3 cạnh của một tam giác vuông cân hay ko. Dễ dàng nhận thấy rằng symbolic execution-based testing sẽ không thể áp dụng cho test program này, vì không có constraint solver nào có thể solve được các hàm thư viện của ngôn ngữ Java.

1 void example2(double x, double y, double z) {

2 if ((Math.round(x) == Math.round(Math.hypot(y, z)))

3 && (y == z)) {

4 // path1

5 else

6 // path2

7 }

8 }

Our proposed approach là thực hiện static program analysis, để xác định các condition có thể solve được bằng constraint solver. Kết quả thực hiện static program analysis cho kết quả condition (y == z) có thể được solve bằng constraint solver Z3 để thu được mutated individual là {y = 1, z = 1}. Hai mutated individual sẽ được đưa vào thủ tục điều chỉnh của improved GA.

### Example3 test program

Cũng tương tự như example2, từ 3 input là góc và 2 cạnh của một tam giác, function này sẽ sử dụng các hàm thư viện của ngôn ngữ Java xác định xem đó là một tam giác thường, cân, đều hay không phải là một tam giác.

1 void example3(double corner,double edge1,double edge2){

2 if (corner > 0 && corner < Math.PI)

3 {

5 if (edge1 == edge2) {

6 if(Math.abs(Math.toDegrees(corner) - 60) < 0.01){

7 // path1: Equilateral

8 }

9 else {

10 // path2: Isosceles

11 }

12 }

13 else {

14 // path3: Scalene

15 }

16 }

17 else {

18 // path4: Not a triangle

19 }

20}

## GA parameters setting

Parameter settings of traditional GA and improved GA are as following:

* Length of the chromosome: 3
* Selection method: random
* Two-point crossover probability (pc): 0.5
* Mutation probability (pm): 0.1
* Stopping criteria: all test target paths are covered

Each test program still requires other parameters below:

Table 2. GA parameter setting for each program

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Program** | **Type** | **Range** | **Max gen** | **Pop size** |
| example1 | double | [-10000,10000] | 150 | 250 |
| example2 | double | [0,10000] | 150 | 250 |
| example3 | double | [0,5] | 150 | 250 |

* Type: type of input variables
* Range: range of input variables
* Maxgen: maximum population generation for each time to run GA
* Popsize: number of chromosome for each population

## Results

Kết quả sinh test data của improved GA so với traditional GA được thể hiện ở trong bảng sau. Chúng ta sẽ đánh giá theo 2 tiêu chí là số lượng các test path được phủ và số lần phải thực hiện sinh test data.

### Test path coverage

Tiêu chí này được đánh giá dựa trên việc GA có khả năng sinh ra được test data mà có thể coverage được bao nhiêu test path của test program.

Table 3. Test path coverage

|  |  |  |  |
| --- | --- | --- | --- |
| **Program** | **Feasible path** | **Standard GA** | **Improved GA** |
| example1 | 2 | 1 | 2 |
| example2 | 2 | 1 | 2 |
| example3 | 4 | 2 | 4 |

From Table 4, we can see that for 2 test programs (example2() and example3()) thì improved GA có thể sinh ra được số test path coverage cao hơn so với traditional GA.

### Test data generation counts

Tiêu chí này đánh giá dựa trên số lần sinh test data để có thể phủ được toàn bộ feasible paths trong test program. Trong trường hợp không thể sinh được test data để phủ được toàn bộ feasible paths thì test data generation counts = *maxgen* x *popsize*.

Table 4. Test data generation counts

|  |  |  |  |
| --- | --- | --- | --- |
| **Program** | **Feasible path** | **GA** | **Improved GA** |
| example1 | 2 | cannot cover all paths | 252 |
| example2 | 2 | cannot cover all paths | 9658 |
| example3 | 4 | cannot cover all paths | 1098 |

From Table 4, we can see that improved GA chỉ cần sử dụng số lần sinh test data hữu hạn để sinh ra được test data phủ được toàn bộ các test path của test program, trong khi GA thì không thể làm được việc này.

# 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, we have improved the GA in order to generate test data automatically for feasible execution paths.

Our proposed approach is from a given test program, chúng tôi tìm các condition mà GA sẽ khó hoặc không thế sinh ra được test data mà có thể phủ được. Sau đó chúng tôi sẽ dùng constraint solver tool Z3 để giải các condition này. Kết quả giải ra được từ Z3 sẽ được sử dụng lại trong GA ở thủ tục sinh ra các quần thể mới.

The experimental results on these test programs shows that improved GA generated test data can cover all feasible paths having path conditions which cannot be covered by test data generated from standard GA.

Hạn chế của proposed approach in this paper là mới chỉ áp dụng được constraint solver để solve được một constraint trong test program. Trong tương lai chúng tôi sẽ mở rộng cách tiếp cận để có thể dùng được constraint solver giải được nhiều path conditions in a test program.

# REFERENCES

1. B. Antonia, "Software Testing Research: Achievements, Challenges, Dreams," in 2007 Future of Software Engineering: IEEE Computer Society, 2007.
2. G. J. Myers, The Art of Software Testing, 2nd edition: John Wiley & Sons Inc, 2004.
3. B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, Inc, New York, NY, USA, 1982.
4. E. J. Weyuker, The applicability of program schema results to programs, International Journal of Parallel Programming, vol. 8, 387-403, 1979.
5. C. K. James, A new approach to program testing, in Proceedings of the international conference on Reliable software Los Angeles, California: ACM, 1975.
6. T. Y. Chen, T. H. Tse, and Z. Zhiquan, Semiproving: an integrated method based on global symbolic evaluation and metamorphic testing, in Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis Roma, Italy: ACM, 2002.
7. S. Nguyen Tran and D. Yves, Consistency techniques for interprocedural test data generation, ACM SIGSOFT Software Engineering Notes, vol. 28, 108-117, 2003.
8. G. M. C C Michael, M Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering, vol. 27, 1085-1110, 2001.
9. B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering, vol. 16, 870-879, 1990.
10. B. Korel, Dynamic method for software test data generation, Software Testing, Verification & Reliability, vol. 2, 203-213, 1992.
11. B. Korel, Automated test data generation for programs with procedures, in Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis San Diego, California, United States: ACM, 1996.
12. S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, Application of genetic algorithms to software testing (Application des algorithmes genetiques au test des logiciels), in Proceedings of 5th International Conference on Software Engineering and its Applications Toulouse, France, 625-636, 1992.
13. J. Wegener, A. Baresel, and H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology, vol. 43, 841-854, 2001.
14. J. Wegener, B. Kerstin, and P. Hartmut, Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing, in Proceedings of the Genetic and Evolutionary Computation Conference: Morgan Kaufmann Publishers Inc., 2002.
15. S. Levin and A. Yehudai, "Evolutionary Testing: A Case Study, in Hardware and Software, Verification and Testing, 155-165, 2007.
16. W. Joachim, Andr, Baresel, and S. Harmen, Suitability of Evolutionary Algorithms for Evolutionary Testing, in Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment: IEEE Computer Society, 2002.
17. J. H. Holland, Adaptation in Nature and Artificial Systems, Addison-Wesley, Reading, MA, 1975.
18. 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.
19. Bryan F. Jones, Harmen-Hinrich Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms, Software Engineering, 11(5):299-306, September 1996.
20. 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.
21. 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.
22. 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.
23. Moataz A. Ahmed and Irman Hermadi, GA-based Multiple Paths Test Data Generator, Computers & Operations Research, 35:3107-3124, October 2008.
24. 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.
25. 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.
26. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html>
27. Jan Malburg and Gordon Fraser, Search-based testing using constraint-based mutation, Journal Software Testing, Verification & Reliability, Volume 24 Issue 6, pages 472-495, September 2014.
28. M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood, Genetic Algorithms and the Automatic Generation of Test Data, Technical Report RR/95/195 [EFoCS-19-95], University of Strathclyde, Glasgow G1 1XH, U. K, 1995.
29. <https://z3.codeplex.com/>