Test Data Generation Approach for Basis Path Coverage

Shujuan Jiang
School of Computer Science and Technology, China University of Mining and Technology Xuzhou, 221116, Jiangsu Province, China
shjjiang@cumt.edu.cn

Yanmei Zhang
School of Computer Science and Technology, China University of Mining and Technology Xuzhou, 221116, Jiangsu Province, China
ymzhang@cumt.edu.cn

Dandan Yi
School of Computer Science and Technology, China University of Mining and Technology Xuzhou, 221116, Jiangsu Province, China
ddyi@cumt.edu.cn

ABSTRACT
On the basis of determining the feasibility of paths, this paper proposes an evolutionary approach to generating test data for feasible basis path coverage. First, the structure of the program under test is expressed by a control flow graph, and the target paths are encoded into the form of hybrid-coding that efficiently combines the statement label with the outcome of a conditional statement (i.e. T or F). Then, the genetic algorithm is employed to generate test data for multiple paths coverage, and the fitness function of an input data (an individual) takes into account the degree of the execution track matching the target paths. Finally, the proposed approach is applied in several benchmark programs. The experimental results show that the proposed approach cannot only avoid redundant test but also improve the efficiency of test data generation effectively.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging –*Testing tools, Debugging aids, Tracing.*

General Terms
Algorithms

Keywords
Feasible Basis Path Coverage, Test Data Generation, Control Flow Graph, Genetic Algorithm.

1. INTRODUCTION
Structural testing is frequently used in software testing. It should follow certain test covering standards to design test data according to the logic structure of the program under test. Path coverage is one of the most important criteria that investigate the sufficiency of software testing. It requires that every path in a program should be executed at least once.

The path of a program is a sequence of statements that execute sequentially. The execution path of a program is often very complex. For example, for a program that includes loops and conditional branch statements, it tends to include a large number of paths. Therefore, it will need great resources to test all the paths. However, both test resources and test time are limited; therefore, it is difficult to test all of the paths. Even for a very small program, it may contain a large amount of logical paths, which make complete path coverage impractical. Therefore, the test target needs to be reduced from all possible paths to enough paths. So, based on the universality and test difficulty of the complex software, how to effectively improve the test efficiency is one of the most important problems that need to be resolved in path coverage test. In this case, testers often choose a coverage criterion that is effective and takes lower overhead, or choose a finite subset from the complete logic paths to test. To this problem, one of the major solutions is basis path coverage.

Basis path test is a structural testing technique. It is a technology of reducing the scale of path test, which reduces not only test operation amount but also the repeated generation of test data. Test data generation for basis path coverage can be described as: First construct the control flow graph (CFG) for the test program; and then calculate the cyclomatic complexity of the CFG and determine the basis paths that contain only the independent paths; furthermore, take the basis paths as target paths and search the test data set for them in input fields. For any given target path, we design the test data for these paths to ensure them traversed at least once. That is, at least these test data should be able to cover the basis paths.

However, there are some paths in a program that are infeasible. Hedley et al [11] had carried on the related research on the feasibilities of paths. They pointed out that if there is no input for making a target path to be executed, the path is considered infeasible. The infeasible paths affect the accuracy and efficiency of test data generation, and increase the cost of testing.

In this paper, on the basis of determining the feasibility of paths, this paper proposes an evolutionary approach to generating test data for feasible basis path coverage. The technology adopts genetic algorithm (GA) to generate test data for multi-path coverage, and it need not to search test data for infeasible paths, which can greatly improve the successful rate of the search. The approach only search test data for feasible paths, therefore, it can improve the accuracy and efficiency of test data generation.

2. THE EXPRESSION OF TARGET PATHS
The structures of programming languages can be divided into three types: sequential structure, branch structure and loop structure. The program under test can be expressed as CFG: CFG = (V, E, s, e). Where, V represents a vertex-set of all vertexes in the graph, representing computational statements or expressions; E represents a edge-set of all directed edges in the graph, representing transfer of control between nodes; s and e represent entry vertex and exit vertex respectively. For an orderly sequence $<V_0, \ldots, V_i, \ldots, V_n>$, $V_i \in V$, if the node V_{i+1} can be executed immediately after the execution of V_i, a path can be constituted by the sequence.

Definition 1 Basis Path Set [5] (BPS). Each path in basis path set has the following characteristics:

1) Each path is an independent path;
2) All branches in the test program are visited;
3) The paths that do not belong to the basis path set can be obtained by linear calculation of the basis paths.

Each independent path in basis path set is called a basis path.

According to the measurement of McCabe [12], cyclomatic complexity of a CFG is equivalent to the number of basis paths.

DOI: 10.1145/180921.2180936 http://doi.acm.org/10.1145/180921.2180936

CC(CFG)=e-n+2=number of partitioned areas= number of predicate nodes+1 (1)

Where, e is the number of edges, and n is the number of nodes.

BPS is constructed by baseline flipping method [12] in CFG.

For branch structure or IF-ELSEIF-ELSE structure, when calculating the number of judgment nodes, all of the actual judgment nodes (including every ELSEIF statements and CASE statements) should be included.

The true branch is indicated by the letter T and the false branch is indicated by the letter F. Then the paths from the beginning statement of a program to the end statement can be encoded. An example program maxMin is used to demonstrate the expression method of a path. Figure 1(a) and Figure 1(b) show its source code and CFG respectively.

```
void maxMin(int a, int b, int c){
1       int max=0,min=0;
2       if(a>b){
3         max=a;min=b;}
4       else{
5         max=b;min=a;}
6       if(max<c){
7         max=c;}
8       if (max>c){
9         min=c;}
 }
```

Figure 1(a). Source code of maxMin

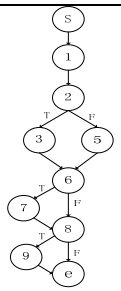

Figure 1(b). CFG of maxMin

Each target path is encoded into the form of hybrid-coding that efficiently combines the statement label with the outcome of a conditional statement (i.e. T or F). Taking Figure 1(b) for example, the basis paths and their corresponding coding are shown below respectively:

$Path_1$=(s, 1, 2T, 3, 6T, 7, 8T, 9, e);
$Path_2$=(s, 1, 2F, 5, 6T, 7, 8T, 9, e);
$Path_3$=(s, 1, 2T, 3, 6F, 8T, 9, e);
$Path_4$=(s, 1, 2T, 3, 6T, 7, 8F, e).

3. ALGORITHM DESIGN

Definition 2 Genetic Algorithm for Test Data Generation (GATD) can be defined as a 9-tuples model: GATD= (C, F, P_0, M, Φ, Γ, Ψ, PT, T). Among them, C is path encoding method; F is individual fitness function; P_0 is initial population; M is population size; Φ is selection operators; Γ is crossover operator; Ψ is mutation operator; PT is the target paths to cover; T is the terminating conditions.

3.1 Fitness Function Designs

Fitness function is the key factor in GA of the evolution process, and its rationality decides the performance of test data generating method. In order to design an appropriate fitness function, we propose a novel generation method of test data set to cover multiple target paths through one time' running of GA. We can compare the advantages and disadvantages of the different evolutionary individuals through one time' running of GA, as well as generate test data for multi-path coverage.

Because target paths are encoded into the form of hybrid-coding that efficiently combines the statement label with the outcome of a conditional statement (i.e. T or F), our approach can truly reflect the matching degree of the path that traversed by test data and each target path, which is advantageous for accurately calculating the fitness value of test data.

There are two ways for calculating fitness value [4]: path-wise and branch-wise (a.k.a. statement-wise and predicate-wise).

The path-wise way is used to measure the deviation degree of the paths that traversed by test data x and the target paths[4], and the deviation degree is recorded as $V(x)$. Take the program in Figure 2(a) for example, if the current target statement is statement 3. There are 2 input data x_1=[-1 , 2, 4] and x_2 = [2, 1, 0] to traverse paths: $s(x_1)$={s, 1, 4, 5, e}, $s(x_2)$={s, 1, 2, 4, e}. The input data x_2 makes conditional statement 2 executed, and x_1 deviates from the branch that the target statement 3 is in after executing the conditional statement 1. Therefore, $V(x_1)$ is greater than $V(x_2)$, and the fitness of x_2 is better than x_1.

```
// input A, B, X
1 if A>1
2   {if B= =0
3     {X=X/A;}}
4 if X>=1
5     {X=X+1}
}
```

Figure 2(a). Source code of example program

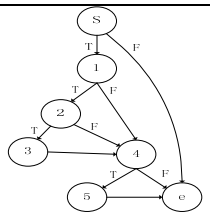

Figure 2(b). CFG of the example program

The branch-wise way is used to calculate the difference between the execution track and the target path in term of predicate values for the "unmatched node-branches" [4], and the difference is recorded as $Dist(x)$. When there is unmatched branch node between the execution track and the target path g_j, according to Korel' proposed branch function[10], the corresponding calculating method of expression for $Dist(x)$ is shown in Table 1. Where, k is the minimum stride of input data. For the complex branch predicate, $Dist(x)$ is the compound of simply predicate.

Table 1. Dist(x) of predicate values

predicate	Dist(x) take false branch	predicate	Dist(x) take false branch
A=B	|A-B|	A>B	(B-A)+k
A≠B	k	A≥B	B-A
A<B	(A-B)+k	A&&B	Dist(A)+Dist(B)
A≤B	A-B	A ‖ B	min(Dist(A), Dist(B))

Suppose an input data (an individual) of the test program is x, the fitness of the individual x is recorded as *fitness(x)*, the execution track is recorded as a character string $s(x)$, and the number of bits for the string is recorded as $|s(x)|$. Suppose when the population is evolved to the n^{th} generation, the number of the target paths is $g(n)$, and the target path i is g_i. The number of bits of target path g_i is recorded as $|g_i|$. *fitness (x)* is caculated according to the matching degree between $s(x)$ and g_i, (i =*1*, *2*, … , $g(n)$). However, when calculating the matching degree between $s(x)$ and g_i, we need to compare each bit of the coding between $s(x)$ and g_i in turn from left to right. $t_{ij}(s(x))$ reflects whether the j^{th} bit encoding of the $s(x)$ and g_i is same, if so, $t_{ij}(s(x))$ is 1; otherwise, $t_{ij}(s(x))$ is 0. Obviously, we need to compare at most min ($|s(x)|$, $|g_i|$) times.

In general, the more the number of bits that are the same, the closer the individual is to the test data of the target path in the bitwise comparison. But when there are two or more individuals that the numbers of bits of their execution tracks are equal to the number of the bits of a target path, we are unable to judge which one is the best. In general, different coding bits have different contribution to closing to the target path, the earlier the coding bit is compared, the more its contribution is. Therefore, in order to accurately judge the fitness value of each individual x and distinguish which one is the best, we need to distinguish the matching degree according to the different encoding bits and design different weight w_i for different encoding bits of a path.

Where, the value of w_i depends on the locations of the different coding bits. In general, the earlier the coding bit is compared, the bigger its weight is. Thus, this paper adopts the decrement method to assign different weights according to the different encoding bits. Recording the matching degree of $s(x)$ and g_i as $V_j'(x)$, $V_j'(x)$ is expressed as:

$$V_i'(x) = \sum_{i=0}^{m} w_i \times t_{ij}(s(x)) = \sum_{i=0}^{m} (m-i) \times t_{ij}(s(x)) \quad (2)$$

Where, n= min ($|s(x)|$, $|g_i|$).

From front to back the bigger the number of bits that are the same in a row, the individual is closer to the test data of the target path. Thus, when we calculate the fitness value of x, we need to make the following adjustments for formula (2). In the process of the comparison of $s(x)$ and g_i from front to back, we record the number of bits that are the same in a row from the first bit as C_k, and record the matching degree that has been adjusted as $V_i''(x)$, then we have the following formula:

$$V_i''(x) = C_k \times V_i'(x) \quad (3)$$

When the fitness value of two or more individuals is the same, we are unable to judge which one is the best if we only consider $V''_i(x)$. In this case, we need to calculate $Dist(x)$ in order to accurately calculate the matching degree of $s(x)$ and g_i. In order to avoid the situation that $V''_i(x)$ is neglected caused by large $Dist(x)$, $Dist(x)$ is normalized as formula (4).

$$Dist_{is(x)}(x) = \sum_{j=1}^{n} Dist_{is(x).j} / (\sum_{j=1}^{n} Dist_{is(x).j} + \alpha) \quad (4)$$

Where, α is a positive integer, here α=1; i is the sequence number of target paths; j is the sequence number of the branch predicate node that exists both in target path and execution track.

From the above analysis, we can see that in order to calculate the matching degree of the execution track of individual x and target path g_i, (a.k.a. fitness value of individual x to target path g_i, recorded as $fitness_i(x)$), we need to take all factors into consideration for $V''_i(x)$ and $Dist_{is(x)}$. In order to comprehensively consider the matching degree of the execution track of individual x and all target paths, we need to take its average as the fitness value of individual x, recorded as $fitness(x)$, and then we can obtain the fitness function of evolutionary method to generating test data for multi-path coverage:

$$fitness(x) = \frac{1}{g(n)} \sum_{i=1}^{g(n)} fitness_i(x) \quad (5)$$

Taking the program in Figure 2(a) as an example, Figure 2(a) shows its source code, and Figure 2(b) shows the corresponding CFG of the example program. Assuming that the encoding of the target paths are g_1 ="sT, 1T, 2F, 4F, e" and g_2="sT, 1F, 4T, 5, e" while evolving to the t^{th} generation. $s(x_1)$, which is the execution track of the first individual x_1, is "sT, 1F, 4F, e"; $s(x_2)$, which is the execution track of the second individual x_2, is "sT, 1T, 2F, 4T, 5, e". In the following, we will calculate the fitness of x_1 and x_2 respectively.

For the fitness value of individual x_1: first, we compare "sT, 1F, 4F, e" with "sT, 1T, 2F, 4F, e",

$fitness_1(x_1) = V''_1(x_1) = C_1 \times V'_1(x_1)$
$= 1 \times (3 \times 1 + 2 \times 0 + 1 \times 1) = 5;$

Then, we compare "sT, 1F, 4F, e" with "sT, 1F, 4T, 5T, e", and can obtain that

$fitness_2(x_1) = V''_1(x_1) = C_2 \times V'_2(x_1)$
$= 2 \times (3 \times 1 + 2 \times 1 + 1 \times 0) = 10.$

Thus, the fitness value of individual x_1:

$$fitness(x_1) = (1/2) \times \sum_{i=1}^{2} fitness_i(x_1) = (1/2) \times (5 + 10) = 7.5.$$

In a similar way, the fitness value of individual x_2 can also be obtained:

$$fitness(x_2) = (1/2) \times \sum_{i=1}^{2} fitness_i(x_2) = (1/2) \times (27 + 7) = 17.$$

By the calculation of the fitness value for the above individuals, we can easily tell the merits of them: x_2 is better than x_1.

3.2 Terminating Conditions for the Algorithms

In this section, we give the steps of GA for evolutionary method to generating test data for basis paths coverage.

Step 1 construct CFG of test program, calculate the cyclomatic complexity of the CFG, derive the basis path set, and select target paths.
Steps 2 instrument test program.
Step 3 design the process of the GA for evolutionary method to generating test data for multi-path coverage, as follows:
Step 3.1 initialization: initial population P, encode individual, and so on;
Step 3.2 decode evolutionarily individual, execute test program;
Step 3.3 determine if there are paths exactly matched, if so, save the corresponding test data for evolutionary individual, remove it from the target paths; if not, goto step 3.4;
Step 3.4 calculate the fitness of evolutionary individual according to formula (3) or (4);
step 3.5 select regenerated individuals according to fitness;
Step 3.6 according to the fitness value of two parents, design crossover operator, generate two new individuals;
Step 3.7 for the two individuals, select their mutation operator based on their fitness, design mutation operators for the corresponding binary variations, generate new individuals;
Step 3.8 produce a new population by crossover and mutation operators, cumulate evolutionary generations, if the number of generation over threshold, goto step 3.9, else goto step 3.2;
Step 3.9 end.
Steps 4 determine whether the termination criteria are met, if so, stop evolution, decode the best individuals, generate test data.

4. EXPERIMENTS

In order to verify the validity of the proposed method, we select 4 benchmark programs and 4 industrial programs as test programs, and we compare the results of the three methods by Gong et al. [2], Ahmed et al. [4] and ours.

Ahmed's method: More than one target paths are considered one time. The sum value of the deviation degree V(x) by path-wise way and the difference between the execution track and the target path D(x) by branch-wise way is as the fitness value for an individual to each target path. And the fitness value of the individual is the average of the total fitness value for the individual to each target path. Through one time' running of genetic algorithm, it can generate the test data set of all target paths.

Gong's method: More than one target paths are considered one time. When the fitness value for an individual to each target path is calculated, the deviation degree V(x) by path-wise way is taken into consideration, but the difference between the execution track and the target path D(x) by branch-wise way is ignored. The fitness value of the individual is the average of the total fitness value for the individual to each target path. Through one time' running of genetic algorithm, it can generate the test data set of all target paths.

DOI: 10.1145/180921.2180936 http://doi.acm.org/10.1145/180921.2180936

4.1 Experiments in Benchmark Programs

4.1.1 The experimental data and parameters specification

In this section, in order to explain and validate the method proposed in this paper, we select 4 test programs as SUTs for experiment, which are: triangle classifier [2, 4], bubble sort [2, 4], minimum-maximum [2, 4], and the program combined with minimum-maximum and triangle classifier (mmt) [2, 4]. The comparison is conducted from two different evolutionary indicators such as evolutionary generations and the evolutionary time for generating the test data of the same target path. For the different methods, experiments are set to the same parameters, the same initial population, and the same run times. The running environment is VC++6.0, the frequency of the machine is 2.80GHz, and the memory is 2GB [2]. Among them, program descriptions are shown in Table 2, and experimental parameters are set as shown in Table 3. The termination condition of the experiment is that the test data of all target statements is found or evolutionary generations reach the maximum number (5000). If there is a path that has not been covered in the maximum evolutionary generations, the path is considered unreachable.

Table2. Program specifications

No.	SUTs	Array length	No. of paths	No. of infeasible paths	No. of feasible basis paths
1	triangle classifier	3	40	0	8
2	bubble sort	3	8	2	5
3	minimum-maximum	3	16	4	4
4	mmt	3	40	22	7
5	bubble sort	4	64	40	10

Table 3. Parameters setup

Parameters	Selection method	Crossover method	Mutation method
Value	Roulette wheel	Single point	Single point
Parameters	**Coding method**	**Crossover probability**	**Mutation probability**
Value	Mixed coding	0.9	0.25

(1) Triangle classifier

Given three numbers, it is a program to classify whether these numbers form a triangle or not. If they are, then the program determines whether the triangle is scalene, isosceles, or equilateral. The program has seven selection statements.

(2) Bubble sort

Given an array of numbers, it is a program to sort these numbers in an increasing order. The program has two loops that are nested and one selection that is nested inside the inner loop. For bubble sort program, we conduct 2 groups experiments, the length of the array are 3 and 4 respectively.

(3) Minimum-maximum

Given an array of numbers, the length of the array is variable, and restricts the content of the array to integer numbers. It is a program to find the minimum and maximum numbers within the array. The program has one loop and two sequential selection statements inside the loop.

(4) Combined program of minimum-maximum and triangle classifier (mmt)

In order to investigate the performance of our fitness function, this program is formed by combining minimum-maximum program and triangle classifier program to allow more complexity. Given three numbers, this combined program outputs both the value of minimum-maximum and the classification of triangle. The program has six conditional statements. In the part of triangle classifier, there are three selection statements in which all the decisions are compound predicates.

4.1.2 Experimental results

For triangle classifier program, we first choose 8 feasible basis paths as the target paths according to the different data ranges. During the experiment, we record the evolutionary generations for finding the test data for each path, and calculate the average of evolutionary generations for 100 times execution of GA. Because the evolutionary time of each time is a millisecond, we record the total evolutionary time of 100 times. Experimental results are shown in Table 4.

It can be seen from Table 4 that for different ranges of input data to generate the test data of target paths, average evolutionary generations and evolutionary time are the least by the approach of this paper.

In order to further verify the performance of the proposed approach in this paper, we also conducted some experiments on other test programs. The range of input data is in [0, 1000]. Where, the population size and the number of target paths of the test programs are shown in Table 5. Average evolutionary generations and evolutionary time by the three methods are also shown in Table 5.

Table 4. Experimental results of triangle classifier program

Data range	Population size	Evolutionary generations			Evolutionary time (Unit: s)		
		Our method	Gong's method	Ahmed's method	Our method	Gong's method	Ahmed's method
[0,100]	10	5.2	8.5	19.3	0.16	0.31	0.65
[0,200]	30	16.5	21.6	43.1	0.31	0.47	0.73
[0,500]	50	35.6	40.9	315.7	0.62	0.73	2.07
[0,1000]	80	53.8	67.1	493.6	0.93	1.32	4.83

Table 5. Experimental results of the other benchmark programs

Program	Array length	Population size	No. of feasible basis path	Average evolutionary generations			Evolutionary time (Unit: s)		
				Our method	Gong's method	Ahmed's method	Our method	Gong's method	Ahmed's method
bubble sort	3	10	6	4.1	6.3	12.3	0.064	0.168	0.351

minimum-maximum	3	100	4	5.6	7.8	15.9	0.149	0.736	2.658
mmt	3	100	7	7.4	9.6	21.9	0.745	1.217	2.928
bubble sort	4	30	10	13.1	17.9	50.2	0.235	0.413	0.862

It can be seen from Table 5: Along with the increasing of input data range, it increasingly is difficult to search test data. For example, for bubble sort program, the average evolutionary generations and the evolutionary time for generating the test data of target paths are both increased with the increasing of input data range.

For evolutionary generations, (1) the average evolutionary generations of our method is less than that of Gong and Ahmed. For example, for the bubble sort program (array length is 3), average evolutionary generations of this method is 4.1, but Gong and Ahmed are 6.3 and 12.3 respectively; (2) average evolutionary generations of Gong's method is less than that of Ahmed. For example, for the minimum-maximum program, average evolutionary generations of Gong's method is 7.8, but it is 15.9 when using Ahmed's method.

For evolutionary time, (1) the evolutionary time of our method is less than that of Gong and Ahmed. For example, for the mmt program, average evolutionary time of this method is 0.745 seconds, but Gong and Ahmed are 1.217 seconds and 2.928 seconds respectively; (2) average evolutionary time of Gong's method is less than that of Ahmed. For example, for the bubble sort program (array length is 4), average evolutionary time of Gong's method is 0.413 seconds, but it is 0.862 seconds when using Ahmed's method.

We can see from Table 5 that for all of above 4 benchmark programs, in order to generate test data for the target paths, both average evolutionary generations and evolutionary time by our approach are the least compared with the other two approaches, especially for the evolutionary time. This fully shows that for the above programs under test, our method is superior to the methods of Gong [2] and Ahmed [4]. From the results of the bubble sort program, we can see that our method is particularly significant for the complex paths.

4.2 Experiments in Industry Programs

4.2.1 Experimental design

To further verify the validity of the proposed method in this paper, 4 industry programs (Nanoxml\Elevator\Vector\Red-Black-Tree) are selected as SUTs for the further experiments, which are provided by SIR (Software-artifact Infrastructure Repository)[13], the popular Open Source software websites. There are a large number of conditional statements in these programs, and there are correlations between conditional statements. Their source code can be downloaded from the SIR. Because the test data of the paths in each function have influence on the test data of the paths in whole program, we select some sub functions in each program for the experiments. Table 6 shows the basic information about the tested programs. We can find that the lines of code for the 4 tested programs range from hundreds to thousands, and the basis paths for the selected functions also have great differences. They represent the programs of different levels of difficulty and types.

Table 6. Industry program specifications

Program	Loc	No. of selected functions	No. of feasible basis paths
Nanoxml	7646	3	12
Elevator	580	7	96
Vector	254	2	17
Red-Black-Tree	334	11	26

In order to further verify the advantages of the proposed method in this paper, we conduct another group of experiments to compare with the two multi-goal methods of Gong and Ahmed respectively. In the experiment, all paths are binary encoding, using roulette wheel selection, single point crossover and single point mutation method. In addition, the other parameters such as selection probability, crossover probability and mutation probability and population size are shown in Table 7. The termination condition of the experiment is that the test data of all target statements is found or evolutionary generations reach the maximum number (20000). If there is a path that has not been covered in the maximum evolutionary generations, the path is considered unreachable.

4.2.2 Experimental results

During the experiments, the algorithm is run 100 times for each test program; meanwhile, the needed average evolutionary generations and evolutionary time of each program is recorded. The experimental results are shown in Table 7.

Table 7. Experimental results of the industry programs

Program	Array length	Crossover probability	Mutation probability	Population size	Average evolutionary generations			Evolutionary time (Unit: s)		
					Our method	Gong's method	Ahmed's method	Our method	Gong's method	Ahmed's method
Nanoxml	3	0.8	0.1	100	37.8	42.1	79.6	1.085	3.213	8.690
Elevator	4	0.9	0.3	300	326.7	437.5	1071.9	9.326	38.916	101.522
Vector	4	0.8	0.1	100	52.3	57.8	116.4	2.130	7.827	22.762
Red-Black-Tree	3	0.9	0.3	200	108.5	130.6	567.3	3.137	11.526	29.951

It can be seen from Table 7: For evolutionary generations, (1) the average evolutionary generations of our method is less than that of Gong and Ahmed. For example, for Nanoxml program, average evolutionary generations of this method is 37.8, but Gong and Ahmed are 42.1 and 79.6 respectively; (2) average evolutionary generations of Gong's method is less than that of Ahmed. For example, for Vector program, average evolutionary generations of Gong's method is 57.8, but it is 116.4 using Ahmed's method.

For evolutionary time, (1) the evolutionary time by our method is less than that of Gong and Ahmed. For example, for Elevator program, evolutionary time of this method is 9.326 seconds, but Gong and Ahmed are 38.916 seconds and 101.522 seconds respectively; (2) the evolutionary time of Gong's method is less than that of Ahmed. For example, for Red-Black-Tree program, the evolutionary time of Gong's method is 11.526 seconds, but it is 29.951 seconds using Ahmed's method.

Overall, for all of above 4 industry programs, in order to generate test data set for the target paths, average evolutionary generations and evolutionary time by our approach are the least compared with the other two approaches, especially for the evolutionary time. From the results of Elevator and Red-Black-Tree programs, we can see again that our method is particularly significant for the complex paths. Through experiments, we find that parameter settings in genetic algorithms have great influence on the generating efficiency of test data: Some complex paths require higher probability and mutation probability to increase the diversity of the population and the speed of evolution, such as Elevator and Red-Black-Tree programs. Therefore, for the problem of evolutionary method to generating test data for multi-path coverage, the proposed method fully illustrates the rationality of the fitness function proposed in this paper.

4.3 Analysis of Experimental Results

(1)Analysis of path coding method and time complexity

For Gong's approach [2], the target paths are encoded into a binary string. First, the program under test is expressed as a CFG, and then the CFG is converted into a Huffman tree and the target paths are encoded. It can be seen that the approach not only increases the encoding difficulty but also tends to confuse path encoding. In other words, it can't really reflect the true path trace that the individual traverses. However, the proposed approach in this paper resolved these existing problems. If 'n' indicates the number of program statements, that is, the number of CFG' nodes, the time complexity of constructing Huffman tree is O($n$log$n$), and the time complexity of encoding is O (n*d), where, d is the depth of the Huffman tree. Therefore, the total complexity of Gong' approach is O ($n$log$n$) +O(n*d). However, for the proposed approach in this paper, we adopt the depth-first strategy to search the CFG of the tested program. All target paths are represented with the combining of the corresponding statement number of the CFG' nodes and flow direction of branch (i.e. T and F). It can be seen that our approach can truly reflect the matching degree of encoding for the paths that traversed by test data set and each target path. So our coding method is advantageous for accurately calculating the fitness value of test data. With our approach, the time complexity of encoding the full path is O (n+e), in turn, the time complexity of encoding basis path is far less than O(n+e), where, e is the number of edges in CFG.

(2)Comparison of methods for designing fitness function

The approach by Gong et al. [2] did not calculate *Dist(x)*—the sum of predicate values for each "unmatched node-branches", which obviously reduces the computing effort. They only calculate *V(x)*—the deviation degree of the execution tracks and the target paths. Therefore, once there are two or more individuals with the same fitness values, it cannot distinguish which individual is better or worse. Therefore, it cannot accurately reflect the matching degree of encoding for the paths traversed by test data set and each target path, so there is a great limitation. However, although the fitness function designed by Ahmed et al. [4] combined the two ways— path-wise ($V(x)$) and branch-wise ($Dist(x)$), *Dist(x)* is the sum of predicate values for each "unmatched node-branches". Therefore, each predicate value needs to be calculated once. Obviously, it would take a very high price. As for the proposed approach in this paper, we do not calculate *Dist(x)* until there are two or more individuals with the same fitness values through calculating $V(x)$. In other words, it needn't to calculate predicate values for every "unmatched node-branches". Thus, it can also save a lot of time. In addition, we propose a novel assigning method for the weight of each bit in path encoding, which can improve the accuracy of fitness value.

5. RELATED WORKS

For the problem of test data generation based on path coverage, there are some solutions mainly aiming at two aspects (a.k.a. single-path coverage approach and multi-path coverage approach).

There are some typical single-path coverage approaches. Lin et al. [15] discussed a genetic algorithm that can automatically generate test data for a selected path. This algorithm took a selected path as a target and executed sequences of operators iteratively for test data to evolve. Even when there were loops in the target path, the algorithm can lead the execution to flow along the target path. This paper showed that genetic algorithms are able to meaningfully reduce time required for lengthy testing by generating test cases for path testing in an automatic way. Xie et al. [3] provoked an approach to construct the fitness function for test case generation in path-oriented ET based on the similarity evaluation techniques. The approach showed superiority over several other path-oriented testing techniques, especially for the complex paths.

For multi-path coverage approach, Ahmed and Hermadi [4] viewed that for each target path, the fitness value of chromosomes is affected by all target paths. Therefore, the fitness value of chromosomes is determined by the integrated fitness value of all target paths. Existing methods of multi-path coverage are mainly: Ahmed et al. [4] first proposed evolutionary method to generating test data for multi-path coverage. They transformed the problem of generating test data to the optimization problem of many targets. When using GA to solve the problem, they calculated the fitness value of evolutionary individual on different target paths respectively, which generated many test data for many target paths through one time' running of GA[7]. The experiment has validated that their approach is better than the traditional approach of evolutionary method to generating test data for single path coverage, but the approach is inefficient, which mainly reflected in: the fitness value of an individual for one target path designed by Ahmed et al. [4] combined the two ways: path-wise ($V(x)$) and branch-wise ($Dist(x)$), and the average of the fitness value for all target paths is taken as the fitness value of the individual. It can be seen that *Dist(x)* is the sum of predicate values for each "unmatched node-branches". Thus, each predicate value needs to be calculated once, therefore, it would take a very high price. The low efficiency leads that the approach is greatly restricted in practical application of software testing. For Gong's approach [2], the target paths are encoded into a binary string. They took the degree of the execution track matching the target paths as the fitness function. When the encoding of an execution track and the coding of a target path are full matched, the test data of the target path is found. They have validated that their approach is better than the approach of Ahmed through experiments. But they only use the path-wise way ($V(x)$) to calculate the fitness value, which is not accurate. Cao et al. [6] have proposed a fitness function to generating test data for a specific single path, which is different from the predicate distance applied by most test data generators based on GA. This study had presented two fitness functions to generating test data for a specific path and multi-paths in one search based on similarity between the

target path and the execution path with GAs. However, they ignore that the elements in paths may affect the path similarity with different weights that will be a factor for influencing fitness function.

For the single-path coverage approach, it can only generate test data for one target path through one time' running of genetic algorithm. The evolutionary generations of the test data are the sum of the evolutionary generations of the test data for each target path. In contrast, for the multi-path coverage approach, it can generate test data set of all target paths through one time' running of genetic algorithm. The evolutionary generations of the test data are the evolutionary generation of the test data for the final target path (when the final test data is found). Therefore, it is clear that the latter one is more efficient.

Based on the two methods by Ahmed et al. and Gong et al., this paper proposed a novel evolutionary approach to generating test data for multi-path coverage, which cannot only avoid redundant test but also improve the efficiency of test data generation effectively.

6. CONCLUSIONS

Path coverage is an important method of software testing. However, the existing evolutionary approaches to generating test data for multi-path coverage are inefficient. In order to further improve the efficiency of the evolutionary method to generating test data, this paper proposed an evolutionary method to generating test data for multi-path coverage. The experimental results show that the method in this paper has greater advantages than that of Gong et al. and Ahmed et al. Especially when the programs are complex and contain a lot of infeasible paths, our approach can optimize the target paths and increase the search successful rate. The research achievement of this paper enriches the theory of evolutionary method to generating test data for multi-path coverage, and provides new ideas for improving the efficiency of evolutionary method to generating test data. In addition, this method can also further verify the feasibility of basis paths in static analysis stage.

There are two major works that will be studied in the future. (1) For the test programs with more complex data structures, how to construct a suitable fitness function to address the problem of evolutionary method to generating test data. (2) How to optimize the GA to further improve the efficiency of evolutionary method to generating test data.

ACKOWLEDGEMENTS

This work was supported in part by awards from National Natural Science Foundation under 60970032, Natural Science Foundation of Jiangsu Province under BK2008124, Qing Lan Project and Graduate Training Innovative Projects Foundation of Jiangsu, China under CX10B_157Z.

7. REFERENCES

[1] Shan, J.H., Wang J., and Qi, Z.C. 2004. Survey on path-wise automatic generation of test data. *Chinese J. Electron*. 32, 1, (January. 2004), 109-113.

[2] Gong, D.W. and Zhang, Y. 2010. Novel evolutionary generation approach to test data for multiple paths coverage. *Chinese J. Electron*. 38, 6, (June. 2010), 1-6.

[3] Xie, X.Y., Xu, B.W., Shi, L., and Nie, C.H. 2010. Genetic test case generation for path-oriented testing. *Chinese J. Softw*. 20, 12, (December. 2010), 3117-3136.

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

[5] Binkley, David W. 2009. FlagRemover: a testability transformation for transforming loop assigned flags. *ACM T Softw. Eng. Meth.* 2, 3, (June. 2009), 110-146.

[6] Cao, Y., Hu, C.H., and Li, L.M. 2009. Search-based multi-paths test data generation for structure-oriented testing. In *Proceedings of the 1st ACM/SIGEVO Summit on Genetic and Evolutionary Computation* (Shanghai, China. June 12-14, 2009). GEC '09. ACM, New York, NY, 25-32. DOI=http: //dx.doi.org/ 10.1145/1543834.1543839.

[7] Rajappa, V., Biradar, A., and Panda, S. 2008. Efficient software test case generation using genetic algorithm based graph theory. In *Proceedings of the 1st International Conference on Emerging Trends in Engineering and Technology* (Nagpur, India, July 16-18, 2008). ICETET'08. The IEEE computer society, New York, NY, 298-303. DOI= http: //dx.doi.org/ 10.1109/ICETET.2008.79.

[8] Bouchachia, A. 2007. An immune genetic algorithm for software test data generation. In *Proceedings of the 7th International Conference on Hybrid Intelligent Systems* (Kaiserslautern, Germany, September 17-19, 2007). HIS'07. The IEEE computer society, New York, NY, 84-89. DOI= http: //dx.doi.org/ 10.1109/HIS.2007.37.

[9] Sofokleous, A.A. and Andreou, A.S. 2008. Automatic, evolutionary test data generation for dynamic software testing. *J. Syst. Softw.* 81, 11(November. 2008), 1883-1898.

[10] Chen, Y. and Zhong, Y. 2008. Automatic path-oriented test data generation using a multi-population genetic algorithm. In *Proceedings of the 4th International Conference on Natural Computation* (Jinan, China, August 25-27, 2008). ICNC'08. The IEEE computer society, New York, NY, 566-570. DOI= http://dx.doi.org/10.1109/ICNC.2008.388.

[11] Korel, B. 1990. Automated software test data generation. *IEEE T. Software Eng.* 16, 8, (August. 1990), 870-879.

[12] Hedley, D. and Hennell, M. A. 1985. The causes and effects of infeasible paths in computer programs. In *Proceedings of the 8th International Conference on Software Engineering* (London, UK. August 28-30, 1985). ICSE '85. IEEE Computer Society, Los Alamitos, CA, USA, 259-266.

[13] McCabe, T.J. 1987. Structural Testing: A Software Testing Methodology Using the Cylomatic Complexity Metric. Baltimore: McCabe and Assaciates.

[14] SIR. A repository of software-related artifacts meant to support rigorous controlled experimentation. Available at: http://sir.unl.edu/portal/index.html.

[15] Lin, J.C., and Yeh, P.L. Automatic test data generation for path testing using GAs. *Inform Sciences*. 131, 1–4, (January 2001), 47-64