Using Genetic Algorithms for Test Case Generation in Path Testing

Jin-Cherng Lin and Pu-Lin Yeh
Dept. of Computer Science and Engineering, Tatung University
Taipei 10451, Taiwan
Email: plyeh@ms1.tisnet.net.tw

Abstract

Genetic algorithms are inspired by Darwin's the survival of the fittest theory. This paper discusses a genetic algorithm that can automatically generate test cases to test a selected path. This algorithm takes a selected path as a target and executes sequences of operators iteratively for test cases to evolve. The evolved test case can lead the program execution to achieve the target path. A fitness function named SIMILARITY is defined to determine which test cases should survive if the final test case has not been found.

Keywords: Path testing, Test Cases Generation, Genetic Algorithms.

1. Introduction

The basic concepts of genetic algorithms were developed by Holland [1, 2]. GAs include a class of adaptive searching techniques which are suitable for searching a discontinuous space.

Genetic algorithms have been used to find automatically a program's longest or shortest execution times. In the paper about testing real-time systems using genetic algorithms [8], J. Wegener, et. al. investigated the effectiveness of GAs to validate the temporal correctness of real-time systems by establishing the longest and the shortest execution times. The authors declared that an appropriate fitness function for the genetic algorithms is found, and the fitness function is to measure the execution time in processor cycles. Their experiments using GAs on a number of programs have successfully identified new longer and shorter execution times than those had been found using random testing or systematic testing.

Genetic algorithms have also been used to search program domains for suitable test cases to satisfy the all-branch testing [3, 4, 5]. In their papers about automatic structural testing using genetic algorithms, B. F. Jones, et. al. [3, 5] showed that appropriate fitness functions are derived automatically for each branch predicate. All branches were covered with two orders of magnitude fewer test cases than random testing.

In this paper, we developed a new metric (which is a fitness function) to determine the distance between the exercised path and the target path. The genetic algorithm with the metric is used to generate test cases for executing the target path.

A genetic algorithm to find a test case which generates a given target path is depicted below:

```
Initialize test cases from the domain of the program to be tested at random;
Do
  feed the program with the test cases;
```

```
if any of the test case has reached to the target path;
    output the successful message;
    exit;
endif
Determine which test case should survive with fitness function;
Reproduce the survivors;
Select parents randomly from the survivors for crossover;
Select the crossover sites of the parents;
Produce the next new generation of test cases;
Mutate the new generation of test cases according to the mutation
    probability;
if iteration limit exceeded
    output a failure message
    exit
endif
loop
```

The first generation of test cases is generated at random. Then, the generated cases are fed to the program for execution. One test case will be exercised in one and only one correlated path. The survivors of test cases to the next generation are chosen according to the fitness function, which is a measurement function used to calculate the distances between the executed paths and the target path. Such distances are used to determine which test cases should survive. After all test cases in the present generation are fed, the new generation of test cases is generated by the operators of **reproduction**, **crossover** and **mutation.** The system will automatically generate the next generation of test cases until one of the test cases covers the target path.

Program inputs may be of different types and of complicated data structure, however these inputs can be treated as a single, concatenated bit string denoted as $\{b_1, b_2, \ldots, b_n\}$.

2 Development of the Fitness Function

In branch testing, Hamming Distance has been used to measure the difference between the covered branches and the selected branches as the fitness [4]. This distance metric can only be used to measure the distance of two objects in which they have no specific sequences. For the tested elements in all-nodes or all-branches testing criterion, no tested sequence is needed. However for path testing, two different paths may contain the same branches but in different sequences. The simple Hamming Distance is not longer suitable.

We extend the Hamming Distance from the first order to the n-th order ($n>1$) to measure the distance between two paths. Such extension is hereby named as Extended Hamming Distance (EHD).

Hamming Distance is derived from the *symmetric difference* in set theory. The *symmetric difference* of the set α and the set β (denoted as $\alpha \oplus \beta$) is a set containing the elements either in α or β but not in both. In other words, $\alpha \oplus \beta$ equals to $(\alpha \cup \beta)-(\alpha \cap \beta)$. In this paper, the cardinality of the *symmetric difference* is defined as the *distance* between α and β: $D_{\alpha-\beta}=|\alpha \oplus \beta|$. The notation, $|\alpha \oplus \beta|$, denotes the cardinality of $\alpha \oplus \beta$.

The *distance* $D_{\alpha-\beta}$ is normalized to become a real number:

$$N_{\alpha-\beta} = \frac{D_{\alpha-\beta}}{|\alpha \cup \beta|},$$

where $N_{\alpha-\beta}$ represents the *normalized distance* between sets α and β, and $0 \leq N_{\alpha-\beta} \leq 1$, because $D_{\alpha-\beta} = |\alpha \oplus \beta| = |\alpha \cup \beta| - |\alpha \cap \beta| \leq |\alpha \cup \beta|$.

Replace $D_{\alpha-\beta}$ with $|\alpha \cup \beta| - |\alpha \cap \beta|$ to derive

$$N_{\alpha-\beta} = \frac{|\alpha \cup \beta| - |\alpha \cap \beta|}{|\alpha \cup \beta|} = 1 - \frac{|\alpha \cap \beta|}{|\alpha \cup \beta|}$$

$$\Rightarrow (1 - N_{\alpha-\beta}) = \frac{|\alpha \cap \beta|}{|\alpha \cup \beta|} = M_{\alpha-\beta},$$

where $M_{\alpha-\beta}$ = (1- the *normalized distance* between α and β). The function $M_{\alpha-\beta}$ is named as *SIMINARITY*, and is used to measure the similarity between α and β.

In the following section, $M_{\alpha\text{-}\beta}$ is used to measure the similarity between two paths in a program control-flow diagram.

If P is a control-flow diagram of a given program and Q is the set of all complete paths within P, then,

Q = {$\boldsymbol{path_i}$ | $\boldsymbol{path_i}$ is a **complete path** within P}

= {$\boldsymbol{path_1}$, $\boldsymbol{path_2}$, • • •, $\boldsymbol{path_z}$},

where z = the number of complete paths in P,

$\boldsymbol{path_i}$ = the i-th complete path in P, $1 \le i \le z$.

Let S_1^1={g | g is a branch of $\boldsymbol{path_1}$},

S_1^2={ h | h is an ordered pair of cascaded branches of $\boldsymbol{path_1}$},

• • •

S_1^n={ k | k is an ordered n_tuple of cascaded branches of $\boldsymbol{path_1}$, $n \le |S_1^1|$ },

• • •

S_q^t={ r | r is an ordered t_tuple of cascaded branches of $\boldsymbol{path_q}$, $t \le |S_q^1|$ and $1\le q\le z$},

• • •

The first order *distance* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is expressed as, $D_{i-j}^1 = |S_i^1 \oplus S_j^1|$. The normalized first order *distance* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is expressed as, $N_{i-j}^1 = \dfrac{D_{i-j}^1}{|S_i^1 \cup S_j^1|}$. The first order *similarity* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is defined as, $M_{i-j}^1 = 1 - N_{i-j}^1$.

The second order *distance* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is expressed as, $D_{i-j}^2 = |S_i^2 \oplus S_j^2|$. The normalized second order *distance* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is expressed as, $N_{i-j}^2 = \dfrac{D_{i-j}^2}{|S_i^2 \cup S_j^2|}$. The second order *similarity* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is defined as, $M_{i-j}^2 = 1 - N_{i-j}^2$.

The m-th (m=$1..n$) order *distance* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is expressed as, $D_{i-j}^m = |S_i^m \oplus S_j^m|$.The normalized m-th order *distance* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is expressed as, $N_{i-j}^m = \dfrac{D_{i-j}^m}{|S_i^m \cup S_j^m|}$. The m-th order *similarity* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ is defined as, $M_{i-j}^m = 1 - N_{i-j}^m$. The notation D_{i-j}^m is the m-th order EHD between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$. The notation N_{i-j}^m is the m-th order Normalized Extended Hamming Distance (NEHD) between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$. The notation M_{i-j}^m named as the m-th order *SIMILARITY* between $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ where $1 \leqq m \leqq n$. Note that M_{i-j}^m=0 (or N_{i-j}^m=1) if $S_i^m \cap S_j^m = \phi$. It means that $\boldsymbol{path_i}$ and $\boldsymbol{path_j}$ have no common m_tuple cascaded breaches. Larger N_{i-j}^n means greater *difference* between $\boldsymbol{path_j}$ and $\boldsymbol{path_i}$. Contrarily, larger M_{i-j}^n means greater *similarity* between $\boldsymbol{path_j}$ and $\boldsymbol{path_i}$. When $\boldsymbol{path_j}$ and $\boldsymbol{path_i}$ have no common branch, NEHD should take the form of { N_{i-j}^1 =1, N_{i-j}^2 =1, • • •, N_{i-j}^n =1} or { M_{i-j}^1 =0, M_{i-j}^2=0, • • •, M_{i-j}^n=0}, which is resulted from a worst test case. When $\boldsymbol{path_j}$ and $\boldsymbol{path_i}$ are identical, NEHD should take the form of { N_{i-j}^1=0, N_{i-j}^2=0,• • •, N_{i-j}^n=0} or { M_{i-j}^1=1, M_{i-j}^2=1, • • •, M_{i-j}^n=1 }, which is resulted from a perfect test case that force the program to execute along the target path. Therefore, if NEHD is not in the condition of { $N_{i-j}^1 = 0$, $N_{i-j}^2 = 0$, • • •, $N_{i-j}^n = 0$ }, the fitness function *SIMLARITY* between $\boldsymbol{path_j}$ and $\boldsymbol{path_i}$ is defined as: $SIMILARITY_{i\text{-}j} = M_{i-j}^1 \times W_1 + M_{i-j}^2 \times W_2 + M_{i-j}^3 \times W_3 + \bullet\bullet\bullet + M_{i-j}^n \times W_n$.

W's are the weighing factor of fitness and $W_1 < W_2 < W_3 < \bullet\bullet\bullet < W_n$. $n = |S_i^1|$ if $\boldsymbol{path_i}$ is defined as the target path. $n = |S_j^1|$ if $\boldsymbol{path_j}$ is otherwise. Since, if the target path is constructed by n branched, the values of M_{i-j}^{n+1} ,M_{i-j}^{n+2} ,• • should be zeros and are insignificant in *similarity* comparison.

SIMILARITY$_{i\text{-}j}$ determines the fitness between current executed ***path$_j$*** and the target ***path$_i$***. The greater *SIMILARITY*$_{i\text{-}j}$ leads to the better fitness. The higher order *similarity* is more significant than its lower order counterpart. The highest-ordered *similarity* between ***path$_i$*** and ***path$_j$*** (M^n_{i-j}) is therefore the most significant one. The semantics of the tested program has much effect on the values of W's. Determining the values of W's is quite difficult and is usually done via experience. $W_{k+1}= y^* W_k$ means the *(k+1)*th order *similarity* is y times more significant than that of the *k*th order.

Let the least significant weight (W_1) be 1. In experience, the W's for ***path$_i$*** may be assigned as:

$$W_1 = 1,$$
$$W_2 = W_1 \times | S^1_i |$$
$$W_3 = W_2 \times | S^2_i |$$
$$\bullet \ \bullet \ \bullet$$
$$W_n = W_{n-1} \times | S^{n-1}_i |$$

The distance between ***path$_i$*** (the target path) and ***path$_j$*** is larger than the distance between ***path$_i$*** and ***path$_k$*** if *SIMILARITY*$_{i\text{-}k}$ > *SIMILARITY*$_{i\text{-}j}$. Therefore, ***path$_k$*** is closer to the target than ***path$_j$*** is. The *SIMILARITY* function can help the algorithm to search the program domain and to find fitter test cases. Even when there are loops in the target path, the function can help the algorithm to lead the execution to flow along the loops of the target path.

3. Results

In the general path testing experiment, the basic steps used are given below.

(1) Control flow graph construction:

The tested program (Figure 1 Triangle Classifier) determines what kind of triangle can be formed by any three input lengths. The program's control flow diagram is shown in Figure 2. In this program, a set of labels is implemented to indicate when the path is executed. Therefore, executing the implemented program under test with each test case can produce a string of labels for the fitness function.

(2) Target Path selection:

The path "abc" in Figure 2 is the most difficult path to be covered in random testing. The program in Figure 1 has three integers as input parameters. While the three parameters are positive and equal, the path "abc" can be covered. The covered probability of this path is $2^{-30}(=1*2^{-15}*2^{-15}$ for each positive integer is 15 bits). To show the ability of searching test cases for specific paths by using genetic algorithm is much greater than by using random testing, the path 'abc" is selected as the target path in this example.

```
# include <string.h>
char path[256];
int TriangleA(unsigned int a, unsigned int b, unsignet int c)
{   int Triangle = 0;                                          /*b1*/
    if (( a + b > c ) && ( b + c > a ) && ( c + a > b ))       /*b1*/
    {   strcat(path,"a");                           /* instrumentation */
        if (( a != b) && ( b != c )&& ( c != a ))              /*b2*/
        {   strcat(path,"e");                       /* instrumentation */
            Triangle = 1; }                           /*Scalene b3*/
        else
        {   strcat(path,"b");                       /* instrumentation */
            if (((a == b) && (b != c))|| ((b == c) && (c != a))||
               ((c == a) && (a != b)))                         /*b4*/
            {   strcat(path,"f");                   /* instrumentation */
                Triangle = 2; }                      /* Isosceles b5*/
            else
            {   strcat(path,"c");                   /* instrumentation */
                    Triangle = 3; }  }               /*Equilateral 6*/
        }
    else
    {   strcat(path,"d"); }     /* instrumentation ( not a triangle) */
    return ( Triangle );                                       /*b7*/
}
```

Figure 1. Triangle Classifier [6, 7].

The comments (/*b$_1$*/ • • •/*b$_7$*/) denote the program blocks 1 to 7 respectively. In Figure 2, the nodes 1 to 7 indicate the control flow locations of these blocks.

(3) Test case generation and execution:

According to the genetic algorithms, an experimental tool for generating test cases automatically in order to test a specific path is developed. The tool divides the input string of 48-bit length into three genes using the conjunctions between two integers as the fixed crossover site. Pairs of test cases were combined using two-point crossover algorithm. The crossover rate is set to 0.9. Traditionally, the mutation probability is set to the reciprocal of the length of the bit string [4]. Hence, the mutation probability is set to 1/48.

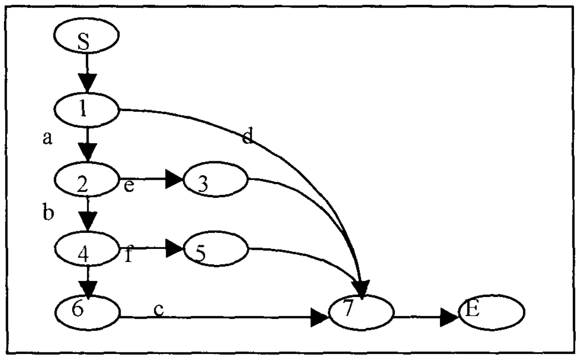

Figure 2. The control flow diagram of the program shown in Figure 1.

In this experiment, the first generation of test cases was chosen from the tested program's domain randomly. Then the tested program was executed with these test cases. The executed results are evaluated by fitness function to determine which test cases should survive to generate the next generation. Again, new generations of test cases are generated by *reproduction*, *crossover* and *mutation*. The average fitness of each generation is steadily improved until the target path is achieved. In the initial generations, the execution of generated test cases were mostly group in the path <d>. In the subsequent generations, the execution of generated test cases gathered in the path <abe>. According to the experiments, about 52.5 percent and 47.5 percent of the first generation test cases, which were generated at random, executed the path <d> and the path <abe> respectively, and none of the test cases executed the other paths. After the fifth generation, all the evolution of the test cases left the path <d> and mostly gathered in the path <abe>. Afterward, the execution of generated test cases approached the path <abf> gradually. Finally, at least a test case reached the path <abc> and succeeded in generating the test case. After one hundred experiments, the results on Table 1 show that the target path was obtained within 10100 test cases (i.e. *100 test cases + 10 generations × 1000 test cases in each generation)* by average. While, based on the theory of probability, it will take random testing 2^{30} tests to reach the target.

Table 1 shows the evolution of test cases from the first generation to the 11th generation. These values were averaged from 100 experiments. Before the fifth generation, the algorithm decreased the number of test cases on the path <d> and increased the number of test cases on the path <ae>. After the fifth generation, the number of test cases on the path <ae> was decreased and the number of test cases on the path <abf> increased speedily.

Generations	•Path <d>	■ Path <ae>	▲ Path <abf>	⊠Path <abc>	The No. of test cases
0	525	475	0	0	1000
1	33	966	1	0	1000
2	8	910	82	0	1000
3	4	870	126	0	1000
4	1	600	399	0	1000
5	0	314	686	0	1000
6	0	214	786	0	1000
7	0	175	825	0	1000
8	0	102	898	0	1000
9	0	54	946	0	1000
10	0	3	96	1	100

Table 1. The Average Number of Test Cases on the Paths of Figure 1 in Each Generation

(4) Test result evaluation:

This step is to execute the selected path with the test cases found in step (3) and to determine whether the outputs are correct or not.

Since there is no guarantee that the algorithm can find the target within definite number of runs, the execution of the algorithm was allowed to continue for 450 generations before it was stopped. Fortunately, the evolutions of the test cases to execute the target path were less than 18 generations in our experiments. We have also applied 100 test cases and 10000 test cases in each generation in the same experiment. It is found that the best result is to apply 1000 test cases in one generation. In summary, the number of individuals of one generation should be large enough to maintain diversity, yet small enough to avoid an excessive number of tests.

4. Conclusion

In this paper, the genetic algorithms are used to generate test cases automatically for path testing. The greatest merit of using the genetic algorithm in program testing is its simplicity. The quality of test cases produced by genetic algorithms is higher than the quality of test cases produced by random way because the algorithm can direct the generation of test cases to the desirable range fast. When Compare to random testing, use of Extended Hamming Distance to derive *SIMILARITY* (a fitness function) is a very useful approach for path testing. This paper shows that genetic algorithms are useful in reducing the time required for lengthy testing meaningfully by generating test cases for path testing in an automatic way.

5. Reference

[1] J. H. Holland, "Genetic algorithms and the optimal allocation of trials" SIAM J. Comput., pp. 89-104, 1973.

[2] J. H. Holland "Adaptation in nature and artificial systems" Addison-Wesley, 1975.

[3] B.F Jones, H.-H. Sthamer, X. Yang and D.E. Eyres, "The automatic generation of software test data sets using adaptive search techniques", Third International Conference on Software Quality Management", Seville (1995), pp. 435-444 (BCS/CMP).

[4] B. F. Jones, H-H. Sthamer and D. E. Eyres, " Automatic Structural Testing Using Genetic Algorithms", Software Engineering Journal, 1996, 9, pp. 299-306.

[5] B. F. Jones, D. E. Eyres, H.-H Sthamer, "A strategy for using genetic algorithms to automate branch and fault-based testing," The Computer Journal, Vol. 41, 1998, pp.98-107

[6] P. N. Lee, "Correspondent Computing", Proceeding of ACM 1988 Computer Science Conference, Atlanta Georgia, pp. 12-19, 1988.

[7] G. Myers, "The art of Software Testing", Wiley, 1979.

[8] J. Wegener, H. Sthmer, B. F. Jone and D. E. Eyres, "Testint Real-Time System Using Genetic Algorithms", Software Quality Journal 6, 1997, pp. 127-153.