ELSEVIER

Information Sciences 131 (2001) 47–64

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

Automatic test data generation for path testing using GAs

Jin-Cherng Lin, Pu-Lin Yeh *

Department of Computer Science and Engineering, Tatung University, Taipei 10451, Taiwan, ROC

Received 1 March 1999; received in revised form 23 May 2000; accepted 30 July 2000

Abstract

Genetic algorithms (GAs) 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 will lead the program execution to achieve the target path. To determine which test cases should survive to produce the next generation of fitter test cases, a metric named normalized extended Hamming distance (NEHD, which is used to determine whether the final test case is found) is developed. Based on NEHD, a fitness function named *SIMILARITY* is defined to determine which test cases should survive if the final test case has not been found. Even when there are loops in the target path, *SIMILARITY* can help the algorithm to lead the execution to flow along the target path.

Keywords: Software testing; Path testing; Genetic algorithms

1. Introduction

The basic concepts of genetic algorithms (GAs) were developed by Holland [7,8]. To date, genetic algorithms are well developed and widely used in the area

* Corresponding author. Tel.: +886-2-25925252 ext. 3295.
E-mail address: plyeh@ms1.tisnet.net.tw (P.-L. Yeh).

0020-0255/01/$ - see front matter
PII: S0020-0255(00)00093-1

of artificial intelligence and have been successfully applied in many fields of optimization [17–21].

It is understood that all occurrences of pre-selected features in a tested program should be executed (e.g. *all-branch testing* requires that each branch in a program should be tested at least once). It is also understood that manually generating a large number of test cases to fulfill the testing criteria is too great an effort. As a result, certain degree of automation is necessary. As Ould [14] suggested, the automation of test case generation is the most important aspect of automatic testing. Automatic test case generation can keep testers from bias when preparing test cases.

Many automatic test case generators have been developed and many classical searching methods have been used to derive test cases [1–4,9]. However, these techniques work only for continuous functions (while most program domains are of discontinuous spaces). This is where GAs step in. GAs include a class of adaptive searching techniques which are able to search a discontinous space.

1.1. The works that apply genetic algorithms to program testing

Genetic algorithms have been used to automatically find a program's longest or shortest execution times. In their paper about testing real-time systems using genetic algorithms [16], 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 that had been found using random testing or systematic testing. However, they did not show the fitness function in their work.

Genetic algorithms have also been used to search program domains for suitable test cases to satisfy the all-branch testing [6,10,11]. In their papers about automatic structural testing using genetic algorithms, Jones et al. 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. The adequacy of the test cases is improved by the genetic algorithm's ability to generate test cases, which are at or close to the branch predicates.

Consider a predicate such as

If $A = B$ then $\cdots$

The fitness function for this predicate is based on the Hamming distance* between A and B. The authors declare that the reciprocal of Hamming distance leads to a suitable fitness function.

*The Hamming distance is a count based on the number of different bits in the bit patterns for A and B (i.e. $A \oplus B$). Hamming distance between two bit patterns of equal length is defined as the number of different bits in the bit patterns. For example, the distance between 000 and 111 is three, while the distance between 010 and 011 is one.

The results of the papers above confirm that genetic algorithms are more efficient and effective than random testing in all-branch testing and finding longest execution time. 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 testing the target path. Such path-testing algorithm is designed to:

1. select a target path from the program under test,
2. sample test cases from the whole domain of the program,
3. execute the program with the test cases to exercise paths, and calculate the fitness between the target path and each exercised path, and then
4. evolve into new generation of fitter test cases,
5. repeat from the step 3 to the step 4 until the fittest test case is found.

2. Genetic algorithms

Genetic algorithms begin with a set of initial individuals as the first generation, which are sampled at random from the problem domain. The algorithms are developed to perform a series of operations that transform the present generation into a new, fitter generation.

Each individual in each generation is evaluated with a fitness function. Based on the evaluation, the evolution of the individuals may approach the optimal solution.

The most common operations of a genetic algorithm are designed to produce efficient solutions for the target problem [15]. These primary operations include:

(a) *Reproduction*: This operation assigns the reproduction probability to each individual based on the output of the fitness function. The individual with a higher ranking is given a greater probability for reproduction. As a result, the fitter individuals are allowed a better survival chance from one generation to the next.

(b) *Crossover*: This operation is used to produce the descendants that make up the next generation. This operation involves the following crossbreeding procedures:

(i) Randomly select two individuals as a couple from the parent generation.

(ii) Randomly select a position of the genes, corresponding to this couple, as the crossover point. Thus, each gene is divided into two parts.

(iii) Exchange the first parts of both genes corresponding to the couple.

(iv) Add the two resulted individuals to the next generation.

(c) *Mutation*: This operation picks a gene at random and changing its state according to the mutation probability. The purpose of the mutation operation is to maintain the diversity in a generation to prevent premature convergence to a local optimal solution. The mutation probability is given intuitively since there is no definite way to determine the mutation probability.

Upon completion of crossover processing and mutation operations, there will be an original parent population and a new offspring population. A fitness function should be devised to determine which of these parents and offsprings can be survived into the next generation. After performing the fitness function, these parents, and offsprings are filtered and a new generation is formed. These operations are iterated until the expected goal is achieved. Genetic algorithms guarantee high probability of improving the quality of the individuals over several generations according to the Schema Theorem [5].

3. Apply genetic algorithms to path testing

Path testing is one kind of software testing methodologies which searches the program domain for suitable test cases such that after executing the program with the test cases, a predefined path can be reached. Since a program may contain an infinite number of paths, it is only practical to select a specific subset path to perform path testing. Genetic algorithms could be applied to path testing if the target paths are clearly defined and an appropriate fitness function related to this goal is built.

In this paper, the selected target path is the goal for the algorithm to approach, and a test case is regarded as an individual of a generation. Test case generation for path testing consists of four basic steps:

(1) *Control flow graph construction*: In this step, the source program is transferred to a graph that represents the control flow of the program (e.g. Fig. 3 is the control flow graph of the program in Fig. 2). A program's control flow graph can be constructed manually or automatically with exiting tools. Each branch of the graph is denoted by a label and different branches correspond with different labels. Each branch of the tested program is instrumented with a procedure or statements that can generate a specific label as an output. The output label indicates which specific branch is executed. The example of original program and its instrumented version are shown in Figs. 2 and 4, respectively. In Fig. 4, the statement *strcat*(*path*,"*e*") is one of the instrumented statements. The instrumented program will generate two sets of output, one by its original function and the other the instrumented statements. The second output is a label string that represents an executed path. When loops exist in the path, the instrumentation reflects the loops with repeating sub-strings.

(2) *Target path selection*: In path testing, some paths might be very meaningful and need to be selected for testing. For example, the path "abc" in Fig. 2 is the most difficult path to be covered in random testing. So that it is selected as the target path in this example to show the ability of genetic algorithm.

(3) *Test case generation and execution*: This step is used to create test cases. After each unwanted path is executed, the algorithm automatically generates new test cases to execute new paths and leads the control flow to the target path. Finally, a suitable test case that executes along the target paths could be generated.

(4) *Test result evaluation*: This step is to execute the selected path and to determine whether the testing criteria or test oracles are satisfied.

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. The generated cases are fed to the program for execution. One test case will be exercised in one and only one correlated path. 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 output paths and the target path. Such

distances are used to determine which test cases should survive. After all the test cases in the present generation are executed, the new generation of test cases is generated by *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.

When applying genetic to the algorithms to the automation of software testing, variable values exist as patterns of bits. When encoding the individual, although the program inputs may be of different types and of complicated data structure, these inputs can be treated as a single, concatenated bit string denoted as $\{b_1, b_2, \ldots, b_n\}$. For example, the parameters of the function *funl*(*float f*,*int m*) can be treated as a 48-bit (6 byte) string. In this string, f occupies 32 bits (4 bytes) and m occupies 16 bits (2 bytes). Each input bit string is referred to as an individual of a generation, an individual is subdivided into genes, and any bit or sub-string in the string can be regarded as a gene. Normally, genetic algorithms operate on the bit strings or an individual of fixed size. The mutation is an operation in which a bit within a bit string is chosen randomly and then flipped from 1 to 0 and from 0 to 1.

3.1. To develop 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 [11]. 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 path may contain the same branches but in different sequences. The simple Hamming distance is no longer suitable.

One contribution of this paper is to extend the Hamming distance from the first-order to the nth order $(n > 1)$ to measure the distance between two path. Such extension is hereby named extended Hamming distance (EHD).

In the paragraph below, a fitness function is developed to measure the distance between two paths.

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

$$0 \leqslant N_{\alpha-\beta} \leqslant 1,$$

because $D_{\alpha-\beta} = |\alpha \oplus \beta| = |\alpha \cup \beta| - |\alpha \cap \beta| \leqslant |\alpha \cup \beta|$.

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

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

A *complete path* is a path whose initial node is the start node and whose final node is an exit node. If P is a control-flow diagram of a given program and Q is the set of all complete paths within P, then

$$Q = \{\boldsymbol{path}_i | \boldsymbol{path}_i \text{ is a } \textbf{complete path} \text{ within } P\}$$
$$= \{\boldsymbol{path}_1, \boldsymbol{path}_2, \ldots, \boldsymbol{path}_z\},$$

where $z =$ the number of complete paths in P,

$\boldsymbol{path}_i =$ the ith complete path in P, $\quad 1 \leqslant i \leqslant z$.

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

$S_1^2 = \{h | h \text{ is an ordered pair of cascaded branches of} \boldsymbol{path}_1\}$,

. . . . . .

$S_1^n = \{k | k \text{ is an ordered } n\text{\_tuple of cascaded branches of } \boldsymbol{path}_1,\ n \leqslant |S_1^1|\}$,

. . . . . .

$S_q^t = \{r | r \text{ is an ordered } t\text{\_tuple of cascaded branches of } \boldsymbol{path}_q,\ t \leqslant |S_q^1| \text{ and } 1 \leqslant q \leqslant z\}$,

. . . . . .

The first-order *distance* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is expressed as

$$D^1_{i-j} = |S^1_i \oplus S^1_j|.$$

The normalized first-order *distance* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is expressed as

$$N^1_{i-j} = \frac{D^1_{i-j}}{|S^1_i \cup S^1_j|}.$$

The first-order *similarity* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is defined as

$$M^1_{i-j} = 1 - N^1_{i-j}.$$

The second-order *distance* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is expressed as

$$D^2_{i-j} = |S^2_i \oplus S^2_j|.$$

The normalized second-order *distance* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is expressed as

$$N^2_{i-j} = \frac{D^2_{i-j}}{|S^2_i \cup S^2_j|}.$$

The second-order *similarity* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is defined as

$$M^2_{i-j} = 1 - N^2_{i-j}.$$

The mth ($m = 1..n$) order *distance* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is expressed as

$$D^m_{i-j} = |S^m_i \oplus S^m_j|.$$

The normalized mth order *distance* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is expressed as

$$N^m_{i-j} = \frac{D^m_{i-j}}{|S^m_i \cup S^m_j|}.$$

The mth order *similarity* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ is defined as

$$M^m_{i-j} = 1 - N^m_{i-j}.$$

The notation D^m_{i-j} is the mth order EHD between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$. The notation N^m_{i-j} is the mth order normalized extended Hamming distance (NEHD) between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$. The notation M^m_{i-j} is named the mth order *SIMILARITY* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ where $1 \leqq m \leqq n$. Note that $M^m_{i-j} = 0$ (or $N^m_{i-j} = 1$) if $S^m_i \cap S^m_j = \phi$. It means that $\boldsymbol{path}_i$ and $\boldsymbol{path}_j$ have no common m_ tuple cascaded breaches. Larger N^n_{i-j} means greater *difference* between $\boldsymbol{path}_j$ and $\boldsymbol{path}_i$. Contrarily, larger M^n_{i-j} 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 $\{N^1_{i-j} = 1,\ N^2_{i-j} = 1, \ldots, N^n_{i-j} = 1\}$ or $\{M^1_{i-j} = 0, M^2_{i-j} = 0, \ldots, M^n_{i-j} = 0\}$, which is resulted from a worst test case. When $\boldsymbol{path}_j$ and $\boldsymbol{path}_i$ are identical, NEHD should take the form of $\{N^1_{i-j} = 0, N^2_{i-j} = 0, \ldots, N^n_{i-j} = 0\}$ or $\{M^1_{i-j} = 1, M^2_{i-j} = 1, \ldots, M^n_{i-j} = 1\}$, which is resulted from a perfect test case that forces the program to execute along the target path. Therefore, if NEHD is not in the

condition of $\{N^1_{i-j} = 0, N^2_{i-j}, = 0, \ldots, N^n_{i-j} = 0\}$, the fitness function *SIMILARITY* between $\boldsymbol{path}_j$ and $\boldsymbol{path}_i$ is defined as

$$SIMILARITY_{i-j} = M^1_{i-J} \times W_1 + M^2_{i-j} \times W_2 + M^3_{i-j} \times W_3 + \cdots + M^n_{i-j} \times W_n,$$

where W's are the weighing factor of fitness and $W_1 < W_2 < W_3 < \cdots < W_n$.

$n = |S^1_i|$ if $\boldsymbol{path}_i$ is defined as the target path.

$n = |S^1_j|$ if $\boldsymbol{path}_j$ is otherwise. Since, if the target path is constructed by n branched, the values of $M^{n+1}_{i-j}, M^{n+2}_{i-j}, \ldots$ should be zero and are insignificant in *similarity* comparison.

$SIMILARITY_{i-j}$ determines the fitness between current executed $\boldsymbol{path}_j$ and the target $\boldsymbol{path}_i$. The greater $SIMILARITY_{i-j}$ leads to the better fitness. The higher-order *similarity* is more significant than its lower-order counterpart. The highest-ordered *similarity* between $\boldsymbol{path}_i$ and $\boldsymbol{path}_j(M^n_{i-j})$ 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 kth order.

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

$$\begin{aligned} W_1 &= 1, \\ W_2 &= W_1 \times |S^1_i|, \\ W_3 &= W_2 \times |S^2_i|, \\ &\cdots\cdots \\ W_n &= W_{n-1'} \times |S^{n-1}_i|. \end{aligned}$$

The distance between $\boldsymbol{path}_i$ (the target path) and $\boldsymbol{path}_j$ is larger than the distance between $\boldsymbol{path}_i$ and $\boldsymbol{path}_k$ if $SIMILARITY_{i-k} > SIMILARITY_{i-j}$. Therefore, $\boldsymbol{path}_k$ is closer to the target than $\boldsymbol{path}_j$ 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.

3.2. Calculating the distance similarity

An example of calculating the distance as well as the similarity between the target path and two executed paths is illustrated below.

In Fig. 1, each branch of the control flow graph is denoted with a label. A sequence of branches that construct a path can be denoted with a string of labels. Table 1 illustrates a minimal set of paths that satisfies the all-path

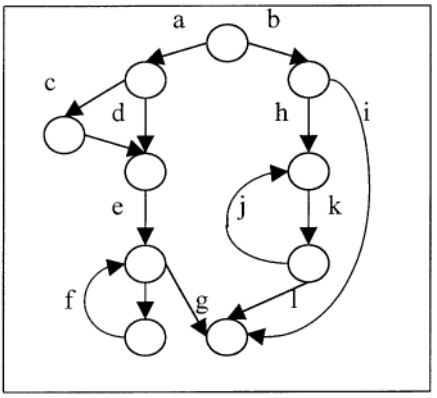


Fig. 1. An example of control flow diagram.

Table 1
The all-path list of the control flow diagram in Fig. 1

$\boldsymbol{path}_1 = \langle b, i\rangle$
$\boldsymbol{path}_2 = \langle b, h, k, l\rangle$
$\boldsymbol{path}_{3\_x} = \langle b, h, \langle k, j\rangle^x, l\rangle$
$\boldsymbol{path}_4 = \langle a, d, e, g\rangle$
$\boldsymbol{path}_5 = \langle a, c, e, g\rangle$
$\boldsymbol{path}_{6\_x} = \langle a, d, e, \langle f\rangle^x, g\rangle$
$\boldsymbol{path}_{7\_x} = \langle a, c, e, \langle f\rangle^x, g\rangle$

strategy in Fig. 1. The number x in the superscript represents the number of loops. In the following illustrations, $\boldsymbol{path}_{7\_2}$ is selected as the target to test. The path $\boldsymbol{path}_{7\_2}$ has two loops in branch 'f' and is one of $\boldsymbol{path}_{7\_x}$. To determine which path is more similar to the target path, two paths ($\boldsymbol{path}_5, \boldsymbol{path}_4$) are selected for demonstration.

3.2.1. The distance and the similarity between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$

The following is a calculation of the distance and the similarity between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$. If $\boldsymbol{path}_{7\_2}$ is the target path (where $\boldsymbol{path}_{7\_2} = \langle a, c, e, \langle f\rangle^2, g\rangle, |S^1_{7\_2} = 5)$, then the distance between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5 (= \langle a, c, e, g\rangle)$ is illustrated below.

(1) The *calculation of* $D^1_{5-7\_2}$, $N^1_{5-7\_2}$ *and* $M^1_{5-7\_2}$

Since

(a) the set of distinct branches in $\boldsymbol{path}_5$ is S^1_5 (where $S^1_5 = \{a, c, e, g\}$),

(b) the set of distinct branches in both $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^1_5 \cup S^1_{7\_2}$ (where $S^1_5 \cup S^1_{7\_2} = \{a, c, e, f, g\}$, and

(c) the set of the first-order *symmetric difference* between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^1_5 \oplus S^1_{7\_2}$ (where $S^1_5 \oplus S^1_{7\_2} = \{f\}$, thus they lead to

$$D^1_{5-7\_2} = |S^1_5 \oplus S^1_{7\_2}| = 1,$$

$$N^1_{5-7\_2} = \frac{D^1_{5-7\_2}}{|S^1_5 \cup S^1_{7\_2}|} = 1/5, \text{ and}$$

$$M^1_{5-7\_2} = 1 - N^1_{5-7\_2} = 4/5.$$

(2) *The calculation of* $D^2_{5-7\_2}$, $N^2_{5-7\_2}$ *and* $M^2_{5-7\_2}$
Since
(a) the set of distinct ordered pairs of consecutive branches in $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ are $S^2_{7\_2}$ and S^2_5 (where $S^2_{7\_2} = \{\langle a,c\rangle, \langle c,e\rangle, \langle e,f\rangle, \langle f,f\rangle, \langle f,g\rangle\}$, $|S^2_{7\_2}| = 5$) and $S^2_5 = \{\langle a,c\rangle, \langle c,e\rangle, \langle e,g\rangle\}$),
(b) the set of distinct ordered pairs of consecutive branches in both $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^2_5 \cup S^2_{7\_2}$ (where $S^2_5 \cup S^2_{7\_2} = \{\langle a,c\rangle, \langle c,e\rangle, \langle e,f\rangle, \langle f,f\rangle, \langle f,g\rangle, \langle e,g\rangle\}$) and
(c) the set of the second-order *symmetric difference* between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^2_5 \oplus S^2_{7\_2}$ (where $S^2_5 \oplus S^2_{7\_2} = \{\langle e,f\rangle, \langle f,f\rangle, \langle f,g\rangle, \langle e,g\rangle\}$), thus they lead to

$$D^2_{5-7\_2} = |S^2_5 \oplus S^2_{7\_2}| = 4,$$

$$N^2_{5-7\_2} = \frac{D^2_{5-7\_2}}{|S^2_5 \cup S^2_{7\_2}|} = 4/6, \text{ and}$$

$$M^2_{5-7\_2} = 1 - N^2_{5-7\_2} = 2/6.$$

(3) *The calculation of* $D^3_{5-7\_2}$, $N^3_{5-7\_2}$ *and* $M^3_{5-7\_2}$
Since
(a) the set of distinct ordered 3_ tuple of consecutive branches in $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ are $S^3_{7\_2}$ and S^3_5 (where $S^3_{7\_2} = \{\langle a,c,e\rangle, \langle c,e,f\rangle, \langle e,f,f\rangle, \langle f,f,g\rangle\}$, $|S^3_{7\_2}| = 4$, and $S^3_5 = \{\langle a,c,e\rangle, \langle c,e,g\rangle\}$),
(b) the set of distinct ordered 3_tuple of consecutive branches in both $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^3_5 \cup S^3_{7\_2}$ (where $S^3_5 \cup S^3_{7\_2} = \{\langle a,c,e\rangle, \langle c,e,f\rangle, \langle e,f,f\rangle, \langle f,f,g\rangle, \langle c,e,g\rangle\}$), and
(c) the set of the third-order *symmetric difference* between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^3_5 \oplus S^3_{7\_2}$ (where $S^3_5 \oplus S^3_{7\_2} = \{\langle c,e,f\rangle, \langle e,f,f\rangle, \langle f,f,g\rangle, \langle c,e,g\rangle\}$), thus they lead to,

$$D^3_{5-7\_2} = |S^3_5 \oplus S^3_{7\_2}| = 4,$$

$$N^3_{5-7\_2} = \frac{D^3_{5-7\_2}}{|S^3_5 \cup S^3_{7\_2}|} = 4/5, \text{ and}$$

$$M^3_{5-7\_2} = 1 - N^3_{5-7\_2} = 1/5.$$

(4) *The calculation of* $D^4_{5-7\_2}$, $N^4_{5-7\_2}$ *and* $M^4_{5-7\_2}$
Since
(a) the set of distinctly ordered 4_tuple of consecutive branches in $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ are $S^4_{7\_2}$ and S^4_5 (where $S^4_{7\_2} = \{\langle a,c,e,f\rangle, \langle c,e,f,f\rangle, \langle e,f,f,g\rangle\}$, and $S^4_5 = \{\langle a,d,f,i\rangle\}$),
(b) the set of distinct ordered 4_tuple of consecutive branches in both $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^4_5 \cup S^4_{7\_2}$ (where $S^4_5 \cup S^4_{7\_2} = \{\langle a,c,e,f\rangle, \langle c,e,f,f\rangle, \langle e,f,f,g\rangle, \langle a,d,f,i\rangle\}$)
(c) the set of the fourth-order *symmetric difference* between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is $S^4_5 \oplus S^4_{7\_2}$ (where $S^4_5 \oplus S^4_{7\_2} = \{\langle a,c,e,f\rangle, \langle c,e,f,f\rangle, \langle e,f,f,g\rangle, \langle a,d,f,i\rangle\}$), thus they lead to

$$D^4_{5-7\_2} = |S^4_5 \oplus S^4_{7\_2}| = 4,$$
$$N^4_{5-7\_2} = \frac{D^4_{5-7\_2}}{|S^4_5 \cup S^4_{7\_2}|} = 1, \text{ and}$$
$$M^4_{5-7\_2} = 1 - N^4_{5-7\_2} = 0.$$

Since $M^4_{5-7\_2} = 0$, the $D^5_{5-7\_2}$, $N^5_{5-7\_2}$, $M^5_{5-7\_2}$ and the higher-order of distances need not to be figured out.

The NEHD between $\boldsymbol{path}_5$ and $\boldsymbol{path}_{7\_2}$ is NEHD_{5-7_2}, i.e. $\{1/5, 4/6, 4/5, 1\}$. Since the fourth-order NEHD between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ is 1, the wth order (where $w \geqslant 4$). NEHD between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_5$ must also be 1. Therefore, $\text{NEHD}_{5-7\_2} = \{1/5, 4/6, 4/5, 1\} = \{1/5, 4/6, 4/5, 1, 1, 1, \ldots\}$.

3.2.2. The distance and similarity between $\boldsymbol{path}_{7\_2}$ *and* $\boldsymbol{path}_4$

Using the same method as in Section 3.2.1, the values of $D^1_{4-7\_2}$, $N^1_{4-7\_2}$, $M^1_{4-7\_2}$, $D^2_{4-7\_2}$, $N^2_{4-7\_2}$ and $M^2_{4-7\_2}$ lead to

$$D^1_{4-7\_2} = 3,$$
$$N^1_{4-7\_2} = 3/6,$$
$$M^1_{4-7\_2} = 3/6,$$
$$D^2_{4-7\_2} = 8,$$
$$N^2_{4-7\_2} = 8/8 = 1, \text{ and}$$
$$M^2_{4-7\_2} = 0.$$

Since the second-order distance between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_4$ is 1, the wth order (where $w > 2$) distance between $\boldsymbol{path}_{7\_2}$ and $\boldsymbol{path}_4$ is also 1. The NEHD between $\boldsymbol{path}_4$ and $\boldsymbol{path}_{7\_2}$ is NEDH_{4-7_2} (where $\text{NEHD}_{4-7\_2} = \{3/6, 1\} = \{3/6, 1, 1, 1, 1\} = \{3/6, 1, 1, 1, 1, \ldots\}$).

To compare $\text{NEHD}_{4-7\_2}(= \{3/6, 1, 1, 1, 1\})$ with $\text{NEHD}_{5-7\_2}(= \{1/6, 4/6, 4/5, 1\})$, it is clear that the distance between $\boldsymbol{path}_4$ and $\boldsymbol{path}_{7\_2}$ is greater than the distance between $\boldsymbol{path}_5$ and $\boldsymbol{path}_{7\_2}$, because the higher the order is, the

more significant the weighting is (where $N^4_{4-7\_2} = N^4_{5-7\_2}, N^3_{4-7\_2} > N^3_{5-7\_2}, N^2_{4-7\_2} > N^2_{5-7\_2}$, and $N^1_{4-7\_2} > N^1_{5-7\_2}$).

At the same time, the weighing factors are obtained below:

$$W_1 = 1,$$
$$W_2 = W_1 \times |S^1_{7\_2}| = 1 \times 5 = 5,$$
$$W_3 = W_2 \times |S^2_{7\_2}| = 5 \times 5 = 25,$$
$$W_4 = W_3 \times |S^3_{7\_2}| = 25 \times 4 = 100.$$

The fitness values of $\boldsymbol{path}_4$ and $\boldsymbol{path}_5$ can be derived below:

$$SIMILARITY_{4-7\_2} = M^1_{4-7\_2} \times W_1 + M^2_{4-7\_2} \times W_2 = (1 - 3/6) \times 1 + (1 - 1) \times 5 = 0.5,$$

$$SIMILARITY_{5-7\_2} = M^1_{5-7\_2} \times W_1 + M^2_{5-7\_2} \times W_2 + M^3_{5-7\_2} \times W_3 + M^4_{5-7\_2} \times W_4 = (1 - 1/6) \times 1 + (1 - 4/6) \times 5 + (1 - 4/5) \times 25 + (1 - 1) \times 100 = 7.5,$$

which shows that $\boldsymbol{path}_5$ is closer to the target path ($\boldsymbol{path}_{7\_2}$) than $\boldsymbol{path}_4$ is or that $\boldsymbol{path}_5$ is more similar to the target path ($\boldsymbol{path}_{7\_2}$) than $\boldsymbol{path}_4$ is.

3.3. The experiment results

In the general path testing experiment, the four basic steps are processed below.

(1) *Control flow graph construction*: The tested program (Fig. 2 Triangle classifier) determines what kind of triangle can be formed by any three input lengths. The program's control flow diagram, which contains four paths, is shown in Fig. 3. After the instrumentation, the tested program is transformed

```
int TriangleA(int a, int b, int c)
{   int Triangle = 0;                                                    /*b1*/
    int c1,c2,c3,c4,c5,c6,c7,c8;                                         /*b1*/
    if (((a+b>c)&&(b+c>a)&&(c-a>b))&&((a>0)&&(b>0)&&(c>0)))              /*b1*/
    {   if (( a != b) && ( b != c )&& ( c != a ))                        /*b2*/
            Triangle = 1;                                        /* Scalene b3*/
        else
        {   if (((a == b) && (b != c))|| ((b == c) && (c != a))|| ((c == a) && (a != b)))   /*b4*/
                Triangle = 2;                                    /* Isosceles b5*/
            else                                                         /*b6*/
                Triangle = 3;                                    /* Equilateral  b6*/
        }
    }
    return ( Triangle );                                                 /*b7*/
}
```

Fig. 2. Triangle classifier [12,13].

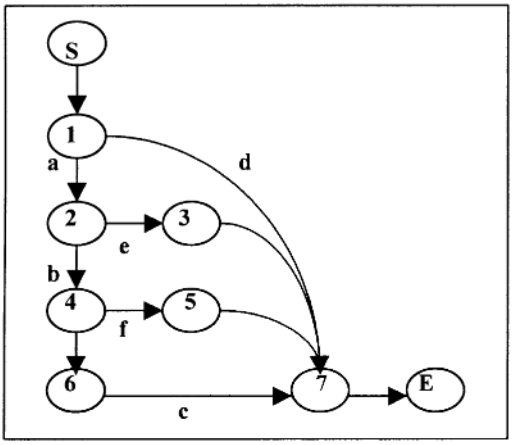


Fig. 3. The control flow diagram of the program shown in Fig. 2.

to the one shown in Fig. 4, In the implemented 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.

The comments $/* \ b_1 \ */ \cdots /* \ b_7 \ */$) denote the program blocks 1 to 7, respectively. In Fig. 3, the nodes 1 to 7 indicate the control flow locations of these blocks.

(2) *Target path selection*: The path "*abc*" in Fig. 2 is the most difficult path to be covered in random testing. The program in Fig. 2 has three integers as input parameters. While the three parameters are positive and equal, the path "*abc*" will 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.

(3) *Test case generation and execution*: According to the genetic algorithms, an experimental tool for automatically generating test cases to test a specific path is developed. The tool divided total 48-bit input string 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. Traditionally, the mutation probability is set to the reciprocal of the length of the bit string [11]. Hence, the mutation probability is set to 1/48.

In this experiment, the first generation of test cases was chosen by the tool from the tested program 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

```
# include <stdio.h>
# 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 b6*/
            }
        }
    }
    else
    {   strcat(path,"d");                                  /* instrumentation ( not a triangle) */
    }
   return ( Triangle );                                                   /*b7*/
}
```

Fig. 4. The example of the program in Fig. 2 after being instrumented.

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 was mostly group in the path $\langle d \rangle$. In the subsequent generations, the execution of generated test cases gathered in the path $\langle a, e \rangle$. According to the experiments, about 52.5% and 47.5% of the first generation test cases, which were generated at random, executed the path $\langle d \rangle$ and the path $\langle a, e \rangle$, 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 $\langle d \rangle$ and mostly gathered in the path $\langle a, e \rangle$. Afterward, the execution of generated test cases approached the path $\langle abf \rangle$ gradually. Finally, at least a test case reached the path $\langle abc \rangle$ and succeeded in generating the test case. After one hundred experiments, the results on Fig. 5 show that the target path was obtained within 10 100 test cases (i.e. 10 *generations* $\times$ 1000 *test cases in each generation* + *100 test cases in the 11th generation)* by average. While, based on the theory of probability, it will take random testing 2^{30} tests to reach the target.

Fig. 6 shows the evolution of test cases from the first generation to the 10th generation. These values were averaged from 100 experiments. Before the fifth

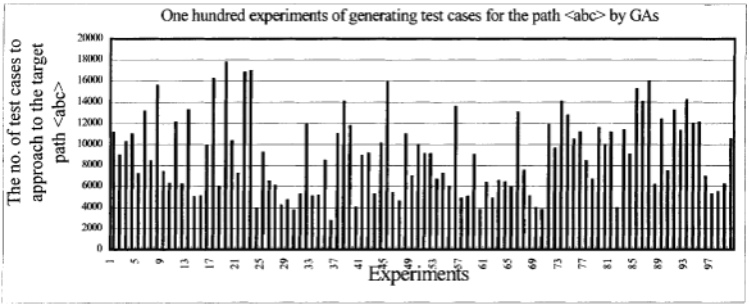
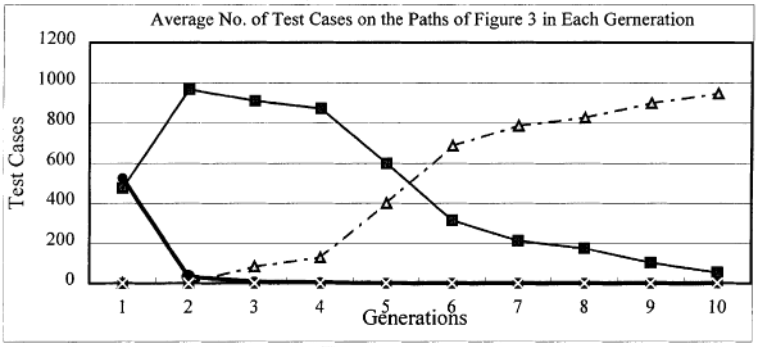


Fig. 5. On hundred experiments of generating test cases for executing the target path $\langle abc \rangle$ in Fig. 4 by using GAs.

generation, the algorithm decreased the number of test cases on the path $\langle d \rangle$ and increased the number of test cases on the path $\langle ae \rangle$. After the fifth generation, the number of test cases on the path $\langle ae \rangle$ was decreased and the number of test cases on the path $\langle abf \rangle$ increased speedily.

(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.



Generations	•Path <d>	áPath <ae>	Path <abf>	⊠Path <abc>	The No. of test cases
1	525	475	0	0	1000
2	33	966	1	0	1000
3	8	910	82	0	1000
4	4	870	126	0	1000
5	1	600	399	0	1000
6	0	314	686	0	1000
7	0	214	786	0	1000
8	0	175	825	0	1000
9	0	102	898	0	1000
10	0	53	946	1	1000

Fig. 6. The average number of test cases on the paths of Fig. 3 in each generation.

Since there is no guarantee that the algorithm can find the target within a 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 10 000 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. As a 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 automatically generate test cases for path testing. The greatest merit of using the genetic algorithm in program testing is its simplicity. Each iteration of the genetic algorithms generates a generation of individuals. In practice, the computation time cannot be infinite, so that the iterations in the algorithm should be limited. Within the limited generations, solutions derived by genetic algorithms may be trapped around a local optimum and, as a result, fail to locate the required global optimum. Although the tested cases generated by such algorithms may be trapped around unwanted paths and fail to locate the required paths, since the test cases of the first generation are normally distributed over the domain of the tested program, the probability of being trapped is very low.

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. Comparing with random testing, using extended Hamming distance to derive *SIMILARITY* (a fitness function) is a very useful approach for path testing. Although a program may contain infeasible paths where no method, including genetic algorithms, can generate suitable test cases, this paper shows that genetic algorithms are able to meaningfully reduce time required for lengthy testing by generating test cases for path testing in an automatic way.

References

[1] J.F. Benders, Partitioning procedures for solving mixed variables programming problems, Numer. Math. 4 (1962) 238.
[2] A. Bertolini, An overview of automated software testing, J. System Software 15 (1991) 133–138.
[3] C.J. Burgess, The automated generation of test cases for compilers, J. Software Test. Verif. Reliab. 4 (2) (1994) 81–100.

[4] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Trans. Software Engrg. and Methodology 5 (1) (1996) 63–86.
[5] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989.
[6] B.F. Jone, D.E., H.-H. Sthamer, A strategy for using genetic algorithms to automate branch and fault-based tesitng, The Computer Journal 41 (1998) 98–107.
[7] J.H. Holland, Genetic algorithms and the optimal allocation of trials, SIAM J. Comput. (1973) 89–104.
[8] J.H. Holland, Adaptation in Nature and Artificial Systems, Addison-Wesley, Reading, MA, 1975.
[9] D.C. Ince, The automatic generation of test data, Computer J. 30 (1) (1987) 63–69.
[10] B.F. Jones, H.-H. Sthamer, X. Yang, D.E. Eyres, The automatic generation of software test data sets using adaptive search techniques, in: Proceedings of the Third International Conference on Software Quality Management, Seville, 1995, pp. 435–444 (BCS/CMP).
[11] B.F. Jones, H.-H. Sthamer, D.E. Eyres, Automatic structural testing using genetic algorithms, Software Engrg. J. 9 (1996) 299–306.
[12] P.N. Lee, Correspondent computing, in: Proceeding of the ACM 1988 Computer Science Conference, Atlanta, Georgia, 1988, pp. 12–19.
[13] G. Myers, The Art of Software Testing, Wiley, New York, 1979.
[14] M.A. Ould, Testing-a challenge to method and tool developers, Software Engrg. J. 6 (2) (1991) 59–66.
[15] G. Syswerda, Uniform crossover in genetic algorithms, in: Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 2–9.
[16] J. Wegener, H. Sthmer, B.F. Jone, D.E. Eyres, Testint real-time system using genetic algorithms, Software Quality J. 6 (1997) 127–153.
[17] E. Alba, J.F. Aldana, J.M. Troya, Genetic algorithms as heuristics for optimizing ANN design, in: Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, Innsbruck, Austria, 1993, pp. 682–690.
[18] S.A. Harp, T. Samad, A. Guha, Towards the genetic synthesis of neural networks, in: Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 360–369.
[19] H. Kitano, Empirical studies on the speed of convergence of neural network training using genetic algorithms, in: Proceedings of the Eighth National Conference on Artificial Intelligence, 1990, pp. 789–795.
[20] P. Robbins, A. Soper, K. Rennolls, Use of genetic algorithms for optimal topology determination in back propagation neural networks, in: Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, Innsbruck, Austria, 1993, pp. 726–729.
[21] W. Schiffmann, M. Joost, R. Werner, Application of genetic algorithms to the construction of topologies for multi-layer perceptions, in: Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, Innsbruck, Austria, 1993, pp. 675–682.