Basis Test Paths Generation Using Genetic Algorithm

Ahmed S. Ghiduk
Dept. of Computer Science
College of Computers and Information Technology
Taif University, Taif, Saudi Arabia
asaghiduk@tu.edu.sa

O. Said
Dept. of Computer Science
College of Computers and Information Technology
Taif University, Taif, Saudi Arabia
o.saeed@tu.edu.sa

Sultan Aljahdali
Dept. of Computer Science
College of Computers and Information Technology
Taif University, Taif, Saudi Arabia
aljahdali@tu.edu.sa

Abstract—**One of the key problems in path testing is building a path through specified set of stalemates particularly which contain loops. Traditional genetic algorithm has been successfully used in software testing activities such as finding test data, selecting test cases and test cases prioritization. In this paper, we introduce a new variable length genetic algorithm. Based on the new algorithm, we present a new strategy for automatically generating a set of basis test paths which can be used as testing paths in any basis path testing technique. We define all elements of genetic algorithm such as chromosome representation, crossover, mutation, and fitness function to be compatible with path generation. In addition, we present a case study to show the efficiency of our strategy.**

Keywords-Genetic Algorithm; Basis Path Testing; Path Generation.

I. INTRODUCTION

Structural testing requires the execution of a set of test paths in the program under testing. Generating this set of paths is a critical task, and its automation is strongly desirable for easing the testing process. This task can influence the efficacy and cost of testing activity.

Basis path testing is one of the very powerful structural testing criteria. Basis path testing requires number of test paths (basis set) equals to the cyclomatic complexity of program [1] such that every path is an independent path and all edges in the control-flow graph (CFG) are covered by all paths in the basis set. In addition, a path that is not contained in the basis set can be constructed by a linear combination of paths in this set.

Some path generation methods have been introduced so far. Bertolino and Marre [2] provided a path generation method by using a reduced CFG. Although all the statements and branches can be covered by the set of paths, it cannot be assured that the set of paths generated by this method is a basis set of paths. Pool [3] discussed a basis set of paths generation method on the depth first search in CFG. It uses a recursive search in the CFG. As Pool's method, the loop is not taken into account. In addition, this method did not consider how to choose the successor of multiple-successors node in a CFG to build a basis path during the construction of the basis set of paths. Guangmei et al. [4] presented an automatic generation method of basis set of paths which is built by searching the CFG by depth-first searching method. In order to avoid that the algorithm will never stop, and for reducing the searching procedure, the sub-path from the multi-indegree nodes to the end node of a program and the sub-path that contains a loop are recorded during the construction of a basis path. This method did not handle infeasible paths. Yan and Zhang [5] presented a method for generating a finite set F of feasible paths which satisfies the basis path coverage criterion. Then, they found a minimal subset S of set F such that S satisfies the test coverage criterion. The first step should check the feasibility of all paths and feasibility checking is quite time-consuming. Zhonglin and Lingxia [6] and Qingfeng and Xiao [7] use cyclomatic complexity in generating a set of linearly independent paths. Many basis paths are infeasible because of data dependences exist in variables involved in decision node. They combine the baseline method with the dependence relationship to avoid selecting infeasible paths. These methods didn't handle loops.

Search-based testing techniques especially traditional genetic algorithms have been successfully used in software testing activities such as finding test data [8, 9]. Bint and Site discussed a path generation method based on genetic algorithm [10]. The defect of this method is that the generated set of paths cannot cover all edges of the CFG because the loop operation is removed. In addition, this method cannot generate a basis set of test paths.

In this paper, we introduce a new variable length genetic algorithm. Based on the proposed algorithm, we introduce a new technique for automatically generating a set of basis test paths. We define all elements of the new genetic algorithm such as representation, crossover, mutation, and fitness function to be compatible with path generation process. We will introduce how the proposed technique can handle the loop and infeasible paths. In addition, we will present a case study to show the efficiency of our new strategy.

The rest of the paper is organized as follows. Section 2 introduces the problem formulation. Section 3 gives some definitions. Section 4 presents our proposed strategy. Section 5 introduces a case study of the proposed method. Section 6 concludes the paper.

II. BASIS PATH TESTING

Structural testing generally requires the execution of a set Q of paths in the program under testing. Determining Q is a very important and critical task, and its automation is strongly

desirable for easing the testing strategy. This task can influence on the efficacy and on the testing effort and costs.

Thomas McCabe came up with the idea of using a vector space to carry out path testing [1]. A vector space is a set of elements along with certain operations that can be performed upon these elements. What makes vector spaces an attractive proposition to testers is that they contain a basis. The basis of a vector space contains a set of vectors that are independent of one another, and have a spanning property; this means that everything within the vector space can be expressed in terms of the elements within the basis. What McCabe noticed was that if a basis could be provided for a program graph, this basis could be subjected to rigorous testing; if proven to be without fault, it could be assumed that those paths expressed in terms of that basis are also correct. The method devised by McCabe [1] to carry out basis path testing has the following four steps:

- Compute the program graph.
- Calculate the cyclomatic complexity. In graph theory, the cyclomatic number is defined as $C(G) = m – n + q$, where m is number of edges in the graph G, n is number of nodes, and q is number of strongly connected components. For a program that has a single entry and exit point, $q = 1$ [6]. In basis path testing, the cyclomatic complexity should be the upper limit for the number of basis paths [11].
- Select a basis set of paths.
- Generate test cases for each of these paths.

Independent path: An independent path is a path of a program, and there is at least one edge in the control-flow graph (CFG) appeared in this path that has never appeared in other paths.

Basis set of path and basis path: A basis set of path is a set of paths, and every path in this set should satisfy the next three conditions:

- Every path should be an independent path.
- All edges in a CFG should be covered by all paths in the basis set.
- Every path that does not contain in a basis set of path can be constructed by the linear operation among paths in this set.

The path contained in a basis set is called a basis path. The problem of this work is defining a new genetic algorithm and using it for generating a set Q of basis paths.

III. BASIC CONCEPTS

We introduce here some basic concepts that will be used through this work.

A. Genetic Algorithms Principles

The basic concepts of genetic algorithms (GAs) were developed by Holland [12]. The GAs start by creating an initial population of individuals, each represented by randomly generated binary string called chromosome. The basic algorithm of GAs, where P(t) is the population strings at generation number t, is as follows:

1. initialize P(t);
2. evaluate P(t);
3. **while** termination condition not satisfied **do**
4. select P(t+1) from P(t);
5. recombine P(t+1);
6. evaluate P(t+1);
7. t = t + 1;
8. **end while**

In the evaluation step, the fitness of each individual is determined. The selection step is used to find pairs of individuals that will be combined in some way to contribute to the next generation. The process of crossover involves two chromosomes swapping chunks of data. Mutation introduces slight changes into a small proportion of population and is representative of an evolutionary step. The above algorithm will iterate until the population has evolved to form a solution to the problem, or until a termination condition is satisfied.

B. Program representation

A program's structure is analyzed on the program flow-graph, i.e., an annotated directed graph which represents graphically the information needed to select the test cases.

A control-flow graph (*CFG*) is a directed graph $G=(N,E)$, with two distinguished nodes —a unique *entry* (n_0) and a unique *exit* (n_k). N is a set of nodes, where each node represents a statement, and E is a set of directed edges, where a directed edge $e = (n, m)$ is an ordered pair of adjacent nodes, called *tail* and *head* of e, respectively.

A dd-graph (*DDG*) is a digraph $G =(N, E)$, where N is a set of nodes and E is a set of edges, with two distinguished edges e_0, e_k (the unique *entry* edge and the unique *exit* edge, respectively), such that any other edge in E is reached by e_0 and reaches e_k, and such that for each node $n \in N$, $n \neq tail(e_0)$, $n \neq head(e_k)$, $(indegree(n) + outdegree(n)) > 2$, while $indegree(tail(e_0)) = 0$ and $outdegree(tail(e_0)) = 1$, $indegree(head(e_k)) = 1$ and $outdegree(head(e_k)) = 0$. An edge e in a dd-graph *DDG* is an ordered pair of adjacent nodes, called tail and head of e, respectively (i.e., $e = (tail(e), head(e))$. A path p of length l in a dd-graph *DDG* is a sequence $p= e_0, e_1, e_2, \ldots, e_l$, where $tail(e_{i+1}) = head(e_i)$ for $i = 1, 2, \ldots, l-1$. A path p is simple if all its nodes, except possibly the first and last, are distinct. A complete path in a dd-graph *DDG* is a path from the *entry* node to the *exit* node of *DDG*. Given a path $p= e_0, e_1, e_2, \ldots, e_l$, then a path $p'= e_i, \ldots, e_j$ from e_i *to* e_j, with $1 \leq i \leq j \leq l$, is called a subpath of p. Antonia Bertolino and Martina Marre provided a procedure to construct the dd-graph by reducing the control flow graph of the program [2]. Figure 1(a) gives an example program, Figure 1(b) shows its control-flow graph, and Figure 1(c) provides its ddgraph.

In our proposed system, we will represent the program under test as a dd-graph according to the above definitions.

IV. OUR PROPOSED STRATEGY

A. Our new genetic algorithm

In this section, we present our proposed GA for automatic generation of basis test paths for the tested software, which uses a new fitness function to evaluate the generated test path. This fitness function depends on the concepts of the number of the adjacent edges in the ddgraph of the software under test. The algorithm searches for test paths that satisfy the three conditions of the basis set of path (see section II). The major components of this GA are discussed below.

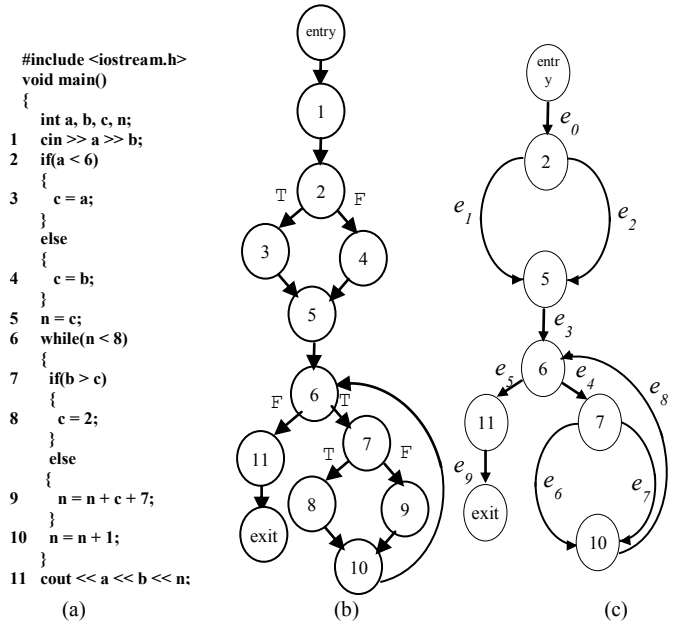

Figure 1. An example program (a), its control-flow graph (b) and its dd-graph (c)

1) The search space of our algorithm:

The search space is the set of all solutions among which the desired solution resides. Each point in the search space represents one possible solution. Suppose that DDG is a ddgraph of tested program. The search space of our new genetic algorithm is D, where

$$D = \{\forall e | e \in DDG \text{ and } e \text{ is reached by entry and reaches exit}\}.$$

In other word, the input domain of our algorithm is the set of all edges of the ddgraph of the program under test. For example the search space of the example program in Figure 1 is the set of edges $D = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$.

2) Encoding

Encoding is a process of representing individual genes. The process can be performed using bits, numbers, trees, arrays, lists or any other objects. The encoding depends mainly on solving the problem. For example, one can encode directly real or integer numbers. On the other hand, it is necessary to develop new genetic operator's specific to the problem.

The proposed GA uses a vector of integer (decimal) numbers as a chromosome to represent the edges in the ddgraph of the program under test. The length of the vector depends on the length of the required basis path. Each edge in the ddagrph is represented by its index in the chromosome. In addition, each cell (value) in the vector (chromosome) is mapped into its corresponding edge using the map:

$$M: i \in chromosome \rightarrow e_i \in ddgraph.$$

For example, suppose the program in Figure 1. Let us consider an example chromosome: 0, 1, 3, 5, 9. So, the given chromosome corresponds to the path $p = \{e_0, e_1, e_3, e_5, e_9\}$.

3) Initial population

As mentioned above, each chromosome is represented by a vector of decimal numbers. We randomly generate PS integer vectors of length 2 to represent the initial population, where PS is the population size. The appropriate value of *PS* is experimentally determined. Each individual in the initial population contains two edges the entry and the exit. For the example in Figure 1, all individuals have the form 0, 9 (e_0, e_9). The minimum length of the chromosome is two and the maximum length equals the number of ddgraph' edges.

4) Evaluation function

The algorithm can use any fitness function to evaluate the generated test path. The fitness function which depends on the concepts of the probability of the adjacent edges in the path is suitable for path generation. The algorithm uses this fitness function to evaluate each test path.

The fitness value $ft(v_i)$ for each chromosome c_i $(i = 1, \ldots, PS)$ is calculated as follows:

$$\mathbf{ft(c_i) = \sum_{j=1}^{d} w_j} \qquad \mathbf{(1)}$$

where, d is the number of adjacent edges in the chromosome i, w_j is the weight (probability) of the edge j in the chromosome i, $w_j = \frac{1}{L}$, where L the length of the chromosome (the number of edges in the chromosome).

The fitness value is the only feedback from the problem for the GA. A test case that is represented by the chromosome vi is optimal if its fitness value $ft(vi) = 1$.

Consider Figure 1(c), suppose 0, 1, 9 and 0, 2, 3, 9 are two chromosomes. The first chromosome contains three genes or edges ($L=3$) while the second contains four genes ($L=4$). The probability of each gene in the first chromosome is $\frac{1}{3}$ and $d=2$ while the probability of each gene in the second chromosome is $\frac{1}{4}$ and $d=3$. According to "(1)", the fitness values of the first and second chromosomes are $\frac{2}{3}$ and $\frac{3}{4}$, respectively.

5) Selection

After computing the fitness of each test path in the current population, the algorithm selects test paths from all the members of the current population that will be parents of the new population. In the selection process, the GA uses the roulette wheel method [13]. This method is described below.

For the selection of a new population with respect to the probability distribution based on fitness values, a roulette wheel with slots sized according to fitness is used. Such roulette wheel is constructed as follows:

- Calculate the fitness value $ft(v_i)$ for each chromosome v_i $(i = 1, \ldots, PS)$.
- Find the total fitness of the population $F = \sum_{i=1}^{PS} ft(v_i)$.
- Calculate the relative fitness value *rft* for each chromosome $rft(v_i) = \frac{ft(v_i)}{F}$.

- Calculate the cumulative fitness value *cft* for each chromosome $\mathrm{cft}(v_i) = \begin{cases} \mathrm{rft}(v_i) & i=1 \\ \mathrm{cft}(v_{i-1}) + \mathrm{rft}(v_i) & i=2,\ldots, ps \end{cases}$.

The selection process is based on spinning the roulette wheel PS times; each time we select a single chromosome for a new population in the following way:

- Generate a random number *r* from the range [0..1].
- If $r < cft(v_1)$ then select the first chromosome v_1; otherwise select the i-th chromosome v_i $(2 \le i \le PS)$ such that $\mathrm{cft}(v_i) \le r < \mathrm{cft}(v_{i+1})$.

Some chromosomes would be selected more than once.

6) Reproduction

In the Reproduction phase, we use three operators, crossover, mutation and breeding (a new operator we have proposed it), which are the key to the power of GAs. These operators create new individuals from the selected parents to form a new population.

Crossover: It operates at the individual level. During crossover, two parents (chromosomes) exchange sub-vector information (genetic material) at a random position in the chromosome to produce two new vectors (offspring). The objective here is to create better population over time by combining material from pairs of (fitter) members from the parent population. Crossover occurs according to a crossover probability. The probability of crossover PX gives us the expected number $PX \times PS$ of chromosomes, which undergo the crossover operation. We proceed in the following way:

For each chromosome in the parent population:

- Generate a random number *r* from the range [0..1];
- If $r < PX$ then select given chromosome for crossover.

Now, we mate selected chromosomes randomly: For each pair of coupled chromosomes we generate a random integer number *pos* from the range [2..*L*-1] (*L* is the number of edges in a chromosome). The number *pos* indicates the position of the crossing point. Two chromosomes $(b_1 \ldots b_{pos} b_{pos+1} \ldots b_n)$ and $(c_1 \ldots c_{pos} c_{pos+1} \ldots c_n)$ are replaced by a pair of their offspring $(b_1 \ldots b_{pos} c_{pos+1} \ldots c_n)$ and $(c_1 \ldots c_{pos} b_{pos+1} \ldots b_n)$.

Suppose that C_1=***(0, 1, 3, 5, 9)*** and C_2=*(0, 2, 3, 4, 9)* are two chromosomes and $pos = 3 \in [2..4]$, where $L = 5$. After applying the crossover operator the new chromosomes are B_1=***(0, 1, 3**, 4, 9)* and B_2=*(0, 2, 3, **5, 9)*.

Mutation: It is performed on a cell-by-cell basis. Mutation always operates after the crossover operator, and changes each cell with the pre-determined probability. The probability of mutation *PM*, gives us the expected number of mutated cells $PM \times L \times PS$. Every cell (in all chromosomes in the whole population) has an equal chance to undergo mutation. So we proceed in the following way:

For each chromosome in the current population and for each cell within the chromosome:

- Generate a random number *r* from the range [0..1];
- If $r < PM$ then mutate the cell by replacing the edge with another edge of its siblings (edges with the same parent are called siblings such as e_1 and e_2).

Suppose that C_1 = *(0, 1, 3, 5, 9)* is a chromosomes. The second cell which has the value 1 will mutate to the value 2. Therefore the new chromosome will be B_1 = *(0, **2**, 3, 5, 9)*.

Breeding: It is a new operator. We have developed this operator to enhance the chromosomes to get a complete path. It performed on a cell level. Breeding always operates after the mutation operator. We generate a random integer number *pos* from the range [2..*L*-1]. The number *pos* indicates the position of the breeding point. So, we proceed in the following way:

For each chromosome in the current population:

- Generate a random integer number pos from the range [2..L-1], L is the length of the chromosome. We can put *pos* = *d* (number of adjacent edges in the chromosome).
- Identify the edge at the position *pos* and randomly select one edge of its successors.
- Then, insert the successor edge at the position *pos+1* and increase the length of the chromosome by one.

Suppose that C_1 = *(0, 1, 3, 4, 9)* is a chromosomes and *pos = 4*. The edge at position 4 is e_4. The successors of e_4 are e_6 and e_7. We randomly select one of the successors and insert it at position 5. The new chromosome is B_1 = *(0, 1, 3, 4, **6**, 9)*.

7) Elitist

The elitist function enhances the current population by storing one copy of the best member of the previous population. If the best member of the current population is worse than the best member of the previous population it exchanges them, and the best member of the current population would replace the worst member of the current population. After that, it stores the best member of the current population.

8) The Stop Conditions

In the traditional GA approach the population would evolve until one individual from the whole set which represents the solution is found. In our case, this condition would correspond to finding groups of path achieving the basis test paths conditions. The evolution stops when a set of individuals has satisfied the required conditions and its fitness value $ft(v_i) = 1$. The solution is this set.

The algorithm will stop and the search will end in two cases. The first case when the generated test paths satisfy the conditions of the basis set of path. The second case when the number of generation reaches the maximum number of generation.

B. The Overall Algorithm of the Strategy

Our proposed GA-based strategy accepts as input the program to be tested, the control-flow graph (CFG) and the ddgraph (DDG) of the program, the entry (e_0) and the exit (e_k) edges of the ddgraph (DDG) and the set S_i of successors of each edge e_i. Also, it accepts the GA parameters: population size, maximum number of generations, and probabilities of the crossover and mutation operators. The algorithm produces a set of basis test paths.

The algorithm generates one basis test path at a time and repeats until the required paths are obtained or the maximum number of generations is exceeded. The overall algorithm is presented in Figure 2.

```
/* A GA algorithm to automatically generate set of basis test paths for a given program */
Input:
    The program to be tested P;
    The control-flow graph (CFG) and the ddgraph (DDG) of P.
    The entry e0 and the exit ek edges of the ddgraph (DDG).
    The set Si of successors of each edge ei.
    Population size (PS);
    Maximum no. of generations (MG);
    Probability of crossover (PX);
    Probability of mutation (PM);
Output:
    Set of basis test paths (BTP) for P.
Begin
    Step 1: Initialization
        for i = 1 to PS
            Initialize each path pi ← φ;
        Initialize the set of basis test paths BTP ← φ;
        nRun ←0;
    Step 2: Generate basis test paths
        While (the set BTP is not a basis set)
            Begin
              nRun ← nRun + 1;
              for i = 1 to PS // Create Initial_Population;
                  Put each path pi ← {e0, ek};
              Current_population ← Initial_Population;
              No_Of_Generations ← 0;
              For each individual of the current population do
                Begin
                    Convert the current chromosome to the corresponding path;
                    Evaluate the current path using equation (1);
                    If (the current path is independent path) then
                        Add the current path to the set BTP;
                        nPaths ← nPaths + 1;
                    End If
              End For;
              Keep the best individual of the current population;
              While (the best individual is not independent path and No_Of_Generations ≤ MG) do
                  Begin
                    Select set of parents of new population from members of current population using
                          roulette wheel method;
                    Create New_Population using crossover and mutation operators;
                    Current_Population ← New_Population;
                    For each individual of Current_Population do
                    Begin
                      Convert current chromosome to the corresponding path;
                      Evaluate the current path using equation (1);
                      If (the current path is independent path) then
                        Add the current path to the set BTP;
                        nPaths ← nPaths + 1;
                      End If
                    End For;
                    Apply Elitist function;
                    Enhance the Current_Population by applying the breeding operator;
                    For each individual of Current_Population do
                    Begin
                      Convert current chromosome to the corresponding path;
                      Evaluate the current path using equation (1);
                      If (the current path is independent path) then
                        Add the current path to the set BTP;
                        nPaths ← nPaths + 1;
                      End If
                    End For;
                    No_Of_Generations ← No_Of_Generations + 1;
                  End While;
            End While;
    Step 3: Produce output
        Return set of basis test paths for P, and set of edges covered by each test path;
        Report on uncovered edges, if any;
End.
```

Figure 2: The Overall Algorithm.

C. A Basis Test Paths Generation Tool

We are implementing a basis test paths generation tool based on our proposed strategy. This tool consists of two main modules:

1) The Analysis Module.

2) Path Generation Module.

Figure 3 shows the overall diagram of our proposed tool. We give a more details of these two modules of our tool in the following subsections.

1) Analysis Module

The analysis module has been built to perform the following tasks:

- Read the program under test.
- Classify program statements and reformats them to facilitate the construction of the program CFG.
- Construct the control-flow graph of the reformatted version of the program (see Figure 1(b)).
- Construct the ddgraph by reducing the control flow graph using the REDUCE algorithm [2]. Figure 1(c) gives an example ddgraph after applying the REDUCE algorithm on the control flow graph.
- Find the set of successors for each edge in the ddgraph.

Pass the control-flow graph, ddgraph, and table of successors to the test path generation module.

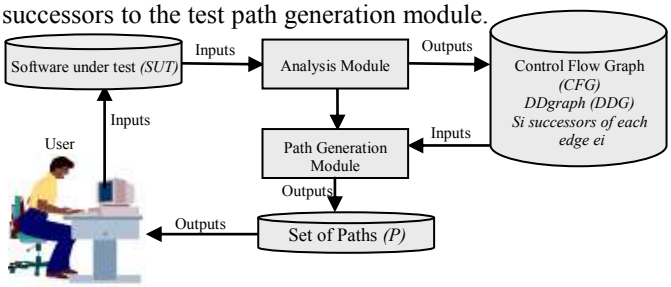

Figure 3: The block diagram of the proposed tool

TABLE I. THE SET OF SUCCESSORS OF EACH EDGE IN THE DDGRAPH.

edge	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
Successors	e_1, e_2	e_3	e_3	e_4, e_5	e_6, e_7	e_9	e_8	e_8	e_4, e_5	NA

In the following paragraphs, we give a brief description for our proposed module for path generation of C++ programs. This module uses the new genetic algorithm to solve the problem of deriving a set of basis test paths.

2) Path Generation Module

The path generation module uses the suggested genetic algorithm to generate set of basis test paths. This module will start by initializing all paths by the *entry* and *exit* edges. Then it will increment each path by adding new edges until get a complete path. For example $\{e_0, e_9\}$ is the initialization of any path then the algorithm will add new edges according to the operators of the genetic algorithm until get a complete path such as $\{e_0, e_2, e_4, e_6, e_9\}$. The algorithm will repeat this process until generating the set of basis test paths or satisfying a stop condition.

V. A CASE STUDY

In this section, we introduce a case study to show how the proposed strategy can find a basis set of test paths for the example program in Figure 1. In the following, we will show all the steps of our proposed genetic algorithm. Suppose that the population size (*PS*) is 4.

1) Initial Population:

The initial population is four individuals each one contains the entry and the exit edges only. Table 2 shows the four chromosomes of the initial populations.

2) Evaluation of the Current Population

Suppose that the current population has the following four chromosomes: C_1=(0, 1, 3, 5, 9), C_2=(0, 2, 3, 4, 9), C_3=(0, 2, 3, 5, 9) and C_4=(0, 1, 3, 4, 9). To find the fitness value of each chromosome, we convert each chromosome into the corresponding path. Then, we use equation (1) to find the fitness value for the chromosome. In each chromosome there are five edges in its corresponding path (L=5), and four adjacent edges (d=4). Then, $w_i = \frac{1}{5}, i = 1,...,5$. Therefore, the

fitness value of $C_1 = ft(C_1) = w_1 + w_2 + w_3 + w_4 = \frac{4}{5} = 0.8$. Table 3 shows the fitness values of the current population.

TABLE II. THE CHROMOSOMES OF THE INITIAL POPULATION.

Chromosome #	Chromosome	Corresponding Path
C1	0, 9	e_0, e_9
C2	0, 9	e_0, e_9
C3	0, 9	e_0, e_9
C4	0, 9	e_0, e_9

TABLE III. THE FITNESS VALUES OF THE CURRENT POPULATION.

Chromosome #	Corresponding Path	Fitness value
C1	e_0, e_1, e_3, e_4, e_9	0.80
C2	e_0, e_2, e_3, e_4, e_9	0.80
C3	e_0, e_2, e_3, e_4, e_9	0.80
C4	e_0, e_1, e_3, e_4, e_9	0.80

3) Selection:

We use the roulette wheel method to select the parents of the next population. The total fitness $F = ft(C_1) + ft(C_2) + ft(C_3) + ft(C_4) = 3.2$. Table 4 shows the computations of roulette wheel.

TABLE IV. THE ROULETTE WHEEL.

C #	Fitness value	Relative Fitness	Cumulative Fitness	r	Parents
C1	0.80	0.25	0.25	0.70	C2
C2	0.80	0.25	0.50	0.20	C1
C3	0.80	0.25	0.75	0.80	C3
C4	0.80	0.25	1.0	0.15	C1

4) Crossover:

Suppose that the probability of crossover $PX = 0.80$. Therefore, the expected number of chromosomes is 4, where $PX \times PS = 0.8 \times 5 = 4$.

TABLE V. THE SELECTED PARENTS FOR CROSSOVER.

Parents	R	The selected parents	New individual pos=3
Pa1=C2	0.65	e_0, e_1, e_3, e_4, e_9	e_0, e_1, e_3, e_4, e_9
Pa2=C1	0.82	---	---
Pa3=C3	0.87	---	---
Pa4=C1	0.40	e_0, e_2, e_3, e_4, e_9	e_0, e_2, e_3, e_4, e_9

5) Mutation:

Suppose that the probability of mutation $PM = 0.15$. Therefore, the expected number of mutated cells is 3, where $PM \times L \times PS = 0.15 \times 5 \times 4 = 3$.

TABLE VI. THE MUTATION OPERATION.

Current population	r	New population
e_0, e_1, e_3, e_4, e_9	0.5, 0.1, 0.1, 0.2, 0.1	e_0, e_2, e_3, e_4, e_9
e_0, e_1, e_3, e_4, e_9	0.1, 0.6, 0.2, 0.1, 0.1	e_0, e_1, e_3, e_5, e_9
e_0, e_2, e_3, e_4, e_9	0.1, 0.1, 0.4, 0.4, 0.1	e_0, e_1, e_3, e_4, e_9
e_0, e_2, e_3, e_4, e_9	0.1, 0.2, 0.1, 0.1, 0.1	e_0, e_2, e_3, e_5, e_9

6) Breeding:

Suppose that the random number *pos* = number of adjacent edges in the chromosome = 4. Then, insert the successor edge at position *5*. Table 7 shows the results of applying the breeding operator. After computing the fitness of the new population, we get two independent paths $p_1 = e_0, e_1, e_3, e_5, e_9$ and $p_2 = e_0, e_2, e_3, e_5, e_9$. The algorithm will repeat the steps from 3 into 6 to get other two independent paths $p_3 = e_0, e_1, e_3, e_4, e_6, e_8, e_5, e_9$ and $p_4 = e_0, e_1, e_3, e_4, e_7, e_8, e_5, e_9$. We can see that p_1 is a sub-path from p_3 and p_4 as well. Where the cyclomatic complexity of the ddgraph in Figure 1 C(DDG) = 10 – 8 + 1 = 3. Therefore, the set of basis test paths consists of the three paths p_2, p_3, and p_4.

TABLE VII. THE BREEDING OPERATION.

Current population	Successors	New population	Fitness value
e_0, e_2, e_3, e_4, e_9	e_6, e_7	$e_0, e_2, e_3, e_4, e_7, e_9$	0.83
e_0, e_1, e_3, e_5, e_9	e_9	$e_0, e_1, e_3, e_5, e_9, e_9$	1.0
e_0, e_1, e_3, e_4, e_9	e_6, e_7	$e_0, e_1, e_3, e_4, e_6, e_9$	0.83
e_0, e_2, e_3, e_5, e_9	e_9	$e_0, e_2, e_3, e_5, e_9, e_9$	1.0

VI. CONCLUSION

We introduced a new genetic algorithm based strategy for generating set of basis test path which can be used as testing paths in any basis path testing technique instead of selecting these paths manually. We defined all key elements of genetic algorithm such as chromosome representation, crossover, mutation, and fitness function. In addition, we present a case study to show the efficiency of our new strategy. Our future work concerns on doing more experiments to measure the efficiency of our strategy and compare it with other work. We will study how the proposed strategy can handle the loop.

REFERENCES

[1] T. McCabe, J. Thomas, Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-99, Washington D.C., 1982.

[2] A. Bertolino, M. Marre, "Automatic Generation of Path Covers Based on the Control flow analysis of computer Programs" IEEE Transaction on Software on software Engineering, vol.20(12), pp. 885-899, 1994.

[3] J. Poole, "A Method to Determine a Basis Set of Paths to Perform Program Testing" http://hissa.nist.gov//publications/nistir5737, 2004

[4] Z. Guangmei, C. Rui, L. Xiaowei, H. Congying "The Automatic Generation of Basis Set of Path for Path Testing", Proceedings of the 14th Asian Test Symposium (ATS '05), 2005.

[5] Jun Yan, Jian Zhang "An efficient method to generate feasible paths for basis path testing" Information Processing Letters, Vol. 107, Issues 3-4, pp. 87-92, 31 July 2008.

[6] Z. Zhonglin, M. Lingxia, "An Improved Method of Acquiring Basis Path for Software Testing" Proceedings of 5th International Conference on Computer Science & Education, pp.1891-1894, China, 2010.

[7] D. Qingfeng, D. Xiao "An Improved Algorithm for Basis Path Testing" International Conference on Business Management and Electronic Information (BMEI), pp. 175 – 178, 2011.

[8] D. Gonga, W. Zhanga, X. Yaob "Evolutionary Generation of Test Data for Many Paths Coverage Based on Grouping" Journal of Systems and Software, In Press, Corrected Proof, Available online 25 June 2011.

[9] P. M. S. Bueno, M. Jino, W. E. Wong "Diversity oriented test data generation using metaheuristic search techniques" Journal of Information Sciences, In Press, Corrected Proof, Available online 23 January 2011.

[10] J. R. Bint, Renate Site, "Optimizing Testing Efficiency with Error Prone Path Identification and Genetic Algorithms" 2004 Australian Software Engineering Conference (ASWEC'04), Australia, pp.106-115, 2004.

[11] S. Haiying, Method and Practice of Software Testing, Beijing: China Railway Publishing House, 2009.

[12] J. Holland, Adaptation in Natural and Artificial Systems, ISBN 0 472 08460 7. University of Michigan Press, Ann Arbor, MI, 1975.

[13] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd Edition, Springer, 1999.