Automatic Generation of Data Flow Test Paths using a Genetic Algorithm

Moheb R. Girgis
Department of Computer Science, Faculty of Science, Minia University, Egypt

Ahmed S. Ghiduk
College of Computers and Information Technology, Taif University, KSA & Faculty of Science, Beni-Suef University, Egypt

Eman H. Abd-Elkawy
Dept. of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Egypt

ABSTRACT
Path testing a program involves generating all paths through the program, and finding a set of program inputs that will execute every path. Since it is impossible to cover all paths in a program, path testing can be relaxed by selecting a subset of all executable paths that fulfill a certain path selection criterion and finding test data to cover it. The automatic generation of such test paths leads to more test coverage paths thus resulting in efficient and effective testing strategy. This paper presents a structural-oriented technique that uses a genetic algorithm (GA) for automatic generation of a set of test paths that cover the all-uses criterion. In the case of programs that have loops, the proposed technique generates paths according to the ZOT-subset criterion, which requires paths that traverse loops zero, one and two times. The proposed GA uses a binary vector as a chromosome to represent the edges in the DD-graph of the program under test. The set of paths generated by the proposed GA can be passed to a test data generation tool to find program inputs that will execute them. Experiments have been carried out to evaluate the effectiveness of the proposed GA compared to the random test path generation technique.

General Terms
Software Engineering, Software Testing.

Keywords
Automatic test path generation, Data flow testing, Genetic algorithms.

1. INTRODUCTION
Path testing is a structural testing method, which requires that every path through a program to be executed at least once. However, it is generally impossible to achieve this goal, because a program that has loops may contain an infinite number of paths. This problem can be solved by selecting a subset of all executable paths that fulfill certain path selection criterion. The automatic generation of such test paths leads to more test coverage paths thus resulting in efficient and effective testing strategy.

Several path generation methods have been proposed. For example, Bertolino and Marre [1] provided a generalized algorithm that finds a path that covers every arc in a given program control flow graph (CFG). Most other research studies have focused on the automatic generation of a basis set of paths, which is a set of linearly independent test paths, where the number of test paths in this set equals to the cyclomatic complexity of program defined by McCabe [2], (see e.g., [3], [4], [5], [6], [7]).

Genetic algorithms (GAs) have been successfully used in software testing activities such as test data generation, (see e.g., [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]). But very little attention has been paid to use GAs in path testing [21]. For example, Bint and Site [22] developed a variable length GA for identifying the most error prone path clusters in a program; and Ghiduk et al. [23] introduced a strategy for automatically generating a set of basis test paths using a variable length GA.

This paper presents a structural-oriented technique that uses a genetic algorithm for automatic generation of a set of test paths that cover one of the data flow path selection criteria, developed by Rapps and Weyuker [24], namely the all-uses criterion. The genetic algorithm conducts its search by constructing new paths from previously generated paths that are evaluated as effective test paths. In the parent selection process, the GA uses the roulette wheel method. In the case of programs containing loops, the proposed technique generates paths according to the ZOT-subset criterion: "Each loop in a program is iterated zero, one, and two times in execution" [25].

This paper is organized as follows: Section 2 describes the data flow analysis technique used to implement the all-uses criterion. Section 3 describes the proposed GA for automatic test paths generation. Section 4 describes the phases of the proposed GA-based path testing system and gives the results of applying this system to an example program. Section 5 presents the results of the experiments that are conducted to evaluate the effectiveness of the proposed GA compared to the random test paths generation technique.

2. DATA FLOW ANALYSIS
Data flow analysis focuses on the interactions between variable definitions (defs) and references (uses) in a program, i.e. the def-use associations. Data flow analysis techniques use a control flow graph representation of a program to compute def-use associations.

The control flow of a program can be represented by a directed graph, with a set of nodes and a set of edges, called the control flow graph (CFG). Each node represents a statement. The edges of the graph are possible transfers of control flow between the nodes. A path is a finite sequence of nodes connected by edges. A complete path is a path whose first node is the start node and whose last node is an exit node. A path is def-clear with respect to a variable if it contains no new definition of that variable.

In our work, we use a reduced form of the CFG, called the DD-graph, in which each edge represents a DD path (decision-decision path). Figure 2 shows the DD-graph that

corresponds to the CFG of the example program shown in Figure 1. Table 1 shows the DD-paths that correspond to the edges of the DD-graph shown in Figure 2.

```
1.  using System;
2.  using System.IO;
3.  public class prog5
4.  {
5.     static void Main()
6.     {
7.        int a, b, c, n;
8.        a = Int32.Parse(Console.ReadLine());
9.        b = Int32.Parse(Console.ReadLine());
10.       if (a < 5)
11.       {
12.          c = a;
13.       }
14.       else
15.       {
16.          c = b;
17.       }
18.       n = c;
19.       while (n <= 8)
20.       {
21.          if (b > c)
22.          {
23.             c = 3;
24.          }
25.          else
26.          {
27.             n = n + c;
28.          }
29.          n = n + 1;
30.       }
31.    Console.WriteLine(" {0} {1} {2}", a, b, n);
32.    }//end main
33. }//end class
```

Figure 1: Example program.

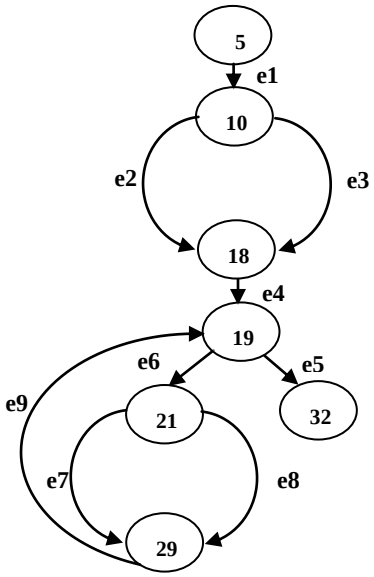

Figure 2: The DD-graph of the example program.

Table 1: The edges of the DD-graph shown in Figure 2 and the corresponding DD-paths.

Edge	DD-Path
e1	5 6 7 8 9 10
e2	10 14 15 16 17 18
e3	10 11 12 13 18
e4	18 19
e5	19 31 32
e6	19 20 21
e7	21 25 26 27 28 29
e8	21 22 23 24 29
e9	29 30 19

The all-uses criterion is one of the data flow testing criteria proposed by Rapps and Weyuker [24]. It requires a def-clear path from each def of a variable to each use of that variable to be traversed. The def-clear paths required to satisfy the all uses criterion, called def-use paths, are constructed from the def-use associations of program variables by using the technique described in [26]. Table 2 shows the list of def-use pairs and killing nodes of the example program. The killing nodes are the set of nodes that must not be included in any def-use path, (nodes containing other defs of the variable).

Table 2: List of def-use pairs and killing nodes of the example program.

Def-Use Pair #	variable	Def-Node	Use-Node	Killing Node
1	a	8	10	-1
2	a	8	12	-1
3	b	9	16	-1
4	c	12	18	16
5	c	16	18	12
6	n	18	19	-1
7	n	29	19	18
8	b	9	21	-1
9	c	12	21	16
10	c	16	21	12
11	c	23	21	12,16
12	c	12	27	16,23
13	c	16	27	12,23
14	n	18	27	-1
15	c	23	27	12,16
16	n	29	27	18
17	n	18	29	27
18	n	27	29	18
19	n	29	29	18
20	a	8	31	-1
21	b	9	31	-1
22	n	18	31	27,29
23	n	29	31	18,27

3. A GENETIC ALGORITHM FOR TEST PATHS GENERATION

This section describes the proposed GA for automatic test path generation. The algorithm searches for test paths that satisfy the all-uses criterion. In the case of programs containing loops, the proposed GA generates paths according to the ZOT-subset criterion.

3.1 Representation

The algorithm uses a binary vector as a chromosome to represent the edges in the DD-graph of the program under test. The length, L, of the vector equals to the number of the edges of the DD-graph of the program under test, including two extra edges representing the entry and exit edges, plus the number of edges contained in loops, as those edges are represented twice. This representation guarantees that the paths generated for programs containing loops satisfy the ZOT-subset criterion. For example, the set of main edges of the DD-graph of the example program, shown in Figure 2, is: e1, e2 … e9, in addition to the entry and exit edges, and the edges contained in the While loop are e6, e7, e8, and e9. A copy of the loop edges are added after the last edge, e9, with numbers starting from 10, i.e., they will be e10, e11, e12, and e13. So, the chromosome length becomes 15, and it takes the following form:

e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12	e13	e14

where e0 and e14 are the entry and exit edges, respectively, and the shaded genes, e10 to e13, represent a copy of the loop edges, e6 to e9.

Let us consider an example chromosome: 110111101110111. Using the above representation, this chromosome represents the following edges:

e0, e1, e3, e4, e5, e6, e8, e9, e10, e12, e13, e14

These edges form the following connected path:

e0, e1, e3, e4, e6, e8, e9, e10, e12, e13, e5, e14

By replacing each edge with its corresponding DD-path, we get the path in terms of the program statements as follows:

5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,30,19,20,21,22,23,24,29,30,19,31,32

3.2 Initial Population

As mentioned above, each chromosome (as a test path) is represented by a binary string of length L. The algorithm randomly generates POPSIZE L-bit strings to represent the initial population, where POPSIZE is the population size. The appropriate value of POPSIZE is experimentally determined. Each test path in the generated population must satisfy the connectivity condition, i.e., it consists of a sequence of connected edges. If the generated chromosome does not represent a connected path, the algorithm discards it.

3.3 Evaluation Function

The algorithm evaluates each test path by determining the set of def-use paths in the program that are covered by this test path. (A test path is said to cover a def-use path, if it includes a subpath, which starts at the def-node and ends at the use node of the def-use path and does not pass through its killing nodes.) The fitness value fitness_value(v_i) for each chromosome v_i (i = 1, …, POPSIZE) is calculated as follows:

$$\text{fitness\_value}(v_i) = \frac{\text{no. of def-use paths covered by } v_i}{\text{total no. of def-use paths}} \quad (1)$$

3.4 Selection

After computing the fitness of each test path in the current population, the algorithm uses the roulette wheel method [27] to select test paths from the effective members of the current population that will be parents of the new population. If none of the members of the current population was effective, all the members of current population are considered the parents of the new population.

3.5 Recombination

The algorithm uses two operators, crossover and mutation, 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 with a predetermined probability pc. During crossover, two parents (chromosomes) exchange substring information (genetic material) at a random position in the chromosome to produce two new strings (offspring). Any of the offspring that does not satisfy the connectivity condition will be discarded.

Mutation: It is performed on a gene-by-gene basis. Mutation always operates after the crossover operator, and changes each gene with the pre-determined probability pm. Every gene (in all chromosomes in the whole population) has an equal chance to undergo mutation. A gene is mutated by replacing its corresponding edge with another edge from its siblings (edges with the same parent are called siblings). If the mutated chromosome does not satisfy the connectivity condition, it will be discarded.

4. THE PROPOSED GA-BASED PATH TESTING SYSTEM

This section describes the phases that comprise the proposed GA-based path testing system. The system is written in C# and consists of the following phases:

1. Static analysis phase.
2. Test path generation phase.

Figure 3 shows the overall algorithm of the proposed system. The two phases are described below.

4.1 Analysis Phase

This phase accepts as input the original program P, analyses it, and produces the following output:

- The static analysis report which contains information about the components of the program P: classes, objects, statements, variables, and functions.
- The CFG of P.
- The list of variables def-use pairs of P.
- The DD-graph of P.

By passing the example program, shown in Figure 1, to the analysis phase of the system, it produces the edges of program's CFG, the edges of program's DD-graph, shown in Table 1 and Figure 2, and the list of its def-use pairs shown in Table 2.

4.2 Test Path-Generation Phase

This phase uses the GA algorithm, described in Section 3, to generate set of test paths that cover the def-use pairs of the given program.

The input to this phase includes:

- List of def-use paths to be covered;
- Number of edges of program DD-graph;
- Population size;
- Maximum no. of generations (MAXGENS);
- Probability of crossover pc;

- Probability of mutation pm;

The output of this phase includes:

- Set of test path that covers the def-use pairs of the given program, if possible. It should be noted that the GA may fail to find paths to cover some of the specified def-use paths when they are infeasible, i.e. no executable path can be found to cover them.
- The test coverage report that shows the generated path(s), and the list of def-use pairs covered by these paths, and the list of uncovered def-use pairs, if any.

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 would correspond to one test path achieving maximum coverage of the program (i.e. traversing all the def-use paths of the program). Whilst this feasible for some programs, the majority of programs cannot be 'covered' by just one test path – it might take many test paths of the program to achieve the desired level of testing. So, we let the population evolves until a combined subset of the population achieves the desired level of coverage. This is done by recording which def-use paths of the program each individual has covered and halting the evolution when a set of individuals has traversed the entire def-use paths of program, if possible. The solution is this set.

Figure 4 shows part of the report produced by the test path generation phase for the example program. This report shows that the GA has found 7 test paths that covered 100% of the def-use pairs shown in Table 2. The generated chromosomes, and the test paths formed from the edges represented by each chromosome, are shown below:

110111000010111 e0,e1,e3,e4,e10,e12,e13,e5,e14 (case 1)

110111101110111 e0,e1,e3,e4,e6,e8,e9,e10,e12,e13,e5,e14 (case2)

110111101111011 e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14 (case 3)

111011000000001 e0,e1,e2,e4,e5,e14 (case 4)

111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14 (case 5)

111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14 (case 59)

110111000011011 e0,e1,e3,e4,e10,e11,e13,e5,e14 (case 91)

Figure 5 shows the test coverage report produced by the system for the 7 test paths generated in 23 generations of the proposed GA for the example program.

The set of paths generated by the proposed GA can be passed to a test data generation tool to find program inputs that will execute them to complete the data flow paths testing of the program under test.

```
/* A GA-based system to automatically generate test paths that cover the all-uses criterion for a given program */
Input:
    The program to be tested P; Population size; Maximum no. of generations (MAXGENS); Probability of crossover pc; Probability of mutation pm;
 Output:
    Set of test paths, and the set of def-use paths covered by each test path;
    List of uncovered def-use paths, if any;
Begin
//The Analysis Phase
   Extract the components of the program P: classes, objects, statements, variables, and functions.
   Form the control flow graph of P. Determine the variables def-use pairs of P.
   Form the DD-graph of P by reducing its control flow graph.
//Test path-Generation Phase
 Step 1: Initialization
   Initialize the def-use coverage vector to zeros;
   Randomly create Initial_Population of chromosomes (test paths) such that each generated test path must satisfy the connectivity condition;
   Current_population ← Initial_Population; def-use coverage percent ← 0; accumulated def-use- coverage percent ← 0; No_Of_Generations ← 0;
   nPaths← 0;
Step 2: Generate test paths
    nEffective ← 0;
    For each member of current population do
   Begin
     Convert the current chromosome to the corresponding path;
     Evaluate the current test path;
       If (some def-use paths are covered) then
         nPaths ← nPaths + 1;
         Add effective test path to set of test paths for P;
         Update the def-use coverage vector;
         Update accumulated def-use- coverage;
         nEffective ← nEffective + 1;
       End If
     End For;
     While (Coverage_Percent ≠100 and No_Of_Generations ≤ MAXGENS) do
     Begin
       If (nEffective > 0) then
         Select set of parents of new population from effective members of current population using roulette wheel method
       Else
         Set of parents of new population ←Current_Population;
       End If;
       Apply crossover, mutation operators to create New_Population such that each new offspring must satisfy the connectivity condition;
       Current_Population ← New_Population; nEffective ← 0;
       For each member of current population do
       Begin
         Convert current chromosome to the corresponding path;
         Evaluate the current test path;
         If (some def-use paths are covered) then
           nPaths ← nPaths + 1;
           Add effective test path to set of test paths for P;
           Update the def-use coverage vector;
           Update accumulated def-use- coverage;
           nEffective ← nEffective + 1;
         End If
       End For;
       Increment No_Of_Generations;
     End While
     Step 3: Produce output
     Return set of test paths for P, and set of def-use paths covered by each path;
     Report on uncovered def-use paths, if any;
End.
```

Figure 3: The overall algorithm of the proposed GA based test paths generation system

5. EXPERIMENTS

The materials of the experiments were 15 small (C#) object oriented programs and structured procedural programs. The used GA parameters were as follows: MAXGENS=100, pc=0.8, pm=0.15 and POPSIZE=4.

The aim of the experiments was to evaluate the effectiveness of the proposed GA compared to the random test (RT) paths generation technique. The random test paths generator selects edges randomly from the DD-graph of the program under test such that these edges form a connected path. To achieve a fair comparison, the random test paths generator was designed to randomly generate sets of POPSIZE test paths in each iteration.

Population Size: 4
Maximum Number of Generation: 100
CROSSOVER PROBABILITY: 0.8
MUTATION PROBABILITY: 0.15
**** GA Started ** * INITIAL POPOULATION ***
1. 110111000010111 e0,e1,e3,e4,e10,e12,e13,e5,e14, (path 1)
2. 110111101110111 e0,e1,e3,e4,e6,e8,e9,e10,e12,e13,e5,e14, (path 2)
3. 110111101111011 e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14, (path 3)
4. 111011000000001 e0,e1,e2,e4,e5,e14, (path 4)
Case 1 :********** SELECTED **********
* Traversed Path: 5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,30,19,31,32,
* FITNESS VALUE : 0.478 * DEF-USE COVERAGE : 47.83 %
* ACCUMULATED DEF-USE COVERAGE: 47.83 %
* COVERED Def_Use_PATHS : 1,2,4,6,7,8,9,17,20,21,23
Case 2 :********** SELECTED **********
*Traversed
Path:5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,30,19,20,21,22,23,24,29,30,19,31,32,
* FITNESS VALUE : 0.087 * DEF-USE COVERAGE : 8.70 %
* ACCUMULATED DEF-USE COVERAGE: 56.52 %
* COVERED Def_Use_PATHS : 11,19
Case 3 :********** SELECTED **********
*Traversed Path:
5,6,7,8,9,10,11,12,13,18,19,20,21,22,23,24,29,30,19,20,21,25,26,27,28,29,30,19,31,32,
* FITNESS VALUE : 0.174 * DEF-USE COVERAGE : 17.39 %
* ACCUMULATED DEF-USE COVERAGE: 73.91 %
* COVERED Def_Use_PATHS : 14,15,16,18
Case 4 :********** SELECTED **********
* Traversed Path: 5,6,7,8,9,10,14,15,16,17,18,19,31,32,
* FITNESS VALUE : 0.130 * DEF-USE COVERAGE : 13.04 %
* ACCUMULATED DEF-USE COVERAGE: 86.96 %
* COVERED Def_Use_PATHS : 3,5,22
***** Selection* The Cases Selected using Roulette Wheel method to be Parents of New Population** are: * Parent 1 = Individual 1 = 110111000010111 =
e0,e1,e3,e4,e10,e12,e13,e5,e14,
* Parent 2 = Individual 4 = 111011000000001 = e0,e1,e2,e4,e5,e14,
* Parent 3 = Individual 2 = 110111101110111 =e0,e1,e3,e4,e6,e8,e9,e10,e12,e13,e5,e14,
* Parent 4 = Individual 3 = 110111101111011= e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14,
*** ** Crossover * The Crossover Operation (Single Point Crossover) *****
* Selected Parents Crossover Position Offspring
* 1 , 2 10 111011000010111 110111000000001
*** The New Population is:
1. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 5)
2. 110111000000001 e0,e1,e3,e4,e5,e14, (path 6)
3. 110111101110111 e0,e1,e3,e4,e6,e8,e9,e10,e12,e13,e5,e14, (path 7)
4. 110111101111011 e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14, (path 8)
Case 5 :********** SELECTED **********
* Traversed Path: 5,6,7,8,9,10,14,15,16,17,18,19,20,21,22,23,24,29,30,19,31,32,
* FITNESS VALUE : 0.043 * DEF-USE COVERAGE : 4.35 %
* ACCUMULATED DEF-USE COVERAGE: 91.30 %
* COVERED Def_Use_PATHS : 10
***** Selection* The Cases Selected using Roulette Wheel method to be Parents of New Population** are:* Parent 1 = Individual 1 = 111011000010111 =
e0,e1,e2,e4,e10,e12,e13,e5,e14,
* Parent 2 = Individual 1 = 111011000010111 = e0,e1,e2,e4,e10,e12,e13,e5,e14,
* Parent 3 = Individual 1 = 111011000010111 = e0,e1,e2,e4,e10,e12,e13,e5,e14,
* Parent 4 = Individual 1 = 111011000010111 = e0,e1,e2,e4,e10,e12,e13,e5,e14,
*** Mutation * The Mutation Operation (Simple Mutation) ***
* Selected Chromosome Mutation Position bits Mutated Chromosome
* 4 2 3 110111000010111
*** The New Population is:
1. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 9)
2. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 10)
3. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 11)
4. 110111000010111 e0,e1,e3,e4,e10,e12,e13,e5,e14, (path 12)
********** NO CASE SELECTED **********
* PARENTS == CURRENT POPULATION **** The New Population is:
1. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 53)
2. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 54)
3. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 55)
4. 110111000010111 e0,e1,e3,e4,e10,e12,e13,e5,e14, (path 56)
********** NO CASE SELECTED **********
* PARENTS == CURRENT POPULATION *
*** Mutation * The Mutation Operation (Simple Mutation) ***
* Selected Chromosome Mutation Position bits Mutated Chromosome
* 3 12 11 111011000011011
*** The New Population is:
1. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 57)
2. 111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (path 58)
3. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 59)
4. 110111000010111 e0,e1,e3,e4,e10,e12,e13,e5,e14, (path 60)
Case 59 :********** SELECTED **********
* Traversed Path:
5,6,7,8,9,10,14,15,16,17,18,19,20,21,25,26,27,28,29,30,19,31,32,
* FITNESS VALUE : 0.043 * DEF-USE COVERAGE : 4.35 %
* ACCUMULATED DEF-USE COVERAGE: 95.65 %
* COVERED Def_Use_PATHS : 13
*** Selection* The Cases Selected using Roulette Wheel method to be Parents of New Population are:* Parent 1 = Individual 3 = 111011000011011 =
e0,e1,e2,e4,e10,e11,e13,e5,e14,
* Parent 2 = Individual 3 = 111011000011011 =
e0,e1,e2,e4,e10,e11,e13,e5,e14,
* Parent 3 = Individual 3 = 111011000011011 =
e0,e1,e2,e4,e10,e11,e13,e5,e14,
* Parent 4 = Individual 3 = 111011000011011 =
e0,e1,e2,e4,e10,e11,e13,e5,e14,
*** The New Population is:
1. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 61)
2. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 62)
3. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 63)
4. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 64)
********** NO CASE SELECTED **********
* PARENTS == CURRENT POPULATION * *** The New Population is:
1. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 85)
2. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 86)
3. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 87)
4. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 88)
********** NO CASE SELECTED **********
* PARENTS == CURRENT POPULATION *
*** Mutation* The Mutation Operation (Simple Mutation) ***
* Selected Chromosome Mutation Position bits Mutated Chromosome
* 3 2 3 110111000011011
*** The New Population is:
1. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 89)
2. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 90)
3. 110111000011011 e0,e1,e3,e4,e10,e11,e13,e5,e14, (path 91)
4. 111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (path 92)
Case 91:********** SELECTED **********
*Traversed Path: 5,6,7,8,9,10,11,12,13,18,19,20,21,25,26,27,28,29,30,19,31,32,
* FITNESS VALUE : 0.043 * DEF-USE COVERAGE : 4.35 %
* ACCUMULATED DEF-USE COVERAGE: 100.00 %
* COVERED Def_Use_PATHS : 12
** GA TERMINATED ** ** NO. OF GENERATIONS = 23
** GENERATED TEST path **
110111000010111 e0,e1,e3,e4,e10,e12,e13,e5,e14, (case 1)
110111101110111 e0,e1,e3,e4,e6,e8,e9,e10,e12,e13,e5,e14, (case 2)
110111101111011 e0,e1,e3,e4,e6,e8,e9,e10,e11,e13,e5,e14, (case 3)
111011000000001 e0,e1,e2,e4,e5,e14, (case 4)
111011000010111 e0,e1,e2,e4,e10,e12,e13,e5,e14, (case 5)
111011000011011 e0,e1,e2,e4,e10,e11,e13,e5,e14, (case 59)
110111000011011 e0,e1,e3,e4,e10,e11,e13,e5,e14, (case 91)

Figure 4: Part of the output report of the path generation phase

Table 3 shows the results of applying the GA technique and the RT technique to the 15 programs. As can be seen from the table, the GA technique outperformed the RT technique in 11 out of the 15 programs in the def-use coverage percentage, and in three of these 11 programs the RT technique did not cover any of the def-use paths. In the other 4 programs, the test paths generated by both techniques reached 100% of the def-use coverage, but the GA technique required less number of generations than the RT technique to achieve this full coverage. For example, for program p#1, the RT technique

required 39 generations to cover 100% of the def-use paths, while the GA technique required only 7 generations. It should be noted that, in the cases where less than 100% coverage is achieved, the programs included some def-use paths that cannot be covered by any test paths due the existence of infeasible paths.

Figure 6 shows a comparison between the number of generations which were required by the GA technique and the RT technique to generate test paths to cover all the def-uses pairs of each tested program. As can be seen from this figure, in 13 out of the 15 programs, the GA technique required less number of generations than the random testing technique to achieve full def-use coverage percentage.

********** Path Number (1)**********
Path: 5 . 6 . 7 . 8 . 9 . 10 . 11 . 12 . 13 . 18 . 19 . 20 . 21 . 22 . 23 . 24 . 29 . 30 . 19 . 31 . 32 .
The newly def use pairs covered by this path:
<(a,8),10>, <(a,8),12>, <(c,12),18>, <(n.18),19>, <(n,29),19>, <(b,9),21>, <(c,12),21>, <(n,18),29>, <(a,8),31>, <(b,9),31>, <(n,29),31>
The def - use pairs not covered yet:
<(b,9),16>, <(c,16),18>, <(c,16),21>, <(c,23),21>, <(c,12),27>, <(c,16),27>, <(n,18),27>, <(c,23),27>, <(n,29),27>, <(n,27),29>, <(n,29),29>, <(n,18),31>
********** Path Number (2)**********
Path: 5 . 6 . 7 . 8 . 9 . 10 . 11 . 12 . 13 . 18 . 19 . 20 . 21 . 22 . 23 . 24 . 29 . 30 . 19 . 20 . 21 . 22 . 23 . 24 . 29 . 30 . 19 . 31. 32
The newly def use pairs covered by this path: <(c,23),21>, <(n,29),29>
The def - use pairs not covered yet: <(b,9),16>, <(c,16),18>, <(c,16),21>, <(c,12),27>, <(c,16),27>, <(n,18),27>, <(c,23),27>, <(n,29),27>, <(n,27),29>, <(n,18),31>
********** Path Number (3)**********
Path :5 . 6 . 7 . 8 . 9 . 10 . 11 . 12 . 13 . 18 . 19 . 20 . 21 . 22 . 23 . 24 . 29 . 30 . 19 . 20 . 21 . 25 . 26 . 27 . 28 . 29 . 30 . 19 . 31 . 32
The newly def use pairs covered by this path: <(n,18),27>, <(c,23),27>, <(n,29),27>, <(n,27),29>
The def - use pairs not covered yet: <(b,9),16>, <(c,16),18>, <(c,16),21>, <(c,12),27>, <(c,16),27>, <(n,18),31>
********** Path Number (4)**********
Pat : 5 . 6 . 7 . 8 . 9 . 10 . 14 . 15 . 16 . 17 . 18 . 19 . 31 . 32 .
The newly def use pairs covered by this path:
<(b,9),16>, <(c,16),18>, <(n,18),31>
The def - use pairs not covered yet:
<(c,16),21>, <(c,12),27>, <(c,16),27>
********** Path Number (5)**********
Path: 5 . 6 . 7 . 8 . 9 . 10 . 14 . 15 . 16 . 17 . 18 . 19 . 20 . 21 . 22 . 23 . 24 . 29 . 30 . 19 . 31 . 32 .
The newly def use pairs covered by this path: <(c,16),21>
The def - use pairs not covered yet: <(c,12),27>, < (c,16),27>
********** Path Number (59)**********
Path :5 . 6 . 7 . 8 . 9 . 10 . 14 . 15 . 16 . 17 . 18 . 19 . 20 . 21 . 25 . 26 . 27 . 28 . 29 . 30 . 19 . 31 . 32 .
The newly def use pairs covered by this path: < (c,16),27>
The def - use pairs not covered yet: <(c,12),27>
********** Path Number (91)**********
Path :5 . 6 . 7 . 8 . 9 . 10 . 11 . 12 . 13 . 18 . 19 . 20 . 21 . 25 . 26 . 27 . 28 . 29 . 30 . 19 . 31 . 32 .
The newly def use pairs covered by this path: <(c,12),27>

Table 3: A comparison between the GA technique and the random RT technique

Program#	No of generations		No of test paths		Def-use coverage %	
	GA	RT	GA	RT	GA	RT
P#1	7	39	4	2	100	100
P#2	2	100	2	0	100	0
P#3	73	100	2	0	100	0
P#4	100	100	3	2	88.88	77.76
P#5	23	100	6	2	100	82.61
P#6	9	12	2	2	100	100
P#7	37	100	5	1	100	40
P#8	51	100	4	1	100	77.78
P#9	41	46	2	3	100	100
P#10	65	100	5	1	100	58.97
P#11	12	82	4	3	100	100
P#12	50	100	8	2	100	65.38
P#13	100	100	4	0	90.90	0
P#14	16	100	6	4	100	94.74
P#15	19	100	3	1	100	88.88

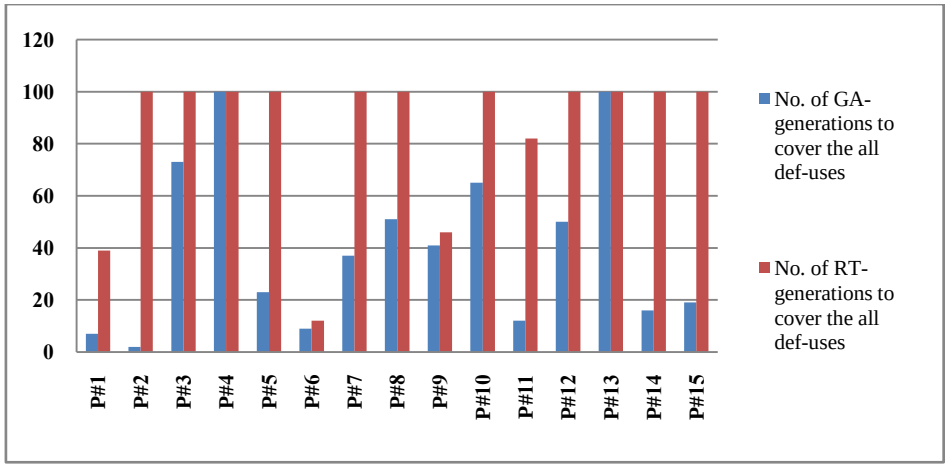

Figure 6: No. of generations required by the GA and the RT techniques to generate test paths to cover all def-uses

6. CONCLUSION

This paper presented a structural-oriented technique that uses a genetic algorithm for automatic generation of a set of test paths that cover the all-uses criterion. The genetic algorithm conducts its search by constructing new paths from previously generated paths that are evaluated as effective test paths. In the case of programs containing loops, the proposed technique generates paths according to the ZOT-subset criterion: "Each loop in a program is iterated zero, one, and two times in execution".

Experiments have been carried out to evaluate the effectiveness of the proposed GA compared to the random test (RT) paths generation technique. The results of these experiments showed that the GA technique outperformed the RT technique, in the def-use coverage percentage, in 11 out of the 15 programs used in the experiments, and in three of these 11 programs the RT technique did not cover any one of the def-use paths. In the other 4 programs, the test paths generated by both techniques reached 100% def-use coverage, but the GA technique required less number of generations than the RT technique to achieve this full coverage. Also, the results showed that in 13 out of the 15 programs, the GA technique required less number of generations than the RT technique to achieve full def-use coverage.

To complete the data flow path testing of the program under test, input data must be found to execute the set of paths generated by the proposed GA. To accomplish this task, we are currently developing an automatic test data generation tool that will check the feasibility of the generated paths and generate test data to execute them.

7. REFERENCES

[1] 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, No. 12, pp.885-899, 1994.

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

[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] M. Pei, E. D. Goodman, Z. Gao, and K. Zhong, "Automated Software Test Data Generation Using A Genetic Algorithm", Technical Report GARAGe of Michigan State University,1994.

[9] M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood, "Genetic Algorithms and the Automatic Generation of Test Data", Technical Report RR/95/195 [EFoCS-19-95], University of Strathclyde, Glasgow G1 1XH, U.K, 1995.

[10] A. E. L. Watkins, "A Tool for the Automatic Generation of Test Data Using Genetic Algorithms", In Proceedings of Software Quality Conference, Dundee, Scotland, 1995.

[11] B. F. Jones, D. E. Eyres, and H. -H. Sthamer, "A strategy for using genetic algorithms to automate branch and fault-based testing', The Computer Journal, Vol. 41, No. 2, 98-107, 1998.

[12] R.P. Pargas, M. J. Harrold, and R. R. Peck, 'Test-Data Generation Using Genetic Algorithms", The Journal of Software Testing, Verification and Reliability, 1999.

[13] P. M. S. Bueno, and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data", The 15th International Conference on Automated Software Engineering (ASE'00), Grenoble, France, 2000.

[14] J. -C. Lin and P. -L. Yeh, "Automatic test data generation for path testing using GAs", Information Sciences, 131 (1-4), 47-64, 2001.

[15] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating Software Test Data by Evolution", IEEE Transactions on Software Engineering, Vol. 27, No. 12, 1085-1110, 2001.

[16] P. M. S. Bueno and M. Jino, "Automatic Test Data Generation For Program Paths Using Genetic Algorithms", International Journal of Software Engineering and Knowledge Engineering, December 2002, Vol. 12, No. 06, pp. 691-709.

[17] M. R. Girgis, "Automatic test data generation for data flow testing using a genetic algorithm", Journal of Universal computer Science, vol. 11, no. 5, pp. 898-915, 2005.

[18] A. S. Ghiduk, M. J. Harrold, M. R. Girgis, "Using genetic algorithms to aid test-data generation for data flow coverage", Proc. of 14th Asia-Pacific Software Engineering Conference (APSEC 07), 2007, pp. 41-48. IEEE Press.

[19] D. W. Gong, W. Q. Zhang, X. J. Yao, "Evolutionary Generation of Test Data for Many Paths Coverage Based on Grouping", Journal of Systems and Software, Vol. 84, No.12, pp. 2222–2233, 2011.

[20] M. R. Girgis, A. S. Ghiduk, and E. H. Abd-Elkawy, "An Approach For Enhancing Regression Testing Using Genetic Algorithm and Data Flow Analysis", International Journal of Intelligent Computing and Information Science, Vol.13, No. 2, APRIL 2013, pp. 115-132.

[21] I. Hermadi, C. Lokan, and R. Sarker, "Genetic Algorithm Based Path Testing: Challenges and Key Parameters", Second WRI World Congress on Software Engineering, 2010

[22] 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.

[23] A. S. Ghiduk, "Automatic Generation of Basis Test Paths Using Variable Length Genetic Algorithm" International Journal of Information Processing Letters, vol. 114, pp. 304-3016, 2014.

[24] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information", IEEE Transactions on Software Engineering, Vol. 11, No. 4, pp. 367-375, 1985.

[25] Girgis, M. R. (1992) 'An experimental evaluation of a symbolic execution system', Software Engineering Journal, 7 (4), 285-290.

[26] M. R. Girgis, "Using symbolic execution and data flow criteria to aid test data selection", The Journal of Software Testing, Verification and Reliability, Vol. 3, No. 2, pp. 101-112, 1993.

[27] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, Mass., 1989.