Genetic Algorithm Based Test Data Generator

Irman Hermadi
Department of Information and Computer Science
King Fahd University of Petroleum & Minerals
KFUPM Box # 868, Dhahran 31261, Saudi Arabia
irmanher@ccse.kfupm.edu.sa

Moataz A. Ahmed
Department of Information and Computer Science
King Fahd University of Petroleum & Minerals
KFUPM Box # 785, Dhahran 31261, Saudi Arabia
mahmed@ccse.kfupm.edu.sa

Abstract- Effective and efficient test data generation is one of the major challenging and time-consuming tasks within the software testing process. Researchers have proposed different methods to generate test data automatically, however, those methods suffer from different drawbacks. In this paper we present a Genetic Algorithm-based approach that tries to generate a set of test data that is expected to cover a given set of target paths. Our proposed fitness function is intended to achieve path coverage that incorporates path traversal techniques, neighborhood influence, weighting, and normalization. This integration improves the GA performance in terms of search space exploitation and exploration, and allows faster convergence. We performed some experiments using our proposed approach, where results were promising.

1 Introduction

Software needs to be tested properly and thoroughly, such that anything that goes wrong can be detected and fixed in advance, before delivery to the user. However, even well tested software is not guaranteed to be bug-free. In general, most of the bugs reported by the users are uncovered via the execution of untested code. This is because either the order in which statements were executed in actual use differed from that during testing, a combination of untested input values are given, and/or the user's environment was never tested [21].

In general, testing itself is defined as the process of executing a program with the intent of finding errors. Hence, a pair of input and its expected output, which is called a *test case*, is said to be successful if it succeeds to uncover errors, and not vice versa. An input datum for a tested program, which is subset of a test case, is called test data. Commonly, the major task of software testing is the design of a minimal set of test cases that reveals as many faults as possible by satisfying particular criteria, called *test adequacy criteria*, e.g. branch coverage [22][15].

Software testing is laborious and time-consuming work; it takes almost 50% of the software system development resources [19][3]. If the testing process could be automated, it should reduce the cost of software development significantly. Other benefits of automated testing includes the following: the number of test cases can be optimized and the test runs can be considerably faster, the testing execution can be done during night-shift and remotely, and the amount of routine work can be reduced [1].

For years, many researchers have proposed different methods to generate test data automatically, a.k.a. test data/case generator [4]. However, those methods suffer from different drawbacks as discussed later. In this paper, we try to overcome such drawbacks by proposing a GA-based approach that integrates path traversal techniques, neighborhood influence, weighting, and normalization.

The paper is organized as follows. Section 2 gives a brief background on software testing. Section 3 describes test case generators and existing genetic algorithm-based test data generators. Section 4 describes the proposed approach followed by section 5 that presents the experimental results and analysis. Section 6 concludes the paper.

2 Software Testing

Generally, software-testing techniques are classified into two categories: static analysis and dynamic testing.

In static analysis, a code reviewer reads the program source code, statement by statement, and visually follows the logical program flow by feeding an input. This type of testing is highly dependent on the reviewer's experience. Typical static analysis methods are *code inspections*, *code walkthroughs*, and *code reviews* [15].

In contrast to static analysis, which uses the program requirements and design documents for visual review, dynamic testing techniques execute the program under test on test input data and observe its output. Usually, the term testing refers to just dynamic testing.

Dynamic testing can be classified into two categories: black-box and white-box. White-box testing is concerned with the degree to which test cases exercise or cover the logical flow of the program [15]. Therefore, this type of testing is called logic-coverage testing or structural testing, because it sees the structure of the program. Black-box testing, on the other hand, tests the functionalities of software regardless of its structure view, a.k.a. functional or specification –based testing. The functional testing is interested only on the outcome in response to given input data.

This paper focuses on logic-coverage testing. Adequacy of logic-coverage testing can be judged using

0-7803-7804-0 /03/$17.00 © 2003 IEEE

different criteria: statement, decision (a.k.a. branch), condition, decision/condition, multiple-condition coverage, and path-coverage; ordered from the weakest to the strongest [15][22][24].

Statement coverage criterion requires every statement in the program to be executed at least once. A stronger one is known as decision or branch coverage. This criterion states that each decision has a **TRUE** and **FALSE** outcome at least once.

A criterion that is sometimes stronger than decision coverage is condition coverage. This criterion requires each condition in a decision takes on all possible outcomes at least once.

Although the condition coverage criterion appears to satisfy the decision coverage criterion, it does not always do so. If the decision **IF (A AND B)** is being tested, the condition coverage criterion would allow one to write two test cases – **A** is **TRUE**, **B** is **FALSE**, and **A** is **FALSE**, **B** is **TRUE** – but this would not cause the **THEN** clause of the **IF** statement to execute.

The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.

A weakness with decision/condition coverage is that although it may appear to exercise all outcomes of all conditions, it frequently does not because certain conditions mask other conditions. But, errors in logical expressions are not necessarily made visible by the condition coverage and decision/condition coverage criteria. A criterion that overcomes this problem is multiple condition coverage. This criterion requires one to write sufficient test cases such that all possible combinations of condition outcomes in each decision, and all points of entry, are invoked at least once.

The utmost coverage is achieved by path coverage [24]. *Path coverage* test adequacy criterion is concerned with the execution of (selected) paths in the program. Since in a program with loops the execution of every path is usually infeasible, complete path testing is not considered in such cases as a feasible testing goal.

In this paper we adopt the path coverage criterion as our testing objective.

3 Test Data Generator

A variety of methods to develop test data/case generators have been proposed over the years [4]. Each method was meant to satisfy a certain test adequacy criterion, and conform to a desired testing objective [10].

The process of automatic test case generation has two steps:

1. Instrumentation: Basically, instrumentation is the process of inserting the probes (tags) at the beginning/ending of every block of code, e.g. at the beginning of each function, before and after the true and false outcomes of each condition. For example in path coverage test criterion, these tags will be used to monitor the traversed path within the program while it is executed with certain test input data.
2. Test Data Evolution: This is a loop where the program is executed with some initial input data (randomized or seeded), feedback is collected, and the input data is adjusted until satisfactory criteria are achieved. The feedback is sort of objective value assigned to the current input data. Furthermore, this objective value can be converted into fitness value that reflects its appropriateness according to the given test criteria.

Searching an input datum from a pool (domain/set) of possible input data that conforms to the test adequacy criteria, e.g., forcing traversing a specific path, is a searching problem. Moreover, it is not only a searching problem; it is an optimization problem [8].

In the early age of automation of software testing, most of the test data generators were using gradient descent algorithms. However, these algorithms are inefficient, and time-consuming, and could not escape from local optima in the search space of the domain of possible input data. Accordingly, meta-heuristic search algorithms have been employed in test data generators as a better alternative. Wegener *et al.* have showed the suitability of using evolutionary algorithms in software testing [20].

Genetic algorithms (GAs) are based on the evolutionary theory [6]. The basic steps of genetic algorithms are the following [5]:

1. Create an initial population of candidate solutions,
2. Compute the fitness values of each of these candidates,
3. Select all the candidates that have the fitness values above or on a threshold, and
4. Make perturbation to each of these selected candidates using genetic operators, *e.g.* crossover.

These steps, except the first initialization step, are repeated until any/all the candidate solutions become solution(s). Before GA can be used, there are four domain-dependent things to do: defining genetic representation of the problem solutions, computing fitness values, selecting candidate selection methods, and defining genetic operators.

GA based test data generator employs a genetic algorithm as its primary search engine in seeking all the suitable test data according to its test adequacy criteria. Researchers have done some work in developing GA based test data generators. Pei *et al.* developed GA based test data generator, which uses binary representation, to achieve path coverage [17]. They used the number of matching predicates (or branching nodes) between traversed subpaths and target subpaths as their heuristic, and a scaling factor to adjust the matching. Roper *et al.* selected branch coverage as their test objective and used directed graph to represent the target program [18]. Actually, Roper *et al.* determined the subset of input domain of the program in the initial input data to limit the

search space by selecting a path from this graph followed by the execution of this path symbolically in order to give a predicate for that path. They defined the fitness function as the number of matched branch directions. Jones *et al.*, on the other hand, computed the fitness function as the summation of predicate branch values along with their condition values to achieve branch coverage [7]. Jones *et al.* used unrolled control flow graph for their path representation, such that the graph is acyclic. In 1999, Pargas *et al.* improved Jones *et al.*'s work by implementing control dependence graph to compute the fitness function instead of using control flow graph [16]. The work done by Michael *et al.* employed the standard as well as differential GA in their test data generator to achieve condition-decision coverage test adequacy criteria, which has higher coverage than the previous approaches [9][10][11][12][13][14]. A test-data generator for stress testing was developed by Alander *et al.* to produce test cases in order to find problematic situations, *e.g.* processing time extremes. They use response time or memory usage warnings from the operating systems to calculate the fitness values [1].

Common observations over existing GA based test data generators are that they work only on a single target, e.g. in the case of path coverage testing, trying one and only one target path. Actually, the fitness functions of most existing GA-based test data generators are designed to generate test date to test only one target at a time. These fitness functions do not consider the competition between and the influence of the different candidate solutions (chromosomes) when trying to generate multiple test data for multiple targets at the same time.

Most of the existing approaches are expensive in term of computation time and efficiency because they do not incorporate any measure that reflects the closeness of a candidate solution to the other candidates into their fitness functions.

A detailed critical survey and review of current approaches can be found in [23].

4 Proposed Fitness Function

In this paper we present an approach that tries to generate multiple test data to cover multiple target paths at one run. Our proposed fitness function is intended to achieve path coverage that incorporates path traversal techniques, neighborhood influence, weighting, and normalization. This integration guides the search (GA) better in terms of search space exploitation and exploration[2].

Mainly the proposed overall fitness of any given individual (chromosome) is calculated as an aggregation of the individual's fitness with respect to the different target paths. The fitness of an individual with respect to a target path depends on two measures: the number of violations V and the distance D. V is the number of violation nodes, that is the number of decision nodes of the target path that either result in a different decision values in the path traversed by the chromosome, or are not encountered in the traversed path by the chromosome. For example, if the decision node "**IF (A ≥ B)**" should result in **TRUE** in the target path, but it results in **FALSE** in the traversed path, then this is considered a violation. Any inner decision nodes will accordingly be considered as violations too.

The overall distance, D, between a target path and the traversed path is calculated as the sum of the predicate distances of the outer violation nodes. A predicate distance is calculated as follows.

If branch predicate is (A > B) then

$$F(X) = (B - A) + k = \begin{cases} + & ; \text{if } (A \le B), \text{ which is FALSE} \\ - \text{ or } 0 & ; \text{if } (A > B), \text{ which is TRUE} \end{cases}$$

where k is the smallest unit of numbers that the computer can represent; i.e. the number resolution.

If branch predicate is (A ≥ B) then

$$F(X) = B - A = \begin{cases} + & ; \text{if } (A < B), \text{ which is FALSE} \\ - \text{ or } 0 & ; \text{if } (A \ge B), \text{ which is TRUE} \end{cases}$$

If branch predicate is (A = B) then

$$F(X) = \begin{cases} |A - B| & ; \text{if } (A \ne B), \text{ which is FALSE} \\ 0 & ; \text{if } (A = B), \text{ which is TRUE} \end{cases}$$

If branch predicate is (A) then

$$F(X) = \begin{cases} \text{large number} & ; \text{if (NOT A), which is FALSE} \\ 0 & ; \text{if (A), which is TRUE} \end{cases}$$

For example, a program contains only two predicates that are ordered sequentially. First predicate (hence n_1 stands for node 1), IF Max < A THEN Max = A, and the second one, IF Min > A THEN Min = A. Given a target path {1 0 2 1}, i.e. node 1 is TRUE and node 2 is FALSE, Max=10, Min=2, and for a given chromosme, A=12, k=1 (assume that integer numbers are used) and the chromosome index is 1. Hence, the traversed path = {1 ((10-12)+1) 2 ((2-12)+1)} = {1 -1 2 -9}. The distance D = 0 + 9, since, in node 1 is not in violation, while in node 2 is in violation. Accordingly, V = 0 + 1.

Accordingly, the test data generation problem becomes a minimization problem where the objective is to minimize the violations and distance. In summary, the overall fitness of any given individual (chromosome) is calculated as follows:

1. For each target path, calculate the distance and violation. To maintain relative meaning between the D and V measures and make them comparable, both D and V are normalized. The normalized measures range between zero and one. D is normalized by the maximum distance of a chromosome in trying to satisfy all the current target paths, or by the maximum distance of all chromosomes in trying to satisfy all the current target paths. V is normalized by the length of its target path that is being compared, and not the traversed one.
2. For each target path, the neighborhood influence is considered. The fitness of a chromosome is influenced by all current target paths, since it is trying to satisfy any of the current target paths. In calculating the fitness of an individual, we consider other individuals within the population in order to overcome cases

where an individual is so close to satisfy a target path, while another individual has just satisfied that target path.

3. Weights are used to determine the relative contribution of the distance and violation to the fitness calculation. If no relative contribution is suggested, both the distance and violation of each individual are given equal weights. Otherwise, weights, which range between zero and one and sum-up to one, are manually set.
4. In order to get the overall fitness value we sum up and normalize the whole fitness values with respect to the different target paths for each chromosome based on the neighborhood approach taken.

Due to the limited number of pages allocated to this paper all the formal details and equations of the fitness function calculation have been omitted. For all the details, interested readers may consult [23].

5 Experimental Setup

5.1 Path Traversal Approach

In our experiments, we adopted two kinds of path traversal approaches to determine the distance D and violation V; these are *path-wise* and *predicate-wise*.

Path-wise approach considers all nodes from the first violation node to the rest of the path as violations in comparing the target path with the chromosome's traversed path. Thus, the number of violations equals to the number of nodes from the first violation node down to the end of the target path.

Predicate-wise approach allows considering non-violated nodes that appear after the occurrence of violated ones.

5.2 Experiments

We have conducted eight different experiments based on the following:

1. Two path traversal approaches (PTA): path-wise (*pt*) and predicate-wise (*pr*),
2. Two weights settings: no weight (*nw*) and 0.1-0.9 weights (*D-V*), and
3. Two possible neighborhood influence (NI) considerations: other paths influence (*op*) and paths-and-chromosomes influence (*oc*). In *op*, chromosomes are competing with each other in trying to cover any of the target paths in a run. While, in *oc*, chromosomes are not only competing but also influencing each other's fitness, for instance, when a chromosome covers a target path, all other chromosomes' fitness are affected as that target path is no longer a target since it has been covered already.

In all, the fitness values are normalized and influenced by other target paths.

5.3 Software Under Test (SUT)

SUT is the tested code as an experimental object. We have selected three SUTs: minimum-maximum (*mm*), triangle classifier (*tc*), and combination of minimum-maximum and triangle classifier (*mt*).

Given an array of numbers, minimum-maximum is a program to find the minimum and maximum numbers. It has two sequential selection statements inside a loop in which all the conditions (predicates) are simple/primitive.

Given three numbers, triangle classifier is a program to classify whether these numbers form a triangle or not. If they are, then the program determines whether the triangle is scalene, isosceles, or equilateral. Triangle classifier has three nested selection statements in which all the decisions are compound predicates.

Given three numbers, *mt* is a program that outputs both the minimum-maximum and the triangle classification as well. This program is formed from *mm* and *tc* to allow more complexity when investigating the performance of our approach.

5.4 GA Parameters

Some parameters to set are stopping criteria, selection functions, maximum number of generations (NOG), number of chromosomes (NOC), generation gap (GG), crossover probability (X-P), and mutation probability (M-P).

We use either the stratification of all target paths or the NOG as the stopping criteria. Roulette Wheel Selection function is adopted. We have used the following values: NOG: 30, 100, 300, and 500; NOC: 30, 50, and 100; GG: 0.5, 0.8, and 1.0; X-P: 0.5, 0.8, and 1.0; M-P: 0.1, 0.3, and 0.8.

5.5 Experimental Plan

For each SUT, based on its built control flow graph, a set of target paths are constructed and selected such that all the logical paths are covered. A target path can be either feasible or infeasible. In a set of target paths, unless the complete testing has finished, we have no idea which ones are feasible and which ones are not.

The number of experiments is 32 (eight different experiments for each of the three SUTs, plus one complete set of experiments for *mm* when excluding the infeasible paths).

6 Experimental Results and Analysis

Experiments show that, in general, compared to existing related works [23], using our fitness function, GA is taking less number of generations with smaller populations to find the number of selected target paths at one time. In other words, the total number of individuals (that is the population size multiplied by the number of generations) examined in our case is less than the sum of the total number of individuals across all runs required in related works.

Several observations on our proposed fitness functions are: increasing the population size will increase the

number of computations significantly since the number of comparisons is the multiplication between the number of chromosomes and the number of target paths, i.e. there is a trade-off that must be balanced. Usually, many target paths are satisfied with individuals in the first generation. This is due to the initial set of target paths that is relatively large, combined with the exploration attained by the randomized selection of the initial population, which allows hitting some of the target paths in the initial set. Later on, the set of target paths becomes smaller as previously satisfied target paths are removed form the set. Exploration and exploitation are, then, taken care of by the neighborhood influence and adjusting the weight. It is worth noting too that infeasible target paths do not hinder the effort to find the feasible ones rather they are helping (see the experiments no 1-16 Table 1).

Table 1 summarizes the experiments.

No	O	T			PF	LG	Remark
		NI	PTA	W			
1	*mm*	*op*	*pt*	*nw*	11	2	NOP=21
2				0.1	11	7	NOIF=8
3			*pr*	*nw*	12	14	W=0.1
4				0.1	10	5	NOG=200
5		*oc*	*pt*	*nw*	13	10	NOC=30
6				0.1	13	16	GG=0.8
7			*pr*	*nw*	13	35	X-P=0.9
8				0.1	12	100	M-P=0.1
9	*mm*	*op*	*pt*	*nw*	13	154	NOP=13
10				0.1	12	22	NOIF=0
11			*pr*	*nw*	12	16	GG=0.8
12				0.1	12	13	NOG=200
13		*oc*	*pt*	*nw*	12	10	NOC=30
14				0.1	12	12	W=0.1
15			*pr*	*nw*	12	4	X-P=0.9
16				0.1	13	11	M-P=0.1
17	*tc*	*op*	*pt*	*nw*	4	25	NOP=4
18				0.1	4	3	NOIF=0
19			*pr*	*nw*	4	7	GG=0.8
20				0.1	4	7	NOG=100
21		*oc*	*pt*	*nw*	4	12	NOC=30
22				0.1	4	5	W=0.1
23			*pr*	*nw*	4	15	X-P=0.9
24				0.1	4	14	M-P=0.1
25	*mt*	*op*	*pt*	*nw*	19	19	NOP=52
26				0.1	19	41	NOIF=No
27			*pr*	*nw*	19	143	t Known
28				0.1	19	15	GG=0.8
29		*oc*	*pt*	*nw*	18	26	NOG=150
30				0.1	19	11	& 100
31			*pr*	*nw*	19	38	NOC=30
32				0.1	19	66	W=0.1
							X-P=0.9
							M-P=0.1

Table 1: Experimental results of full combination

Legend:

COV = Coverage (%) = (PF/NOP) * 100
G-T = A pair of generation number and its number of satisfied target paths
$G\text{-}T_i$ = i^{th} G-T
GG = Generation gap
F = Feasible path
IF = Infeasible path
O = Experimental object
T = Experimental treatment
NOF = Number of feasible paths
NOIF = Number of infeasible paths
NOP = Number of (target) paths
NOG = Number of generation
NOC = Number of chromosomes
PF = Number of found target paths
LG = Generation in which the last target path was found

Table 2 and Table 3 present generation-to-generation achievements.

No	$G\text{-}T_1$	$G\text{-}T_2$	$G\text{-}T_3$	$G\text{-}T_4$	$G\text{-}T_5$
1	1-7	2-4			
2	1-9	2-1	7-1		
3	1-7	2-3	6-1	14-1	
4	1-9	6-1			
5	1-9	2-1	4-1	5-1	10-1
6	1-9	2-2	12-1	16-1	
7	1-7	2-3	13-1	18-1	35-1
8	1-7	2-2	3-2	100-1	
9	1-8	2-2	3-1	16-1	154-1
10	1-6	2-2	3-2	4-1	22-1
11	1-8	2-2	9-1	16-1	
12	1-6	2-2	4-2	7-1	13-1
13	1-8	2-1	3-1	8-1	10-1
14	1-7	2-2	5-1	8-1	12-1
15	1-8	2-2	3-1	4-1	
16	1-8	2-4	11-1		
17	1-3	25-1			
18	1-2	2-1	3-1		
19	1-2	2-1	7-1		
20	1-2	4-1	7-1		
21	1-2	3-1	12-1		
22	1-2	2-1	5-1		
23	1-2	4-1	15-1		
24	1-2	4-1	14-1		

Table 2: Generation to generation achievements for experiments no 1-24

No	$G\text{-}T_i$ ($1 \le i \le NOG$)
25	1-7; 2-1; 5-1; 6-1; 8-2; 9-1; 10-1; 11-1; 12-1; 13-1; 16-1; 19-1
26	1-8; 3-2; 6-1; 7-2; 8-1; 10-1; 11-2; 34-1; 41-1
27	1-9; 2-3; 4-2; 5-2; 6-1; 12-1; 143-1
28	1-8; 2-2; 3-1; 6-1; 7-2; 9-1; 10-2; 15-2
29	1-8; 2-2; 3-1; 4-1; 5-1; 6-1; 8-2; 11-1; 26-1
30	1-6; 2-3; 4-1; 5-3; 6-2; 7-1; 8-1; 9-1; 11-1
31	1-7; 2-2; 4-1; 5-3; 6-1; 7-2; 10-2; 38-1
32	1-7; 2-2; 4-1; 11-1; 15-1; 16-1; 17-1; 19-1; 29-1; 35-1; 42-1; 66-1

Table 3: Generation to generation achievements for experiments no 25-32

The following plots show the best fitness values of some selected experiments. The selected experiments are good representatives of the 32 experiments we have conducted.

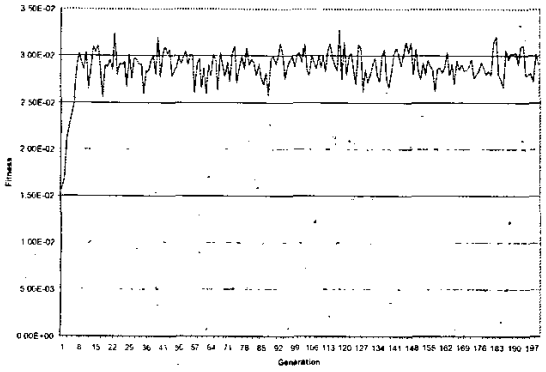

Figure 1: The best fitness of experiment no 5

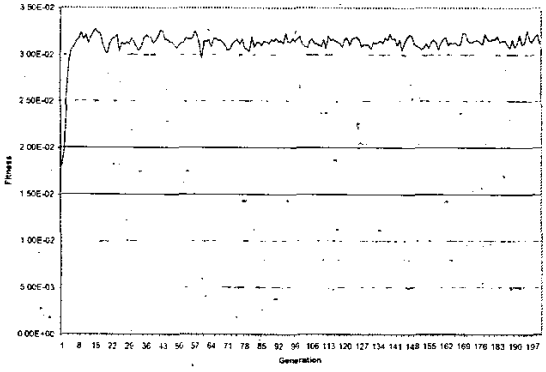

Figure 2: The best fitness of experiment no 6

Figure 1 and Figure 2 present the best fitness of experiment 5 and 6, in which the higher fitness corresponds to the lower (objective) value since we have minimization problem. Those graphs are fluctuated over the generations. This is due to the fact that those target paths that are covered in one generation are removed from the list of target paths of the next generations, which might negatively affect the fitness of those chromosomes with best fitness in that next generation.

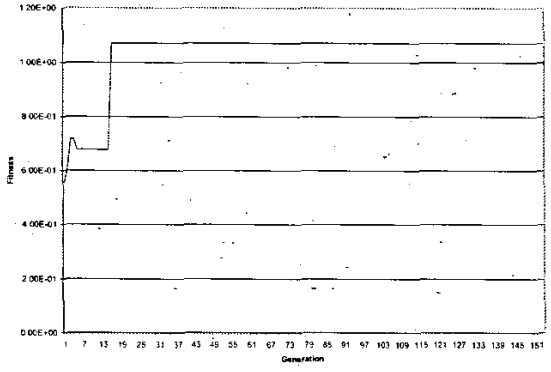

Figure 3: The best fitness of experiment no 9

In Figure 3, the best chromosomes are not influenced by the other chromosomes in the population. Hence, the curve is straight.

Figure 4 and Figure 5 present the best fitness of experiment no 30 and 31.

For more details the reader may consult [23].

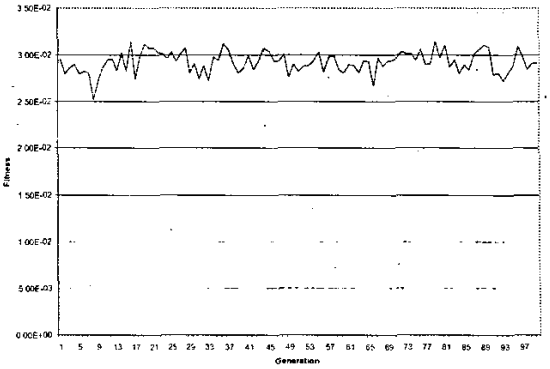

Figure 4: The best fitness of experiment no 30

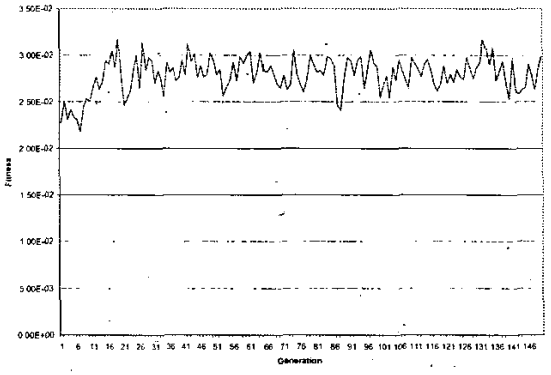

Figure 5: The best fitness of experiment no 31

7 Conclusion

An approach for GA-based multiple test data generation to cover multiple target paths has been presented. The proposed fitness function integrates path traversal techniques, neighborhood influence, weighting, and normalization. This integration improves the GA performance in terms of search space exploitation and exploration, and allows faster convergence. Experimental results have shown that the proposed fitness function is promising.

Acknowledgements

The authors would like to acknowledge the support received from King Fahd University of Petroleum and Minerals.

References

[1] J.T. Alander, T. Mantere, and P. Turunen. Genetic Algorithm Based Software Testing. 1997.

[2] A. Baresel, H. Sthamer, and M. Schmidt. Fitness Function Design to improve Evolutionary Structural Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO-2002, New York, USA, 9-13th July 2002.

[3] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, New York. 1982.

[4] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the Second Conference on Computer Science and Engineering* in Linkoping, pp. 21-28. ESCEL, October 1999.

[5] D.E. Goldberg. Genetic Algorithms: in Search, Optimization & Machine Learning. Addison Wesley, MA. 1989.

[6] J. Holland. Adaptation in Natural and Artificial Systems. MIT Press, MA. 1975.

[7] B. Jones, H. Sthamer, and D. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal* 11(5), September 1996, pp. 299-306.

[8] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, Volume 16, No. 8, pp. 870-879, August, 1990.

[9] G. McGraw, C. Michael, and M. Schatz. Generating Software Test Data by Evolution. Technical report, Reliable Software Technologies, Sterling, VA. February 9, 1998.

[10] C. Michael, G. McGraw, and M. Schatz. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, Volume 27, Number 12, pp. 1085-1110, December, 2001.

[11] C. Michael and G. McGraw. Opportunism and Diversity in Automated Software Test Data Generation. Technical report, Reliable Software Technologies, Sterling, VA. December 8, 1997.

[12] C.C. Michael, G.E. McGraw, M.A. Schatz, and C.C. Walton. Genetic Algorithms for Dynamic Test Data Generation. Technical report; Reliable Software Technologies, Sterling, VA. May 23, 1997.

[13] C. Michael, G. McGraw, M. Schatz, and C. Walton. Genetic algorithms for dynamic test data generation. In *Proceedings of the 12th IEEE International Automated Software Engineering Conference* (ASE 97), pp. 307-308, Tahoe, NV, 1997.

[14] C. Michael and G. McGraw. Automated Software Test Data Generation for Complex Programs. Technical report, Reliable Software Technologies, Sterling, VA. 1998.

[15] G.J. Myers. The Art of Software Testing. John Wiley & Sons, New York. 1979.

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

[17] M. Pei, E.D. Goodman, Z. Gao, and K. Zhong. Automated Software Test Data Generation Using A Genetic Algorithm. 1994.

[18] M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood. Genetic Algorithms and the Automatic Generation of Test Data. 1995.

[19] I. Sommerville. Software. Engineering. 6th Ed. Addison-Wesley, USA. 2001.

[20] J. Wegener, A. Baresel, and H. Sthamer. Suitability of Evolutionary Algorithms for Evolutionary Testing. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 26-29, 2002.

[21] J.A. Whittaker. What Is Software Testing? And Why Is It So Hard? *IEEE Software*. February, 2000.

[22] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29 (4): 366 – 427. December 1997.

[23] I. Hermadi. *Genetic Algorithm Based Test Data Generator*, M.Sc. Thesis, Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, September 2003.

[24] P. McMinn. Improving Evolutionary Testing in the Presence of State Behaviour. PhD Transfer Report, University of Sheffield. October 2002.