Available online at www.sciencedirect.com
ScienceDirect
Computers & Operations Research 35 (2008) 3107–3124
computers & operations research
www.elsevier.com/locate/cor

GA-based multiple paths test data generator

Moataz A. Ahmed[a,*,1], Irman Hermadi[b]

[a]Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[b]Department of Computer Science, Bogor Agricultural University, Bogor 16144, Indonesia

Available online 7 February 2007

Abstract

Developers have learned over time that software testing costs a considerable amount of a software project budget. Hence, software quality managers have been looking for solutions to reduce testing costs and time. Considering path coverage as the test adequacy criterion, we propose using genetic algorithms (GA) for automating the generation of test data for white-box testing. There are evidences that GA has been already successful in generating test data. However, existing GA-based test data generators suffer from some problems. This paper presents our approach to overcome one of these problems; that is the inefficiency in covering multiple target paths. We have designed a GA-based test data generator that is, in one run, able to synthesize multiple test data to cover multiple target paths. Moreover, we have implemented a set of variations of the generator. Experimental results show that our test data generator is more efficient and more effective than others.

Keywords: Software testing; Path testing; Genetic algorithms; Test data generator

1. Introduction

Testing in software industry is a laborious and time-consuming work; it consumes about 50% of software development cost [1–3]. In general, the goal of software testing is to generate a set of minimal number of test cases such that it reveals as many defects as possible by satisfying particular criteria, called *test adequacy criteria*, e.g. path coverage. Automated software testing can significantly reduce the cost of developing software.

Testing is defined as the process of executing a program with the intent of finding errors [3]. 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. In other words, a good test case is one that has a high probability of detecting an as-yet undiscovered error [3]. An input datum for a tested program, which constitutes an element of the test case, is called *test datum*.

For years, many researchers have proposed different methods to generate test data/cases automatically, i.e. different methods for developing test data/case generators [4]. Commonly, searching for an input datum within the domain of possible input data is dealt with as an optimization problem [5].

Researchers have shown the suitability of using evolutionary computations in software testing and developed genetic algorithms (GA)-based test data generators [6,7]. In a recent study, however, Hermadi has observed that existing GA-based test data generators can generate only one test datum at a time [8]. Accordingly, in trying to generate a set of

* Corresponding author.
E-mail addresses: moataz.ahmed@leros.net (M.A. Ahmed), irman.hermadi@ilkom.fmipa.ipb.ac.id (I. Hermadi).
[1] Currently a CTO with LEROS Technologies Corporation, Fairfax, VA 22030, USA.

0305-0548/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2007.01.012

test data (i.e., more than one test datum) to satisfy particular criteria under consideration, any of the existing test data generators should be used more than one time (one run for each required test datum). This practice, however, does not take advantage of the fact that some of the required test data can be readily available as by-products when trying to find other test data. This, hence, makes those existing test data generators inefficient in trying to generate multiple test data.

In this paper, we focus on path coverage as our test adequacy criteria. The goal of the research presented here is to off-load the work in test generation by efficiently generating multiple test data for multiple-path coverage. The paper presents a GA-based test data generator that is capable of generating, in one GA run, multiple test data to cover multiple target paths. The paper also presents several fitness function candidates. These candidates consider a variety of building blocks, path traversal techniques, as well as normalization, weighting, and rewarding schemes.

The remainder of the paper is organized as follows. Section 2 gives a brief background on software testing. Section 3 discusses the GA-based approach for test data generation. It includes a discussion of related work. Section 4 presents our GA-based multiple-path test data generator along with possible variations used. Section 5 discusses the various experiments conducted to empirically validate the approach. Section 6 presents a comparison with other work. Section 7 concludes the paper and points out possible directions for future research.

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 source code of the program or software under test (hereafter SUT), statement by statement, and visually follows the program logic 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* [9].

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 logic flow of the program [9]. Therefore, this type of testing is also known as logic-coverage testing or structural testing, because it considers the structure of the program.

Black-box testing, a.k.a. functional or specification–based testing, on the other hand, tests the functionalities of software irrespective of its structure. Functional testing is interested only in verifying the output in response to given input data.

This paper focuses on logic-coverage testing. Adequacy of logic-coverage testing can be judged using different criteria: statement, decision (a.k.a. branch), condition, decision/condition, multiple-condition coverage, and path coverage [9,10].

Statement coverage criterion requires every statement in the program to be executed at least once. A stronger criterion is known as decision or branch coverage. This criterion requires 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 that 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. Moreover, errors in logical expressions are not necessarily made visible by the condition coverage and decision/condition coverage criteria [9]. 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.

A more practical, widely used and effective coverage is achieved by path coverage [10,11]. *Path coverage* test adequacy criterion is concerned with the execution of (selected) paths in the program. In this paper we adopt the path coverage criterion as our testing objective.

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, i.e. cannot be achieved within an acceptable time with available resources.

3. Test data generators

A variety of methods to develop test data/case generators have been proposed over the years [12,13]. Each method was meant to satisfy a certain test adequacy criterion, and conform to a desired testing objective [10,12,13].

In the early age of automation of software testing, most of the test data generators were using gradient descent algorithms [12,14,15]. However, these algorithms are inefficient, and time-consuming, and could not escape from local optima in the search space of possible input data [7,12,16]. Accordingly, meta-heuristic search algorithms have been employed in test data generators as a better alternative [7,13]. Wegener et al. have showed the suitability of using evolutionary algorithms in software testing [7].

The process of automatic test data generation using GA has two main steps:

1. *Instrumentation*: Instrumentation is, basically, the process of inserting 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, employing the path coverage test criterion, these tags are used to monitor the traversed path within the program while it is being 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 a fitness value assigned to the current input data to reflect its appropriateness according to the given test criteria.

GA are based on the evolutionary theory [17]. The basic steps of GA are the following [13,16]:

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.
4. Make perturbation to each of these selected candidates using genetic operators, e.g. crossover and mutation.

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, defining the fitness function, selecting candidate selection methods, and defining genetic operators.

A 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. In the sequel, we present a summary discussion of some existing works we consider significant with regard to the subject under discussion.

Pei et al. [18] in 1994 observed that most of the test data generators, which were developed in their era, were using symbolic evaluation. They observed that both static and gradient-descent-based dynamic testing were developed. However, they concluded that static testing was not practical, while the dynamic one was not effective. These drawbacks had inspired Pei et al. to develop a single-path-coverage test data generator that employs GA.

Around the same time, Roper et al. [2] in 1995 developed a GA-based test data generator that has an aim to traverse all the branches within a target program. Their generator takes a program and instruments it automatically with probes to provide feedback on the branch coverage achieved.

One year after, Jones et al. [5] in 1996 developed a similar GA-based test data generator to achieve branch coverage. Their major contributions are the use of a sequence of binary strings for individual representation, which is converted to a decimal number prior to the program execution, and the use of unrolled control flow graph (CFG) to represent one, two, or more iterations for each loop, which makes the CFG acyclic. As each branch is executed, the test data generator automatically traverses the CFG to the next branch in a breadth-first manner.

Michael et al. [19–24] in 1997, 1998, and 2001 implemented Korel's function minimization approach [15] in their GA-based test data generator. They have built a test data generator called GADGET (Genetic Algorithm Data Generation Tool), which has the ability to instrument a program automatically with no limitation in the programming language, but it has a restriction that it can only accept scalar inputs. GADGET has the condition–decision coverage as its adequacy criteria. GADGET uses simple GA as well as differential GA. The difference between differential GA and the simple GA is in the recombination process [20]. Michael et al.'s result shows that, in general, the simple GA outperforms

the differential one. GADGET is considered to be the first test data generator to be tested against a large real-world program named b737, which is part of an autopilot system (real-world control software). Michael et al. reported that the performance of random test generation deteriorates for larger programs.

The work done by Pargas et al. [25] in 1999 is an improvement to Jones et al.'s work. The approach they presented also uses branch information to evaluate the fitness function, except it uses control dependency graph for the fitness evaluation, which they claimed it can give more precise fitness evaluation than Jones et al.'s and Michael et al.'s approaches. Pargas et al. parallelized GA to make it faster and also claimed that the approach can provide path coverage with minor modifications.

Bueno and Jino [26] in 2000 proposed an approach that utilizes control and data flow dynamic information. The proposed approach is meant to fulfil path coverage testing. In addition, it also tackles the identification of potentially infeasible program paths by monitoring the progress of the search for required test data. The approach considers a continual population's best fitness improvement as an indication that a feasible path is covered. On the other hand, attempts to generate test data for infeasible paths result, invariably, in a persistent lack of progress.

Lin and Yeh [11] in 2000 extended the work done by Jones et al. In their work, they used the path coverage criterion rather than branch coverage. They also extended the ordinary (weighted) hamming distance such that it can handle different ordering of the target paths that have the same branch nodes. The rationale here is that, in path testing, two different paths may contain the same branches but in different sequences, where the simple hamming distance is no longer suitable. They name the fitness function SIMILARITY, since it calculates the similar items with respect to their ordering within the two different paths, i.e. branches, between the current executed path and the target path. The greater SIMILARITY leads to the better fitness.

Wegener et al. [27] in 2002 developed a fully automatic GA-based test data generator for structural software testing, specifically statement and branch coverage, of real-world embedded software systems. Their fitness function consists of two major building blocks: approximation level and normalized predicate local distance. The approximation level indicates the number of continuously matched branching nodes between the traversed branches by an individual and a target branches (they call it "partial aim"). The local distance is calculated for the individual by means of the branching conditions in the branching node in which the target node is missed. Unfortunately, the report does not describe the normalization scheme of the local distance value. Overall fitness value is the summation of the approximation level value and the local distance value. Although their tool works on only one partial aim after the other, it takes into consideration the execution of a test datum that usually leads to passing several partial aims. Thus, the test soon focuses on those partial aims that are difficult to reach. The stopping criteria used are full statement/branch coverage and number of generations; whichever is satisfied first.

Wegener el al.'s paper does not discuss as whether multiple targets can be covered in one run. However, the approach, or more precisely "the test control", evaluates all individuals generated with respect to all unachieved targets. Thus, other targets found by chance are identified, and individuals with good fitness values for one or more targets are noted and stored for seeding the next subsequent testing of uncovered targets. Furthermore, they reported that full coverage of some programs is achieved, but not for all programs though. According to their paper, they are investigating whether infeasible statements/branches or the number of generations are some of the reasons for not being able to achieve full coverage in some programs.

Hermadi and Ahmed in their paper published in 2003 [28] proposed an approach that satisfies the path coverage criterion. The work tries to generate multiple test data to cover multiple target paths at one run. That paper reported on the early results of the research that lead to the work presented in this paper.

A more detailed discussion and thorough evaluation of existing GA-based test data generators are contained in Hermadi [8]. Hermadi has pointed out some drawbacks related to the existing approaches. Satisfying multiple target paths at a time is among those drawbacks. We present our attempt to develop a GA-based multiple test cases generator in the next section.

4. Multiple paths testing

As has been demonstrated, many GA-based test data generators adopt statement or branch coverage as their objectives. However, by nature, path coverage criterion covers statement and branch coverage criteria; thus, a more effective software structural testing should have path coverage as its objective [29,30].

As the previous section concluded, none of the works on satisfying path coverage focuses satisfying multiple target paths at a time, i.e. covering a set of target paths in a single run of GA. Clearly, covering multiple paths in one run would require incorporating these paths within the fitness calculation. The rationale behind considering multiple paths in one run is based on the observation that while trying to cover a single path, some of the individuals generated cover other paths as a by-product. Accordingly, trying to satisfy multiple paths at a time is expected to greatly increase the effectiveness and efficiency of the test data generator, i.e. attaining more coverage with fewer resources than a single-path test generator that would need multiple runs to cover the same number of paths. In Section 4.1, we discuss the different decisions we considered in designing the fitness function for our GA-based test data generator. We also present a set of different variations of the fitness function.

4.1. Fitness function design

In most of the meta-heuristic search techniques, especially GA, the testing objective is modeled as a fitness function that is to be optimized to find the desired test data. The way in which heuristics of the test data generation problem is incorporated into the fitness function contributes significantly to the performance of the test data generator [31–33].

Hermadi has identified several crucial attributes of a fitness function that may be used in guiding the design of a good fitness function in terms of search effectiveness and efficiency: building blocks, normalization, path traversal method, neighborhood influence, balancing/weighting, adjustment, and rewarding [8]. These attributes guide the design decisions to be made while developing GA-based test data generators for multi-path coverage.

Building blocks: A building block is a constituent of the fitness function. The constituents of a fitness function affect its effectiveness/efficiency in directing the search toward the desired goal. The basic building blocks of our proposed fitness function candidates are based on comparing traversed paths (i.e., paths offered by a GA individual within the population) to target paths in terms of distance D and violation V. D is calculated as the difference between the traversed path and the target path, in term of predicate values for the "unmatched node-branches". For example, consider the predicate "A < B", and assume that this predicate should be "false" within a target path. D will be calculated as ABS(B − A) if A is actually less than B in a given traversed path. V tells how many unmatched nodes exist between the target path and the traversed one. It is worth noting here that $D \geqslant 0$ and $V \geqslant 0$. A node is a branching predicate, i.e. a statement where the program is heading to different branches logically. For example, an IF-THEN-ELSE statement is a node that has two logical branches, i.e. THEN branch and ELSE branch. The objective is to minimize the distance D and violation V. Eq. (1) shows the building blocks of the proposed fitness function:

$$IF_{ij} = D_{ij} + V_{ij}, \tag{1}$$

$$D_{ij} = \sum_{k=1}^{n} D_{ij.k}, \tag{2}$$

$$V_{ij} = \sum_{k=1}^{n} V_{ij.k}, \tag{3}$$

where i is index of target path; j is index of individual; k is index of node in both target path i and traversed path j; and IF is intermediate fitness that looks at the fitness of an individual with respect to one target path. IF is considered as a building block for the overall individual fitness where the fitness with respect to all target paths is considered.

We calculate the predicate value using Korel's distance function [15]. The distance equals to zero if the node-branch of both the target path and the traversed path are matching.

For example, consider Fig. 1, where B1, B2, and B3 indicate branch numbers, T and F stand for True and False, respectively, and numbers that attach to the edges are edge numbers, i.e. to describe the path taken by an input datum.

Let us assume: B1 is "index ⩽ length", B2 is "max < number(index)", and B3 is "min > number(index)". Consider the situation given by a particular individual (say individual number 2) as: index = 1, length = 2, max =10, min =5, number(1) = 2. The corresponding traversed path (TR_2) in this case would be {1 T 2 F 3 T} (i.e., true for Branch 1, false for Branch 2, and true for Branch 3). Consider this target path: TG_1 = {1 T 2 F 3 T}. According to Eqs. (1)–(3), the distance between TG_1 and TR_2, i.e. D_{12}, is equal to the summation of all distances, i.e. distance B1 (D_{B1}), distance B2 (D_{B2}), and distance B3 (D_{B3}). According to Korel's distance function, the distance will be $D_{B1} = 0$, $D_{B2} = 0$, and

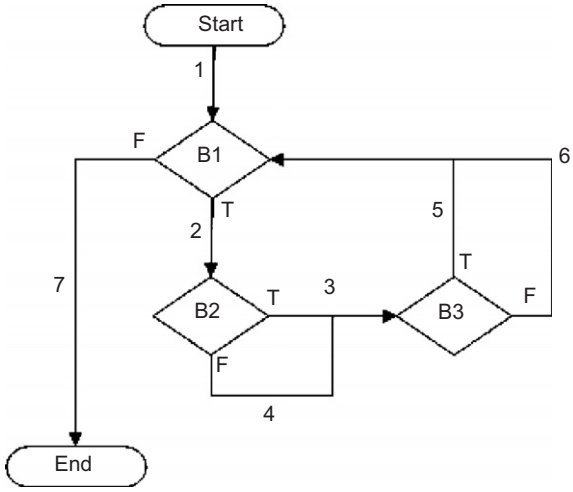

Fig. 1. Control flow graph of an algorithm.

D_{B3}=0; and hence D_{12}=D_{B1}+D_{B2}+D_{B3}=0. In case we have another individual, say TR_3, which has the same variable-value pairs with TR_2 except for number(1) = 6; thus, $D_{13} = D_{B1} + D_{B2} + D_{B3} = 0 + 0 + \text{ABS}(\min - \text{number}(1)) = 1$, and $V_{13} = 1$, since only B3 has been violated.

Normalization: D and V can be computed as either plain or normalized values. Plain (i.e., absolute) distance measure is equal to the summation of all predicate values of unmatched node-branches. On the other hand, plain violations equals to the number of unmatched nodes that is determined by the path traversal approaches, i.e. either path-wise or predicate-wise as discussed below. Obviously, both the plain distance and violation values are not bounded. The values of the building blocks can be normalized across the individuals of the population. Normalization would allow more meaningful comparisons between the different individuals' fitness functions. We did a comparison between fitness functions that use plain building blocks versus those that use normalized ones. With normalization, distance D is normalized by either all target paths or by all target paths as well as other individuals. Violation V, on the other hand, is normalized by the length, i.e. number of branches, of its target path.

Traversal method: The traversal method is the way of measuring the "closeness" between the path exercised by an individual, i.e. a generated input data, and a target path. There are two ways for calculating such "closeness": path-wise and branch-wise (a.k.a. statement-wise as well as predicate-wise). Using the path-wise traversal method, the fitness function does not consider matched (sub) paths after the first deviation (i.e., unmatched node). While, using the predicate-wise method, the fitness function considers subsequent matched (sub) paths after any number of deviations.

Neighborhood influence: In trying to generate multiple test data in one run, a fitness function should consider the pressure of competition among individuals to cover targets. There are two types of fitness value of an individual: its own fitness value and its fitness value that is influenced by the targets and/or other individuals competing to cover similar targets. Accordingly, we consider two distance normalization schemes: based on the target paths (we refer to it as *Op*) only; and based on both target paths and other individuals (we refer to it as *Oc*). In *Op*, we normalize distance over current all uncovered target paths, while *Oc* normalizes distance over both current all uncovered target paths and all existing competing chromosomes in the population.

Weighting: Heuristics might suggest that the different building blocks of the fitness function should have different weights with regard to the contribution to the overall fitness value. We use weights to allow differentiation between the contributions of the distance D and the violation V to the overall fitness.

Adjustment: It applies to any building block of a fitness function and/or to the overall fitness function, according to the selected criterion defined by the test generator designer. The adjustment operations can be addition, multiplication, or whatever necessary actions (e.g. multiplication of any building blocks with a chosen number) the designer considers required for refining the fitness function.

Rewarding: For each target path, the individual that has the highest fitness with regard to that target path compared to other individuals in the population is a winner with regard to that target path, i.e. it is the one whose overall fitness

Table 1
Possibilities of fitness function

No.	Attributes	Values	Code
1	Path traversal approach	Path-wise	Ph
		Predicate-wise	Pr
2	Neighborhood influence	Other paths	Op
		Other paths and chromosomes	Oc
3	Distance and violation normalization	Plain (not normalized)	P
		Normalized	N
4	Weighting scheme	No weight	Wn
		Static	Ws
		Distance-based (dynamic)	Wd
		Violation-based (dynamic)	Wv
5	Rewarding	No reward	Rn
		Reward	Rw
6	Final fitness normalization	Plain (not normalized)	P
		Normalized	N

gets a reward; that is its overall fitness will be positively affected. The rationale, here, is to give such an individual a better chance of survival to the next generation since it is the closest to some target paths.

Final fitness calculation: In order to get the overall fitness value we have to sum up and normalize the whole intermediate fitness, i.e. *IF*, values for each chromosome.

4.2. Possibilities of fitness function

Based on the above discussion on the different decisions that should be made with regard to the fitness function design, Table 1 lists the possible variation points along with their corresponding values. For the sake of easier reference, abbreviations of the possible values are shown under the "Code" column in the table.

The table shows that there are 128 $(=2*2*2*4*2*2)$ fitness function possibilities for all the combination of these attributes.

Based on our intuition, we expect that normalized values would be more meaningful than the plain ones. Accordingly, to reduce the number of possible fitness function candidates to be investigated, we limit the scope to those properties in which that the possible values are normalized. This way, we end up having 32 $(=2*2*4*2)$ possible fitness function candidates.

More details and formal discussion regarding the mathematical formulas governing the fitness functions composition and calculations can be found in Hermadi [8].

5. Experiments

In this section, we present and assess the performance (i.e., strengths and weaknesses) of all proposed fitness functions (i.e., candidates) using several SUTs. The section also discusses the implementation of our GA-based test data generator, including its design, setup, and implementation issues. Finally, we present experimental results and analysis.

5.1. Experiments design

We have conducted seven different experiments using Matlab; each experiment considers one SUT. Each experiment is comprised of sets of runs; one set for each fitness function. Each set is comprised of at least 10 runs, where average performance over each set is reported for each fitness function. The 32 candidate fitness functions are distinguished from each other by their attributes settings (i.e., the values set for the variation points that were discussed in the previous section) as shown in Table 2.

We mainly assess the performance via three graphs: generation-to-generation (G2G) achievement, the best fitness, and cluster convergence (*phi*). G2G achievement is used to analyze the effectiveness and efficiency of each fitness

Table 2
Fitness function design

No.	PT	NI	R	W	Candidate code
1	Ph	Op	Rn	Wn	Ph-Op-Rn-Wn
2				Ws	Ph-Op-Rn-Ws
3				Wd	Ph-Op-Rn-Wd
4				Wv	Ph-Op-Rn-Wv
5			Rw	Wn	Ph-Op-Rw-Wn
6				Ws	Ph-Op-Rw-Ws
7				Wd	Ph-Op-Rw-Wd
8				Wv	Ph-Op-Rw-Wv
9		Oc	Rn	Wn	Ph-Oc-Rn-Wn
10				Ws	Ph-Oc-Rn-Ws
11				Wd	Ph-Oc-Rn-Wd
12				Wv	Ph-Oc-Rn-Wv
13			Rw	Wn	Ph-Oc-Rw-Wn
14				Ws	Ph-Oc-Rw-Ws
15				Wd	Ph-Oc-Rw-Wd
16				Wv	Ph-Oc-Rw-Wv
17	Pr	Op	Rn	Wn	Pr-Op-Rn-Wn
18				Ws	Pr-Op-Rn-Ws
19				Wd	Pr-Op-Rn-Wd
20				Wv	Pr-Op-Rn-Wv
21			Rw	Wn	Pr-Op-Rw-Wn
22				Ws	Pr-Op-Rw-Ws
23				Wd	Pr-Op-Rw-Wd
24				Wv	Pr-Op-Rw-Wv
25		Oc	Rn	Wn	Pr-Oc-Rn-Wn
26				Ws	Pr-Oc-Rn-Ws
27				Wd	Pr-Oc-Rn-Wd
28				Wv	Pr-Oc-Rn-Wv
29			Rw	Wn	Pr-Oc-Rw-Wn
30				Ws	Pr-Oc-Rw-Ws
31				Wd	Pr-Oc-Rw-Wd
32				Wv	Pr-Oc-Rw-Wv

function candidate, while cluster convergence graph is used to analyze the exploration and exploitation behavior of each fitness function candidate. The best fitness graph is meant to analyze the best candidate solution behavior over generations. Having these graphs help in comparing the different candidates. More details about these types of graph can be found in [30]. We have plotted the corresponding graphs for each experiment. However, due to the limited number of pages of the paper we show only two sets of graphs for two SUTs of the seven SUTs. The reader is advised to consult Hermadi [8] for the complete set of graphs.

5.2. SUT's preparation

We have selected seven test programs as SUTs for experimentations; these are: *tc*, *bs*, *is*, *ns*, *mm-t*, *mm-f*, and *mm-i*. These programs and their corresponding target paths are the same considered by other researchers [2,5,11,18,24–28].

1. *Minimum–maximum* (*mm*): Given an array of numbers, *mm* is a program to find the minimum and maximum numbers within the array. The program has two sequential selection statements inside a loop in which all the conditions (predicates) are simple/primitive. During our experiment, we allowed the length of the array to be variable, and restricted the content of the array to integer numbers. *mm-f* and *mm-i* are variations of *mm*. *mm-f* is *mm* with all target paths are feasible ones, while *mm-i* is *mm* with some target paths are infeasible ones.
2. *Triangle classifier* (*tc*): Given three numbers, *tc* 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.

Table 3
GA parameters setup

No.	Parameter	Value
1	Population size	30
2	No of generations	100
3	Generation gap	0.8
4	Selection method	Roulette wheel
5	Crossover method	Single point
6	Crossover probability	0.5 or 0.9
7	Mutation probability	0.1 or 0.3
8	Chromosome type	SUT-based
9	Chromosome size	Variable
10	Allele base	10
11	Allele range	±1000

3. *Bubble sort* (*bs*): Given an array of numbers, *bs* is a program to sort these numbers in an increasing order. The program has two loops that are nested and one selection that is nested inside the inner loop. The outer loop contains compound predicate.
4. *Insertion sort* (*is*): Given an array of numbers, *is* is a program to sort these numbers in an increasing order. The program has two loops that are nested and one selection that is nested inside the inner loop. The inner loop contains compound predicates.
5. *Binary search* (*ns*): Given an array of numbers and a key, *ns* is a program to find a key among these numbers. The program has a single loop that contains a single selection.
6. *Minimum–maximum and triangle classifier* (*mm-t*): Given three numbers, this combined program 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 candidate fitness functions.

Each of the above SUTs poses special characteristics which we would like to investigate the performance of our candidate fitness functions against. For each SUT, we manually instrument the original program without changing its semantic. Then, we construct the corresponding CFG and generate a list of selected target paths from it. We developed our test data generator using Matlab. We also developed an instrumented version of the considered SUTs in Matlab.

5.3. GA parameters setup

In using GA, the values of GA parameters must be set up before hand. Table 3 shows the values for all GA parameters that we use. Selection of these values was subject to trial-and-error practice.

These parameters, however, slightly vary from one experiment to another based on the corresponding SUT. In other words, the same settings was not (could not be) applied across all SUTs. The gain from these variations is that they give an idea on how to set the parameters when dealing with other test programs that have similar characteristics with the SUTs we have used for experimentations.

It is worth noting here that the parameters setting is not only dependent on the characteristics of the SUT, but also on the fitness function candidate adopted as well as the number of target paths being considered. For example, a more complicated SUT with a larger number of target paths is expected to require a larger population size and a larger number of generations to find effective test data (see Fig. 2, where PF and LG represent number of paths found and the last generation where the last feasible target path was found, respectively). Fig. 2 shows that *ns* is an exception in the sense it did not support the expectation that the more the paths, the more the number of generations. This is because the paths of *ns* are considered easier than those of other SUTs.

5.4. GA and fitness function parameters setup

Based on initial trial-and-error results, we selected the rates to be: either 0.1 or 0.3 for static weight, either 0.5 or 0.9 for crossover, and either 0.1 or 0.3 for mutation. The rationale behind the selection of the static weight is that the

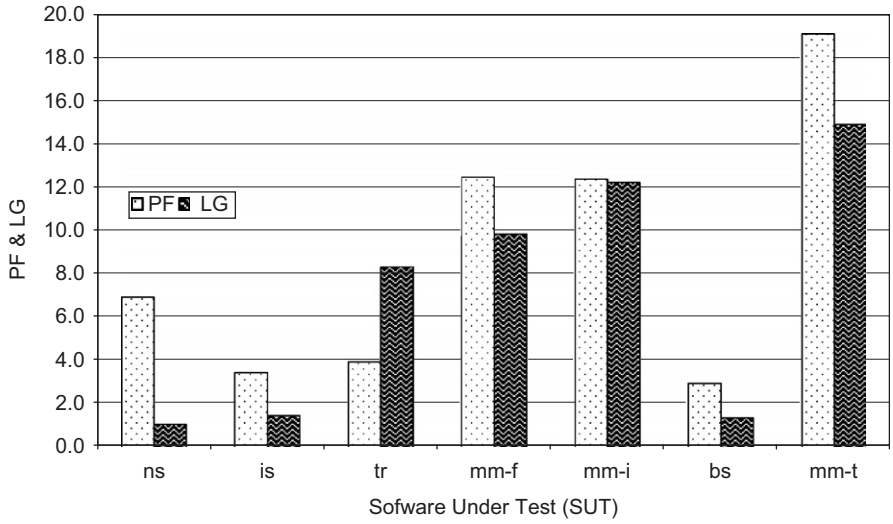

Fig. 2. An average of G2G graph for all SUTs for fitness candidate number 30.

Table 4
Experimentation

No.	SUT	No. of paths	No. of infeasible paths
1	Binary search (*ns*)	7	0
2	Insertion sort (*is*)	4	0
3	Triangle (*tr*)	4	0
4	Minimaxi-f (*mm-f*)	13	0
5	Minimaxi-i (*mm-i*)	21	8
6	Bubble sort (*bs*)	14	11
7	Minmax-tri (*mm-t*)	84	64

predicate distance contributes much less than the violation. As for the crossover and mutation rate, we are trying to maintain a balance between the exploration and exploitation of the search space.

As a pre-experiment to find the best combination to use, we applied all the parameter values combinations to all 32 fitness function candidates using the minimum–maximum (*mm-i*) as a SUT with infeasible paths included in the set of target paths. We used the number of successes, i.e. the number of fitness function candidates that found all the required feasible target paths, to assess the effectiveness of each parameter–value combination.

Two of these combinations have the same number of successes: 0.1-0.5-0.3 and 0.1-0.9-0.3. Accordingly, we just arbitrarily selected one of them, that is 0.1-0.9-0.3. The rationale behind this selection is that it would allow more exploration due to the higher crossover rate.

Table 4 lists all the conducted experiments, where each experiment is composed of 10 runs per fitness function candidate; except for nos. 4, 5, and 7; which are composed of 20 runs each to allow more confidence in the average reported.

Experiments 4 and 5 are meant to observe the effect of infeasible paths on the behavior of the different fitness function candidates.

Paths that are already covered are copied to the list of successful paths along with their corresponding test data; these paths are removed from the list of target paths that are yet to be covered. Hence, during test data generation, there exist two lists: the covered target paths and the (yet-to-be-covered) target paths. At the end, if all target paths were found, the list of target paths will be empty and the list of covered paths will contain all the target paths along with their corresponding test data.

The following sub-sections discuss the experimental results. We organize the discussion as per experiment, i.e. per SUT.

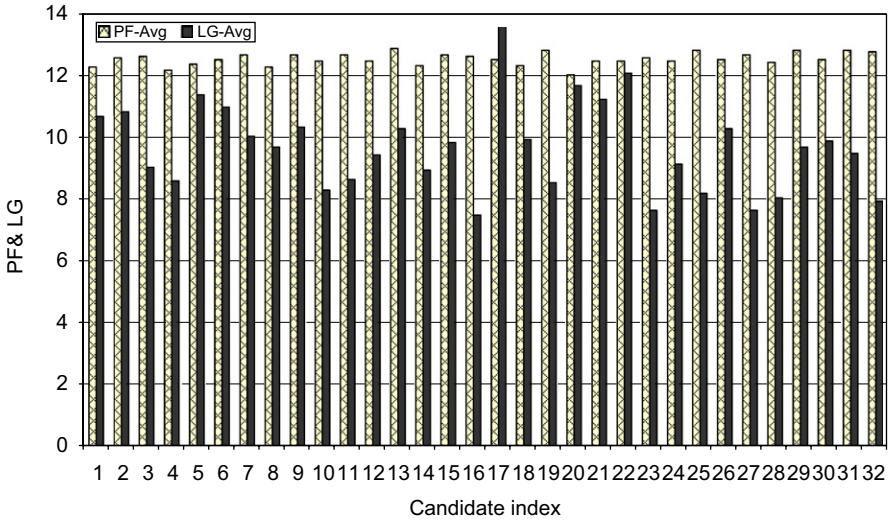

Fig. 3. G2G achievement of *mm-f* on the average over 20 runs.

5.4.1. Binary search (ns)

Almost all (6.96 out of 7, on the average) feasible target paths were found by all the candidates during this experiment. All target paths were found during the first 2 (or 1.25 on the average) generations. These results were based on the average of 10 runs. Moreover, in some runs, all the feasible target paths were found in the first (i.e., initial) generation. Clearly, this behavior was due to the exploration achieved by the random population developed in initial generations.

5.4.2. Insertion sort (is)

During this experiment, as an average of 10 runs, almost all (3.7 out of 4 on the average) feasible target paths were found within 3 (or 1.475 on the average) generations. In some runs, all the feasible target paths were found in the first (initial) generation. The same explanation with *ns* is applicable.

5.4.3. Triangle (tr)

In the triangle classification SUT, all candidates found all (4 out of 4 on the average) feasible target paths were found within not more than 10 (or 7.6 on the average) generations; according to a 10-run experiment.

5.4.4. Minimaxi-f (mm-f)

With regard to the Minimaxi-f SUT, almost all (12.6 out of 13 on the average) feasible target paths were found within not more than 14 (or 9.7 on the average) generations over 20-run experiment (see Fig. 3).

5.4.5. Minimaxi-i (mm-i)

In this experiment, all candidates were able to find almost all (12.52 out of 13 on the average) feasible target paths were found within not more than 36 (or 15.56 on the average) generations; in a 20-run experiment (see Fig. 4). However, from the *phi* graph of the 17th run (arbitrary selected), we can see that some candidates are really doing more exploitation (stable line) of the search space while others are doing more exploration (fluctuated line) (see Fig. 5). Moreover, the best fitness graph indicates that, for the 17th run some of the best individuals of some candidates are indeed affected by other individuals in the population (fluctuated line), i.e. not only affected by the target paths[2] (Fig. 6). The fluctuated lines in this case show that the fitness of the "best" individual may drop from one generation to another due to the competition with other individuals, and/or the removal of the covered-already target paths from the set of targets. A more detailed elaboration is provided in Hermadi [8].

[2] It is worth noting here that the best fitness function graph can go below zero due to the application of the rewarding scheme in some candidates.

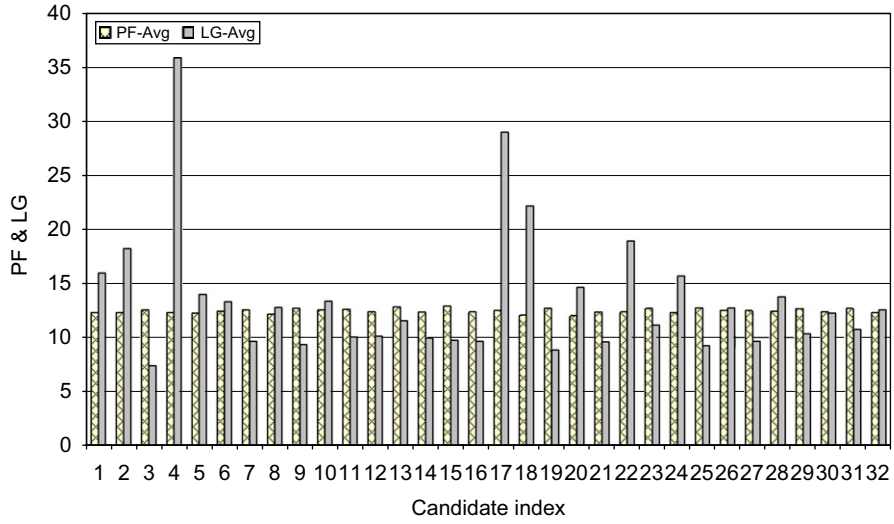

Fig. 4. G2G achievement of *mm-i* on the average over 20 runs.

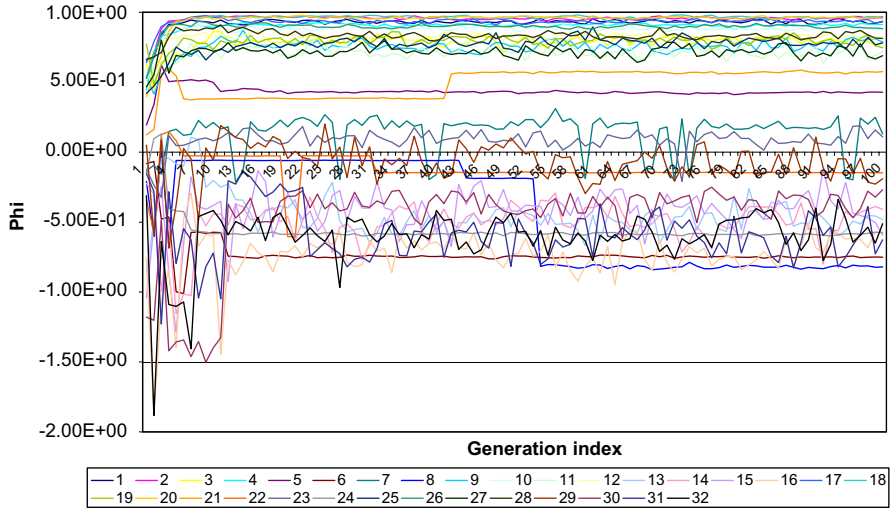

Fig. 5. Phi graph of *mm-i* for a particular run.

5.4.6. *Bubble sort (bs)*

In bubble sort, almost all (2.98 out of 3, on the average) feasible target paths were found within not more than 2 (or 1.06 on the average) generations by all candidates in 10-run experiments. In this case, we could not really see the contribution of each candidate towards finding the target paths, since most of the target paths were found by chance in the first two generations. Obviously, this behavior was due to the exploration achieved by the random population developed in initial generations.

5.4.7. *Minimaxi-tri (mm-t)*

Minimaxi-tri is the most challenging SUT among the set we used in our experiments; this is because it is a combination of *mm* and *tr*, and it has a large number of infeasible target paths. In these experiments, almost all (19.3 out of 20, on the average) feasible target paths were found within 30 (or 19.4 on the average) generations in 20-run experiments (see Fig. 7). However, from the *phi* graph in a particular run (9th run; arbitrarily chosen), we can see that some candidates

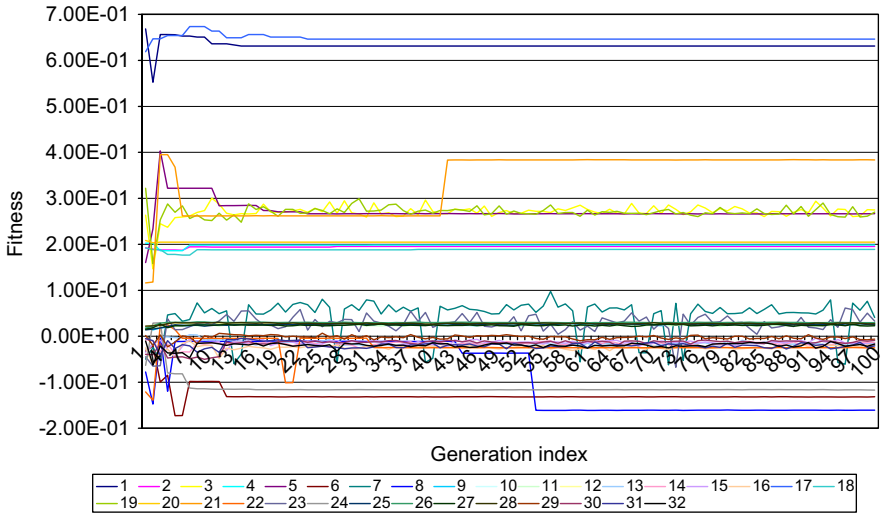

Fig. 6. Best fitness graph of *mm-i* for a particular run.

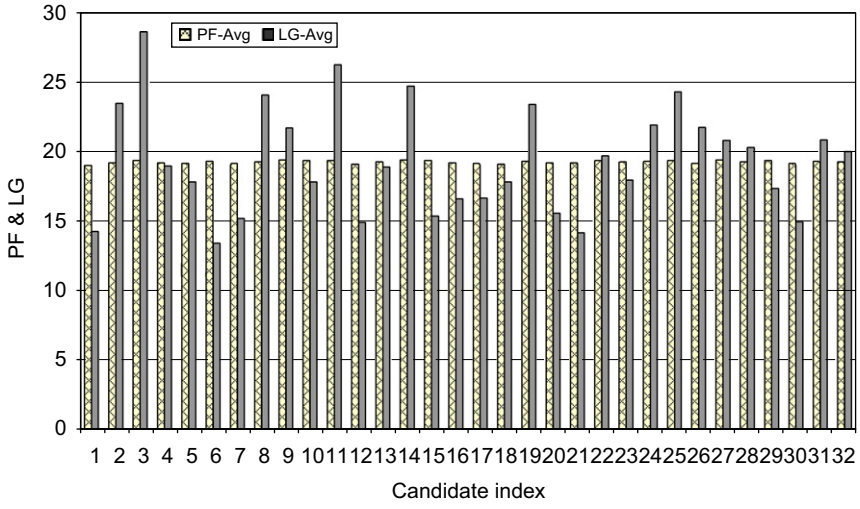

Fig. 7. G2G achievement of *mm-t* on the average over 20 runs.

exploit the search space much more than others do (see Figs. 8 and 9). A more detailed explanation is presented in Hermadi [8].

5.5. Analysis of results

In general, our candidate fitness functions showed to be effective and efficient in handling the required feasible target paths, regardless of the existence of infeasible paths, the path length, and the compound predicates complexity.

The existence of infeasible paths, if any, is not hindering the test data generator to find all given feasible target paths rather it is helping in exploring the search space, i.e. it forces GA to reproduce more explorative inputs. In this case, the candidates that employ rewarding scheme seemed to be more effective in exploring the search space.

In general, predicate-wise candidates are slightly more effective than the path-wise ones, while the path-wise candidates are more efficient than the predicate-wise ones.

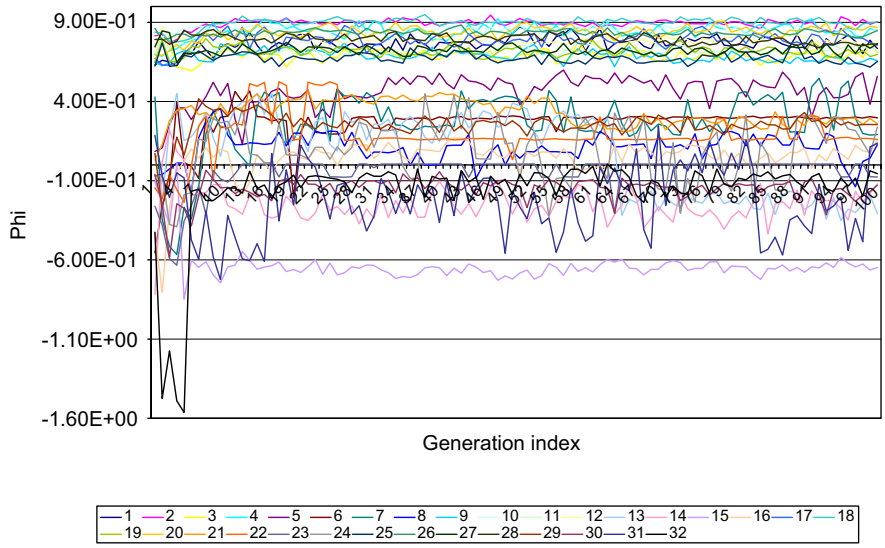

Fig. 8. Phi graph of *mm-t* for a particular run.

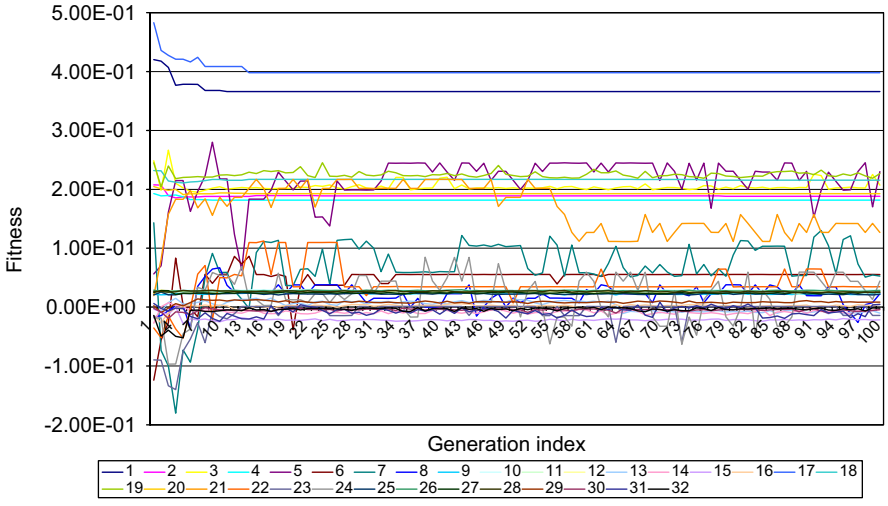

Fig. 9. Best fitness graph of *mm-t* for a particular run.

On the average, the fitness functions that are utilizing neighborhood influence are more effective than otherwise, but not more efficient due to more computation time.

Generally, candidates applying rewarding scheme and/or violation-based weighting are more exploitive than their counterparts.

Usually, many target paths are satisfied by 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. Later on, the set of target paths becomes smaller as previously satisfied target paths are removed from the set. For example in bubble sort (*bs*), almost all (2.98 out of 3) feasible target paths were found within the first two generations, which means that these paths are easy to find randomly.

Increasing the number of target paths, especially the infeasible ones, increases the computation time, since the complexity of the calculation of a candidate is proportional to the number of target paths (for instance, refer to *mm-f* and *mm-i*, see Figs. 3 and 4, respectively).

The type of the predicate influences the search progress; composed predicates (for instance, refer to *tr* and *mm-t*) with the logical operator AND and predicates involving equality relational operator are harder to solve and tend to generate a higher lack of progress in the search.

Deeper branches through the path are harder to satisfy. Longer paths have more constraints to satisfy (for instance, refer to the target paths for *mm-t*) Hence, more computation time is required to generate the input data for these types of paths (see Fig. 2).

6. Comparison with other work

In the set of comparisons presented in this section, we use fitness function number 30; that is considered to be the best on the average across all SUTs; in terms of effectiveness and efficiency.

In Lin's work [11], the "equilateral" target path of a triangle classifier program was selected to show the ability of searching for test cases for a specific path by using GA compared to random testing. Hence, we use the same target paths, and CFG to compare Lin's work with ours.

Actually, the target path that leads to equilateral triangle is the most difficult path to cover by random testing [11], since the path is covered if and only if the three input parameters are positive and equal. The probability of randomly covering this path is 2^{-30} (that is $(2^{15} * 1 * 1)/(2^{15} * 2^{15} * 2^{15})$ where each positive integer is 15 bits). Thus, based on the theory of probability, it would take random testing 2^{30} test cases to reach the target.

Lin's test data generator was able to cover the target path after 10 generations, with a 1000 individuals each; that is a total of 10 100[3] test data on average. Our generator, however, was able to cover the target path using only 180 test cases (that is four generations, with 30 individuals each), on an average of four runs. Moreover, since Lin's approach works on a single target path at a time, we conducted an experiment having one target path at a time. Using our test generator, it takes only 180 test cases (six generations with 30 individuals, i.e. test cases, each) on the average to find the required target path, as well as other target paths (see Table 5).

In Pei's work [18], there are 21 target paths, where eight of them are infeasible, in testing a minimum–maximum (*mm*) program. Among the feasible target paths, the last three paths are the most difficult ones to cover as reported in his work. Thus, we use these target paths for comparison (Table 6).

Table 5
Comparison between Lin's work and ours

Target path	Our approach		Lin's work	
	Found in gen (avg. of three runs)	No. of test data	Found in gen	No. of test data
Single path	4	180	10	10 100
All four paths in Lin's work	4	180	10	10 100

Table 6
Comparison between Pei's work and ours

Target Path	Our approach		Pei94	
	Pop	Results out of gen#	Pop	Results out of gen#
Path 1	30	3	100	15-gen (1429 runs)
			50	68-gen (3042 runs)
Path 2	30	2	500	2-gen (1385 runs)
			50	79-gen (3512 runs)
Path 3	30	8	1000	14-gen (14 986 runs)
All three paths	30	7		

[3] Lin's generator found the required target path in the 11th generation in a 100th individual for a single target path that is considered to be the most difficult one among others.

Table 7
The results of our work using candidate number 30 over 20 runs for minimum–maximum

Run	The last three and most difficult paths in Pei's work											
	Path 1			Path 2			Path 3			All three paths		
	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100
Avg	2.85	2.05	1.35	2.25	2.20	1.45	7.55	5.25	3.60	7.30	6.25	3.55
Std	1.95	0.83	0.49	1.25	1.20	0.60	10.11	2.02	1.54	3.45	3.63	1.93
Min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	1.00	1.00	1.00
Max	8.00	3.00	2.00	5.00	5.00	3.00	47.00	9.00	7.00	17.00	14.00	8.00

Table 8
The results of our work after 20 runs for triangle

Run	Target paths					
	1-2-4-6-8 path			All paths		
	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100
Avg	6.35	5.35	3.45	5.90	5.50	3.80
Std	4.09	1.84	1.67	2.99	2.70	1.91
Min	2.00	2.00	1.00	3.00	2.00	1.00
Max	21.00	9.00	7.00	13.00	13.00	9.00

The results in Table 6 show that Pei's test generator covers the three target paths within 15, 2, and 14 generations on the average, respectively. While with our test generator, the average of three runs shows that the respective target paths were covered within 3, 2, and 8 generations with smaller population sizes. In other word, our test data generator examined less number of individuals (i.e., possible test data) in order to cover the same target paths that Pei covered. Please note that since Pei's approach works on a single target path at a time. For a fair comparison, we conducted the experiment having one target path at a time. We have also conducted another experiment having all three target paths at one time. The results are shown in Table 6.

Row no. 4 of the table shows our generator was able to find all of them within seven generations based on the average of three runs. It is also worth noting that Table 6 shows that the number of generations needed to cover a certain path depends on its level of difficulty; since Pei reported that Path 3 is the most difficult one [18].

Table 7 depicts the detailed results of the different runs of our test generator, over 20 runs; trying to cover the three most difficult target paths in Pei's work either by a single path or all paths at a time. This table is meant to show the consistency of the results.

On the average, the larger population sizes the smaller number of generations required to find the target path(s).

Table 8, below, shows the results of our generator, based on 20 runs, in trying to cover only the "equilateral" target path. The table also shows the results when trying to satisfy all target paths at a time.

On the average, the larger the population sizes the smaller number of generations required to find the target path(s), which is also supported by smaller standard deviation.

7. Conclusion and future work

In this paper, we have presented a GA-based test data generator that is capable of generating multiple test data to cover multiple target paths in one run. We have demonstrated the capabilities of the proposed approach through empirical validation, and compared a number of variations of the proposed generator.

The experimental results show that our test data generator is more effective and more efficient than existing generators; due to the fact that it allows covering multiple target paths with less number of test data examined.

However, the following are concerns that limit the generator's scalability and usability:

1. Manual CFG construction takes a considerable amount of time to do.
2. Manual target paths identification requires tester creativity.
3. The process of manual program instrumentation is programming language dependent, which must be done carefully such that it does not change the semantic of the program. Hence, manual instrumentation requires a considerable amount of work and time.
4. With regard to predicate-wise traversal, our fitness function does not consider the matched *sub-paths* that have unmatched positions for a possible positive contribution to the fitness value. It only considers those sub-paths that have the same positions.

Future work will try to address the above limitations. Moreover, we will also try to investigate capabilities to allow automatic identification of potential infeasible program paths. Testing object oriented software will be another objective of our future research.

Acknowledgments

The authors wish to acknowledge King Fahd University of Petroleum and Minerals (KFUPM) for utilizing the various facilities in carrying out this research. Many thanks are due to the anonymous referees for their detailed and helpful comments.

References

[1] Beizer B. Software testing techniques. New York: Van Nostrand Reinhold; 1982.
[2] Roper M, Maclean I, Brooks A, Miller J, Wood M. Genetic algorithms and the automatic generation of test data. ⟨http://citeseer.ist.psu.edu/135258.html⟩, 1995.
[3] Whittaker JA. What is software testing? And why is it so hard? IEEE Software 2000; 17(1):70–9.
[4] Berndt DJ, Fisher J, Johnson L, Pinglikar J, Watkins A. Breeding software test cases with genetic algorithms. In: Proceedings of the thirty-sixth Hawai'i international conference on system sciences (HICSS-36), Hawaii; January 2003.
[5] Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithms. Software Engineering Journal 1996;11(5):299–306.
[6] Berndt DJ, Watkins A. Investigating the performance of genetic algorithm-based software test case generation. In: Proceedings of the eighth IEEE international symposium on high assurance systems engineering (HASE'04), University of South Florida; March 25–26, 2004. p. 261–2.
[7] Wegener J, Baresel A, Sthamer H. 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.
[8] Hermadi I. Genetic algorithm based test data generator. Master thesis, Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, Saudi Arabia; May 2004.
[9] Myers GJ. The art of software testing. New York: Wiley; 1979.
[10] Hamlet D. Foundations of software testing: dependability theory. Portland State University; 1994.
[11] Lin JC, Yeh PL. Using genetic algorithms for test case generation in path testing. In: Proceedings of the ninth Asian test symposium (ATS'00), Taipei, Taiwan; December 4–6, 2000.
[12] Edvardsson J. A survey on automatic test data generation. In: Proceedings of the second conference on computer science and engineering, Linkoping: ESCEL; October 1999. p. 21–8.
[13] McMinn P. Search-based software test data generation: a survey. Software testing, verification, and reliability, vol. 14 (2). New York: Wiley; 2004. p. 105–56.
[14] Gupta N, Mathur AP, Soffa ML. Automated test data generation using an iterative relaxation method. In: Foundations of software engineering. 1998. p. 231–44.
[15] Korel B. Automated software test data generation. IEEE Transactions on Software Engineering 1990;16(8):870–9.
[16] Goldberg DE. Genetic algorithms: in search, optimization and machine learning. Reading, MA: Addison-Wesley; 1989.
[17] Holland J. Adaptation in natural and artificial systems. Cambridge, MA: MIT Press; 1975.
[18] Pei M, Goodman ED, Gao Z, Zhong K. Automated software test data generation using a genetic algorithm. Technical Report, GARAGe of Michigan State University; June 1994.
[19] McGraw G, Michael C, Schatz M. Generating software test data by evolution. Technical Report, Reliable Software Technologies, Sterling, VA; February 9, 1998.
[20] Michael CC, McGraw GE, Schatz MA. Generating software test data by evolution. IEEE Transactions on Software Engineering 2001;27(12):1085–110.
[21] Michael CC, McGraw GG. Opportunism and diversity in automated software test data generation. Technical Report, Reliable Software Technologies, Sterling, VA; December 8, 1997.

[22] Michael CC, McGraw GE, Schatz MA, Walton CC. Genetic algorithms for dynamic test data generation. Technical Report, Reliable Software Technologies, Sterling, VA; May 23, 1997.
[23] Michael CC, McGraw GE, Schatz MA, Walton CC. Genetic algorithms for dynamic test data generation. In: Proceedings of the 12th IEEE international automated software engineering conference (ASE 97). Tahoe, NV, 1997. p. 307–8.
[24] Michael CC, McGraw GE. Automated software test data generation for complex programs. Technical Report, Reliable Software Technologies, Sterling, VA; 1998.
[25] Pargas RP, Harrold MJ, Peck PR. Test-data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability 1999;9(4):263–82.
[26] Bueno PMS, Jino M. Identification of potentially infeasible program paths by monitoring the search for test data. In: Proceedings of the fifteenth IEEE international conference on automated software engineering (ASE '00), Grenoble, France; 11–15 September 2000. p. 209–18.
[27] Wegener J, Buhr K, Pohlheim H. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In: Proceedings of the genetic and evolutionary computation conference, GECCO-2002, New York, USA; 9–13th July 2002.
[28] Hermadi I, Ahmed MA. Genetic algorithm based test data generator. In: Proceedings of the congress on evolutionary computation 2003 (CEC2003), Canberra, Australia; 8–12 December 2003.
[29] Sthamer HH, Wegener J, Baresel A. Using evolutionary testing to improve efficiency and quality in software testing. In: Proceedings of the second Asia-Pacific conference on software testing analysis and review, Melbourne, Australia; 22–24th July 2002.
[30] Sthamer HH. The automatic generation of software test data using genetic algorithms. PhD dissertation, University of Glamorgan; November 1995.
[31] Baresel A, Sthamer, H, Schmidt, M. 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.
[32] Belanche LA. A study in function optimization with the breeder genetic algorithm. LSI Research Report LSI-99-36-R, Universitat Polit'ecnica de Catalunya; 1999.
[33] Munteanu C, Lazarescu V, Radoi C. A new strategy in optimization using genetic algorithms. In: Proceedings of IEEE Melecon '98, vol. 1, 1998. p. 415–9. ISBN 0-7803-3879-0.