Hindawi Publishing Corporation
Computational Intelligence and Neuroscience
Volume 2014, Article ID 591294, 12 pages
http://dx.doi.org/10.1155/2014/591294

Research Article

Genetic Algorithm-Based Test Data Generation for Multiple Paths via Individual Sharing

Xiangjuan Yao[1] and Dunwei Gong[2]

[1] *College of Science, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China*
[2] *School of Information and Electrical Engineering, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China*

Correspondence should be addressed to Xiangjuan Yao; yxjcumt@126.com

Received 28 April 2014; Revised 16 September 2014; Accepted 19 September 2014; Published 16 October 2014

Academic Editor: Jianwei Shuai

The application of genetic algorithms in automatically generating test data has aroused broad concerns and obtained delightful achievements in recent years. However, the efficiency of genetic algorithm-based test data generation for path testing needs to be further improved. In this paper, we establish a mathematical model of generating test data for multiple paths coverage. Then, a multipopulation genetic algorithm with individual sharing is presented to solve the established model. We not only analyzed the performance of the proposed method theoretically, but also applied it to various programs under test. The experimental results show that the proposed method can improve the efficiency of generating test data for many paths' coverage significantly.

1. Introduction

One of the approaches to improve the quality of software is to do a large number of tests before delivery and usage in order to detect bugs or faults in software. Software testing is an expensive, tedious, and labor-intensive task and requires significant human effort [1]. If the process of testing can be automated, it will undoubtedly shorten the period of software development and improve the quality of software, so as to enhance the market competitiveness. One of the most important issues in automated software testing is the generation of effective test data satisfying the selected test adequacy criteria.

It has been proved that many software test problems can come down to those of generating test data for paths coverage [2, 3], which can be described as follows: for a given path of a program under test, search for a test datum in the input domain of the program, such that the traversed path of the test datum is just the desired one.

In recent years, it is becoming a promising direction to generate test data for complex software using the genetic algorithm (for short, GA) and has achieved many research results [4]. But most GA-based test data generation methods for path coverage intend to cover target paths one by one, which make the process of test data generation inefficient.

In this study, we established a mathematical model of generating test data for multiple paths coverage, which takes each optimization problem corresponding to one target path as a subproblem, and a number of subproblems form an overall optimization problem. This model is different from those existing multiobjective problems due to the specificity of generating test data.

On this basis, we proposed a multipopulation genetic algorithm to solve the proposed optimization problem. In our algorithm, each subpopulation optimizes one subproblem, so the fitness functions of different subpopulations differ from each other. All subpopulations evolve in parallel. A very key step of our algorithm is the individual sharing of different subpopulations; specifically, every time when the evolutionary operations of a generation finish, the algorithm not only determines whether an individual is an optimal solution of the subpopulation it belongs to, but also does that for the other subpopulations. By this way, the efficiency of finding optimal solutions for each subproblem improves with the complexity of the algorithm not increasing obviously.

We not only analyzed the performance of the proposed method theoretically, but also applied it to different programs under test for evaluation. The experimental results show that

the proposed method can significantly improve the efficiency of generating test data for many paths' coverage.

This paper is divided into nine sections, and the remainder is organized as follows: Section 2 briefly reviews the related works; Section 3 gives a model of generating test data for multiple path coverage; a multipopulation genetic algorithm is proposed to solve the model in Section 4; Section 5 analyzes the performance of the proposed algorithm theoretically; the experiments are presented in Section 6; Section 7 discusses possible threats to the validity of the proposed method; finally, conclusion is presented in Section 8.

2. Related Work

This section provides a survey on GA-based software testing. First, some basic methods of automatic software testing are introduced. Then, we review the main works on GA-based test data generation. Finally, we talk about the challenges of path coverage testing.

2.1. Automatic Software Testing. Since the process of software testing is highly time and resource consuming, many automatic approaches have been developed to facilitate the process and decrease its cost, which can be divided into four categories, namely, random method, static method, dynamic method, and heuristics method.

Random method generates test data by randomly sampling the input space of a program under test [5]. This approach is simple but has certain blindness in generating test data. Some improved methods have been proposed to heighten the diversity of test data [6, 7].

Static method only needs static analysis and transformation, without involving actual execution of the program under test, such as symbolic execution [2, 8], and domain reduction [9]. But this method usually requires a significant amount of algebra and (or) interval arithmetic [10].

Dynamic structural method of generating test data was firstly proposed by Miller and Spooner [11], which needs real execution of the program under test, in order to obtain useful information [3].

Different from dynamic method, the process of generating test data by heuristics method is not completely determined. Heuristics method usually recurs to some sort of heuristic algorithms, such as the genetic algorithm, simulated annealing, tabu search, and scatter algorithm, of which the GA is the most widely used [12].

2.2. GA-Based Test Data Generation. As an efficient search-based optimization algorithm, the GA shows special advantage and efficiency in solving problems with high complexity, such as the problems of large space, multipeak, and nonlinear. Therefore it has become a research hotspot to automatically generate test data with GAs and produced encouraging results [13].

Gong and Yao [14] used a GA to generate test data for statement coverage based on testability transformation. Yao et al. [15] proposed an approach to reduce target statements according to their dominant relations and the test suite covering the reduced set of target statements was generated by a GA.

Miller et al. [16] used GAs to generate test data satisfying branch coverage criterion. The experimental results show that the test suite obtained by GAs can achieve or be very close to branch coverage. Baars et al. [17] presented an algorithm for constructing fitness functions that improve the efficiency of search-based testing when trying to generate branch adequate test data. Alshraideh et al. [18] proposed a multiple-population algorithm to improve the efficiency of branch coverage testing. The experimental results showed that the proposed method outperforms the single-population algorithm significantly.

Michael et al. [19] used a GA to generate test data satisfying condition coverage criterion. In their work, the problem of test data generation is reduced to a function minimization, and the function is minimized using one of two genetic algorithms in place of the local minimization techniques.

As for the works of GA-based software testing for path coverage criterion, we will introduce them individually in Section 2.3.

Besides traditional structural software testing, Bühler and Wegener [20] applied an evolutionary algorithm to functional testing. Watkins and Hufnagel [21] used two GAs to generate a couple of test data pieces and then trained a decision tree using them, in order to obtain an agent model which distinguishes the merit of test data. Ferrer et al. [22] presented a method of automatically generating test data by considering multiple objectives: maximizing the coverage and minimizing the oracle cost.

2.3. GA-Based Path Testing. Path coverage testing is the strongest sufficiency criterion in white box testing. Automatically generating data for paths coverage remains a challenging problem [23].

Bueno and Jino [24] and Watkins and Hufnagel [25] used a GA to obtain test data fulfilling path coverage, respectively. Mei and Wang [26] proposed a method that can automatically generate test cases for selected paths using a special genetic algorithm. In their algorithm, the best chromosome called queen crosses with the selected drones, which enhances the exploitation of global optimal solutions.

Hermadi and Ahmed [27] have observed that existing GA-based test data generators can generate only one test datum for one test goal at a time. When there are many target paths to be covered, the generator has to be run many times. In fact, the generated individuals when trying to find test data covering a path may be just test data covering other target paths. This, hence, makes those existing test data generators inefficient in trying to generate test data for multiple paths.

Wegener et al. [28] developed a fully automatic GA-based test data generator for structural software testing. In their approach, all generated individuals are evaluated with regard to all unachieved partial aims. Partial aims reached by chance are identified, and the individuals with good fitness values for one or more partial aims are noted and stored for seeding the subsequent testing of uncovered targets. But they only considered one partial aim for optimization at a time, which means that they solved the problems of generating test

data one by one. Furthermore, they did not discuss whether multiple targets can be covered in one run. Besides, they reported that full coverage of some programs is achieved but not for all programs though.

Bueno and Jino [24] looked after methods to improve the performance of test data generation by using past input data to compose the initial population for the search. Although these methods can improve the performance of the initial population by reusing test data, they still cannot make full use of the test data generated in the evolutionary process.

Ahmed and Hermadi [29] proposed a GA-based test data generator for multiple paths. In their work, the problem of generating test data for multiple paths is regarded as a multiobjective optimization problem and solved by a multiobjective evolutionary algorithm. In fact, the problem of generating test data for multiple paths is strictly different from traditional multiobjective optimization problems. Therefore, it is necessary to establish an appropriate mathematical model for the problem of generating test data for multiple paths coverage according to its specificity and give a corresponding evolutionary solution.

Gong and Zhang [30] also proposed a test data generation method for multipath coverage. They represent a target path using Huffman encoding method and designed the fitness function according to the Huffman codes of target paths. Their method is simple and has better performance than Ahmed's method, but the fitness function cannot distinguish individuals well.

In order to stop searching as soon as all feasible paths have been covered, Hermadi et al. [31] proposed method for determining when it is no longer worthwhile to continue searching for test data to cover uncovered target paths. Compared to searching for a standard number of generations, an average of 30–75% of total computation was avoided in test programs with infeasible paths, and no feasible paths were missed due to early termination. The extra computation in programs with no infeasible paths was negligible.

3. Mathematical Model of Test Data Generation for Multiple Paths

In order to illustrate conveniently, we first introduce several concepts. Then, an objective function is constructed in order to transform the problem of generating test data into an optimization one. On this basis, the optimization model of generating test data for multiple paths coverage is established.

3.1. Basic Concepts

Control Flow Graph (CFG) [1]. The CFG of a program Φ is a directed graph $G = (N, E, s, e)$, where N is the set of nodes, E is the set of edges, and s and e are unique entry and exit nodes of the graph, respectively. Each node n is a statement in the program; each edge (n_i, n_j) represents a transfer of control from node n_i to node n_j.

Path [1]. A path of a CFG is a sequence $P = n_1, n_2, \ldots, n_k$, such that there exists an edge from node n_i to $n_{i+1}, i = 1, 2, \ldots, k-1$.

For large-scale programs, the sequence of a path may be very long. We represent a path using a (0, 1)-string for simplicity. Suppose that there are m conditional statements in path P, denoted as $\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_m$. Define

$$\gamma_i = \begin{cases} 0, & P \text{ inludes the false branch of } \mathscr{C}_i \\ 1, & P \text{ inludes the true branch of } \mathscr{C}_i. \end{cases} \quad (1)$$

Thus we obtain a (0-1)-string $\gamma_1\gamma_2\cdots\gamma_m$ of length m. In program Φ, the mapping between a path and such a (0, 1)-string is one to one. Without special illustration, a path is represented by such a (0, 1)-string in this study.

Let the input vector of program Φ be $X = (x_1, x_2, \ldots, x_s)$, and let the domain of x_i be D_i; then the *input domain of* Φ is $D(\Phi) = D_1 \times D_2 \times \cdots \times D_s$. When program Φ adopts X as an input, the traversed path is denoted by $P(X)$. We call the first dissimilar character of P and $P(X)$ their *bifurcation*.

3.2. Structure of Objective Function. The key problem of applying GAs to test data generation is the construction of a suitable objective function. The goodness of a candidate test datum is often expressed in terms of the closeness that the test datum fulfills the test goal. The approach to forming an objective function typically involves two parts: *approach level* ($\mathscr{AL}$) and *branch distance* ($\mathscr{BD}$) [3, 24, 25].

The approach level assesses how close an execution comes to reaching the predicate which controls the test object. If $P \neq P(X)$, we define the approach level of input X to a target path P as the number of characters between the bifurcation of P and $P(X)$ to the last character of P, denoted by $\mathscr{AL}_P(X)$; otherwise, we define $\mathscr{AL}_P(X) = 0$. X covers path P if and only if $\mathscr{AL}_P(X) = 0$.

For example, suppose that $P = 1001001$ is a target path, $P(X_1) = 1001110$, and $P(X_2) = 1001001$; then $\mathscr{AL}_P(X_1) = 3$ and $\mathscr{AL}_P(X_2) = 0$.

The branch distance assesses how close the predicate comes to evaluating either true or false branch. For example, suppose that a conditional statement is "if $a \geq 12$," and the aim is to execute the true branch. Suppose that the value of a is $a(X)$ after the execution of this statement with input X; then the branch distance of X for branch condition $a \geq 12$ is defined as follows:

$$\mathscr{BD}(X, a \geq 12) = \begin{cases} 0 & \text{if } a(X) \geq 12, \\ 12 - a(X) & \text{others.} \end{cases} \quad (2)$$

Branch distances of different kinds of simple branch conditions are listed in Table 1. For a complex branch condition, branch distance is the composite of those of all simple conditions included in it, which is listed in Table 2.

We define the general objective function $f_P(X)$ of input X to target path P as follows:

$$f_P(X) = \mathscr{AL}_P(X) + \text{normalized}\left(\mathscr{BD}_P(X)\right), \quad (3)$$

where $\mathscr{BD}_P(X)$ refers to the branch distance of X to the conditional statement corresponding to the bifurcation of P and $P(X)$, and function

$$\text{normalized}(x) = 1 - 1.01^{-x}. \quad (4)$$

Function normalized maps the value of $\mathscr{BD}_P(X)$ to interval $[0, 1)$.

Table 1: Branch distances of simple branch conditions [3].

Branch condition	$a \geq b$	$a > b$	$a = b$	$a \neq b$
True	0	0	0	0
False	$b - a$	$b - a + 0.1$	$\lvert b - a \rvert$	1

Table 2: Branch distances of complex branch conditions [3].

Connection	Branch distance
α && β	$\mathscr{BD}(\alpha) + \mathscr{BD}(\beta)$
$\alpha \| \beta$	$\min(\mathscr{BD}(\alpha), \mathscr{BD}(\beta))$

A sufficient and necessary condition of $f_P(X) = 0$ is that the traversed path of X is P; that is, $P(X) = P$; furthermore, the smaller the value of $f_P(X)$, the nearer the X to the data covering P. So the problem of generating test data for path P can be transformed into that of minimizing $f_P(X)$.

For example, see the program in Figure 1. Suppose that the target path is $P = s\ 1\ 2\ 3\ 4\ 5\ e$. There are three conditional statements in P, that is, statements 1, 2, and 4, respectively. P traverses the true branches of all these statements. So we also write $P = 111$. Suppose that $X_1 = (-10, 5, 10)$, $X_2 = (2, 5, 10)$, and $X_3 = (2, 1, 10)$. We obtain $P(X_1) = 01$, $P(X_2) = 101$, and $P(X_3) = 101$. Thus $\mathscr{AL}_P(X_1) = 3 - 0 = 3$, $\mathscr{AL}_P(X_2) = 3 - 1 = 2$, and $\mathscr{AL}_P(X_3) = 3 - 1 = 2$.

In addition, $P(X_1)$ deviates P from the first conditional statement, so the branch distance of X_1 to P is $\mathscr{BD}_P(X_1) = 11$; similarly, we get $\mathscr{BD}_P(X_2) = 5$ and $\mathscr{BD}_P(X_3) = 1$. Thus

$$\begin{aligned} f_P(X_1) &= 3 + 1 - 1.01^{-11} = 3.1037, \\ f_P(X_2) &= 2 + 1 - 1.01^{-5} = 2.0485, \\ f_P(X_3) &= 2 + 1 - 1.01^{-1} = 2.0099. \end{aligned} \tag{5}$$

Although the traversed paths of X_2 and X_3 are the same, the branch distance of X_3 is smaller than that of X_3. Thus X_3 obtains a better objective value than X_2.

3.3. Mathematical Model of Generating Test Data for Multiple Paths Coverage. Let the set of target paths be $\Gamma = \{P_1, P_2, \ldots, P_n\}$; then the problem of generating test data for Γ can be described as follows: find a test suite $\{X_1, X_2, \ldots, X_n\}$, such that $P(X_i) = P_i$. Let the objective function for path P_i using the method proposed in Section 3.2 be $f_i(X)$; then the problem of generating test data for $\{P_1, P_2, \ldots, P_n\}$ can be transformed into an optimization one described as follows:

$$\begin{aligned} &\min \quad f_1(X) \\ &\text{s.t.} \quad X \in D(\Phi) \\ &\min \quad f_2(X) \\ &\text{s.t.} \quad X \in D(\Phi) \\ &\qquad \vdots \\ &\min \quad f_n(X) \\ &\text{s.t.} \quad X \in D(\Phi). \end{aligned} \tag{6}$$

Most existing GA-based test data generation methods take the above problem as n self-governed optimization ones and solve them one by one. Specifically, for each optimization problem $\min f_i(X)$, run a GA in order to find an optimal solution of $f_i(X)$, which is just a test datum traversing target path P_i. Repeat above process, until all optimization problems have been solved. If the number of target paths is n, the GA has to be run n times.

This approach, 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, because different target paths have similarities. Therefore the efficiency of these methods is low when n is large.

Ahmed et al. gave an algorithm of generating test data for multiple paths coverage, but they regarded this problem as a multiobjective optimization one. Thus, their model should be

$$\begin{aligned} &\min \quad F(X) = (f_1(X), f_2(X), \ldots, f_n(X)) \\ &\text{s.t.} \quad X \in D(\Phi). \end{aligned} \tag{7}$$

In fact, the problem of generating test data for multiple paths is strictly different from traditional multiobjective optimization ones. In traditional multiobjective optimization problems, the aim is to find one solution which satisfies all objectives well. In a multiobjective environment, we often encounter conflicting objectives with some trade-off among them. But for the problem of generating test data for paths $P_1, P_2, \ldots, P_n$, what we need is to obtain a test suite $\{X_1, \ldots, X_n\}$, where X_i is an optimal solution of $f_i(X)$, $i = 1, \ldots, n$.

In addition, the number of objective functions in traditional multiobjective optimization problems remains unchanged, while that in the proposed model gradually reduces. Therefore, there is much limitation to take the problem of generating test data as a multiobjective optimization one.

Different from existing methods, we consider the problem of generating test data for n paths coverage as a uniform problem, in which each optimization problem corresponding to one target path is a subproblem. We solve all subproblems at the same time. Thus the problem corresponding to the test data generation for multiple paths coverage can be described as follows:

$$\begin{aligned} &\min \quad f_1(X_1) \\ &\min \quad f_2(X_2) \\ &\qquad \cdots \\ &\min \quad f_n(X_n) \\ &\text{s.t.} \quad X_1, X_2, \ldots, X_n \in D(\Phi). \end{aligned} \tag{8}$$

This model includes n subproblems, each of which is a minimization problem, and all objective functions have

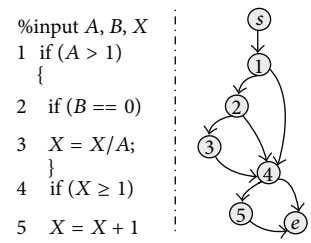

Figure 1: An example program and its CFG.

the same domain. We will seek an algorithm to solve these n problems simultaneously, rather than solve them independently. So problem (8) strictly differs from (6) and (7), and we should seek a suitable method to solve it.

4. Multipopulation GA for Test Data Generation of Multiple Paths

In this section we will give a multipopulation GA to solve problem (8), which is different from traditional multipopulation GAs. The main purpose of our strategy is to expand the search range of each population by individual sharing, so as to improve the efficiency of the algorithm.

4.1. Initialization of Populations. For the ith optimization problem $\min f_i(X_i)$, randomly generating a subpopulation of size m, that is, $\aleph^{(1)}(P_i) = \{X_{i1}^{(1)}, X_{i2}^{(1)}, \ldots, X_{im}^{(1)}\}$, $i = 1, \ldots, n$, where $X_{ij}^{(1)}$ refers to the jth individual in the ith population of the first generation. An individual corresponds to a string by proper encoding. Population size might have some influence on the performance of the algorithms, but this is not a focus of this study, so we just give an appropriate value for it.

4.2. Genetic Operations. As a typical GA, our method mainly includes three kinds of operations, that is, selection, crossover, and mutation.

Individuals are selected according to their fitness, so that good gens have more chances to be copied to the next generation. We adopt objective function $f_i(X_{ij}^{(1)})$ as the fitness of individual $X_{ij}^{(1)}$. Because what we are solving are minimization problems, the smaller the fitness of an individual is, the better we consider it.

Crossover operation exchanges parts of two gene strings in a certain probability to produce two new chromosomes, while mutation operation modifies some of the genes in a string, resulting in a new chromosome. The crossover and mutation rates are denoted by P_c and P_m, respectively. Because parameter setting is not the focus of this work, we just give the value of the parameters based on experience.

Each subpopulation implements these operations independently. By this way, individuals of the tth generation are evolved to the $(i + 1)$th generation, which can be shown as Figure 2.

4.3. Individual Sharing among Different Subpopulations. The biggest difference between traditional multipopulation GAs and the proposed one lies in the following: in traditional multipopulation GAs, subpopulations communicate by means of individual migration, while in our method, subpopulations communicate through individual sharing among subpopulations. Specifically, every time when the evolutionary operations of a generation finish, the algorithm not only determines whether an individual is an optimal solution of the subpopulation it belongs to, but also does that for the other subpopulations. In this way, the individuals of one subpopulation are shared by all other subpopulations, and the probability of finding optimal solutions significantly increases. The implementation of individual sharing is shown as Figure 3.

Because $\mathscr{AL}_p(X) = 0$ if and only if the traversed path of X is just P, we determine whether X is a desired test datum covering P according to the value of $\mathscr{AL}_P(X)$. Suppose that there are n target paths $P_1, \ldots, P_n$. In our algorithm, we can obtain the values of $\mathscr{AL}_{P_1}(X), \ldots, \mathscr{AL}_{P_n}(X)$ in one run of the instrumented program with input X. Thus the individual sharing can be realized with the computation complexity not increasing too much.

4.4. Steps of the Algorithm. Based on the above discussion, the main steps of the proposed algorithm are shown as follows.

Step 1. Set the values of the number of subpopulations n, maximum termination generation T, crossover probability P_c, and mutation probability P_m, where n is equal to the number of target paths.

Step 2. Suppose that the set of target paths is $\{P_1, P_2, \ldots, P_n\}$. For P_i, randomly generate a subpopulation $\aleph^{(1)}(P_i) = \{X_{i1}^{(1)}, X_{i2}^{(1)}, \ldots, X_{im}^{(1)}\}$, $i = 1, 2, \ldots, n$. The value of generation $t = 1$.

Step 3. For subpopulation $\aleph^{(t)}(P_i)$ in the tth generation, calculate the values of

$$\mathscr{AL}_{P_1}\left(X_{ij}^{(t)}\right), \ldots, \mathscr{AL}_{P_n}\left(X_{ij}^{(t)}\right) \tag{9}$$

for individual $X_{ij}^{(t)}$ to all target paths and those of $\mathscr{BD}_{P_i}(X_{ij}^{(t)})$ for individual $X_{ij}^{(t)}$ to path P_i, $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, m$.

Step 4. $f_i(X_{ij}^{(t)}) = \mathscr{AL}_{P_i}(X_{ij}^{(t)}) + \text{normalized}(\mathscr{BD}_{P_i}(X_{ij}^{(t)}))$ is used as the fitness of individual $X_{ij}^{(t)}$ for subpopulation $\aleph^{(t)}(P_i)$ to guide the evolution.

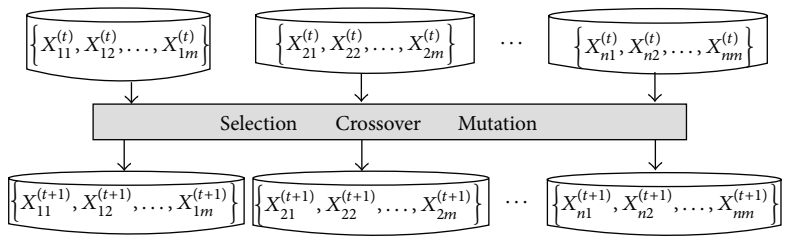

Figure 2: Evolution of individuals in our multipopulation genetic algorithm.

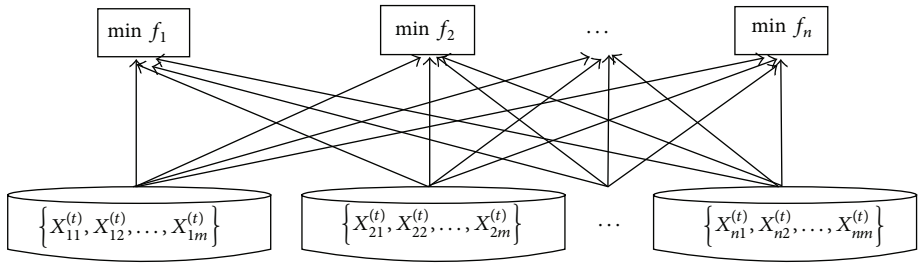

Figure 3: Individual sharing among subpopulations.

Step 5. If there is a $\mathscr{AL}_{P_k}(X_{ij}^{(t)}) = 0$, which means that $X_{ij}^{(t)}$ covers P_k, then $X_{ij}^{(t)}$ is an optimal solution of the kth optimization subproblem. In this case, save $X_{ij}^{(t)}$, delete P_k from the target path set, and terminate the evolution of the kth subpopulation.

Step 6. If the number of subpopulations becomes 0, or the number of generations is larger than T, then stop the evolution and output the test data; otherwise, go to Step 7.

Step 7. Perform genetic operations on $\aleph^{(t)}(P_i)$ to generate offspring population $\aleph^{(t+1)}(P_i)$. let $t = t + 1$ and go to Step 3.

5. Performance Analysis

We will illustrate the performance of the proposed algorithm by analyzing its efficiency and computational complexity.

5.1. Efficiency of Algorithm. Suppose that the set of target paths is $\{P_1, P_2, \ldots, P_n\}$ $(n > 1)$ and $\aleph(P_i)$ is the subpopulation used to optimize the ith subproblem, which is related to the problem of generating test data for P_i. Let T_i be the number of generations in which the ith subpopulation finds the test datum covering path P_i; thus T_i is a random variable. From experiences, we can suppose that $T_i \sim N(\mu_i, \sigma_i^2)$. Let T_{ij}, $i \neq j$, be the number of generations in which the ith subpopulation finds the test datum covering P_j; then T_{ij} is also a random variable. Suppose that the probability of $\aleph(P_i)$ finding the test datum that covers P_j is λ_j; then $P\{T_{ij} = t\} = (1 - \lambda_j)^{t-1}\lambda_j$, $t = 1, 2, \ldots$. For convenience to illustration, we also denote T_i by T_{ii}.

If we use traditional single-objective GAs to solve (3), in the circumstance of using the same population size, the probability of $\aleph(P_i)$ finding an optimal solution within t generations is $P\{T_i \leq t\} = \Phi((t - \mu_i)/\sigma_i)$, where $\Phi(x)$ is the distribution function of standard normal distribution. Thus the probability of all subpopulations finding their optimal solutions within t generations is

$$\begin{aligned}\Gamma_1(t) &= P\{T_1 \leq t, \ldots, T_n \leq t\} \\ &= P\{T_1 \leq t\} \cdots P\{T_n \leq t\} \\ &= \prod_{i=1}^{n} \Phi\left(\frac{t - \mu_i}{\sigma_i}\right).\end{aligned} \tag{10}$$

If we adopt the proposed method to solve (6), then the probability of finding the test datum covering path P_i within t generations is

$$\begin{aligned}&P\{\min(T_{1i}, \ldots, T_{ni}) \leq t\} \\ &\quad = 1 - P\{\min(T_{1i}, \ldots, T_{ni}) > t\} \\ &\quad = 1 - P\{T_{1i} > t\} \cdots P\{T_{ni} > t\} \\ &\quad = 1 - (1 - P\{T_{1i} \leq t\}) \\ &\qquad \times \cdots \times (1 - P\{T_{ni} \leq t\}) \\ &\quad = 1 - \left[1 - \Phi\left(\frac{t - \mu_i}{\sigma_i}\right)\right](1 - \lambda_i)^{t(n-1)}.\end{aligned} \tag{11}$$

Thus the probability of all subpopulations finding all optimal solutions within t generations is

$$\Gamma_2(t) = \prod_{i=1}^{n} \left\{1 - \left[1 - \Phi\left(\frac{t - \mu_i}{\sigma_i}\right)\right](1 - \lambda_i)^{t(n-1)}\right\}. \tag{12}$$

Since $(1-\lambda_i)^{t(n-1)} < 1$, we obtain

$$\begin{aligned}\Gamma_2(t) &> \prod_{i=1}^{n}\left\{1-\left[1-\Phi\left(\frac{t-\mu_i}{\sigma_i}\right)\right]\right\}\\ &= \prod_{i=1}^{n}\Phi\left(\frac{t-\mu_i}{\sigma_i}\right) = \Gamma_1(t).\end{aligned} \tag{13}$$

That is to say, the probability of finding all optimal solutions using the proposed algorithm is larger than that of traditional single-objective GAs. In addition, the more the number of target paths is, the more obvious the advantage of the proposed method is, which can also be easily understood by the following example.

Suppose that the set of target paths is $\{P_1, \ldots, P_5\}$ and $T_i \sim N(500, 100^2)$, $\lambda_j = 1/10000$, $i, j = 1, \ldots, 5$; then the probabilities of finding all optimal solutions within 500 and 600 generations using traditional single-objective GAs are

$$\begin{aligned}\Gamma_1(500) &= \prod_{i=1}^{5}\Phi\left(\frac{500-500}{100}\right) = 0.5^5 = 0.0313,\\ \Gamma_1(600) &= \prod_{i=1}^{5}\Phi\left(\frac{600-500}{100}\right) = 0.8143^5 = 0.3580,\end{aligned} \tag{14}$$

respectively, whereas the probabilities of finding all optimal solutions within 500 and 600 generations using the proposed algorithm are

$$\begin{aligned}\Gamma_2(500) &= \prod_{j=1}^{5}\left\{1-\left[1-\Phi\left(\frac{500-500}{100}\right)\right]\right.\\ &\qquad \left.\times\left(1-\frac{1}{10000}\right)^{500\times(5-1)}\right\}\\ &= 0.5906^5 = 0.0719,\\ \Gamma_2(600) &= \prod_{j=1}^{5}\left\{1-\left[1-\Phi\left(\frac{600-500}{100}\right)\right]\right.\\ &\qquad \left.\times\left(1-\frac{1}{10000}\right)^{600\times(5-1)}\right\}\\ &= 0.8539^5 = 0.4540,\end{aligned} \tag{15}$$

respectively. If the number of target paths increases to 10, and $T_i \sim N(500, 100^2)$, $\lambda_j = 1/10000$, $i, j = 1, \ldots, 10$, then the probabilities of finding all optimal solutions within 500 and 600 generations using traditional single-objective GAs are

$$\begin{aligned}\Gamma_1(500) &= \prod_{i=1}^{10}\Phi\left(\frac{500-500}{100}\right) = 0.5^{10} = 0.000977,\\ \Gamma_1(600) &= \prod_{i=1}^{10}\Phi\left(\frac{600-500}{100}\right) = 0.8143^{10} = 0.1282,\end{aligned} \tag{16}$$

respectively, whereas the probabilities of finding all optimal solutions within 500 and 600 generations using the proposed algorithm are

$$\begin{aligned}\Gamma_2(500) &= \prod_{j=1}^{10}\left\{1-\left[1-\Phi\left(\frac{500-500}{100}\right)\right]\right.\\ &\qquad \left.\times\left(1-\frac{1}{10000}\right)^{500\times(10-1)}\right\}\\ &= 0.6812^{10} = 0.0215,\end{aligned} \tag{17}$$

$$\begin{aligned}\Gamma_2(600) &= \prod_{j=1}^{10}\left\{1-\left[1-\Phi\left(\frac{600-500}{100}\right)\right]\right.\\ &\qquad \left.\times\left(1-\frac{1}{10000}\right)^{600\times(10-1)}\right\}\\ &= 0.8918^{10} = 0.3182,\end{aligned} \tag{18}$$

respectively. As can be seen from these results, in circumstance with 5 target paths, the probabilities of finding all optimal solutions within 500 and 600 generations using the proposed algorithm are 0.0719 and 0.4540, respectively, which are $0.0719/0.0313 \approx 2.3$ and $0.4540/0.3580 \approx 1.3$ times those of traditional single-objective GAs; in circumstance with 10 target paths, the probabilities of finding all optimal solutions within 500 and 600 generations using the proposed algorithm are 0.0215 and 0.3182, respectively, which are $0.0215/0.000977 \approx 22$ and $0.3182/0.1282 \approx 2.5$ times those of traditional single-objective GAs. The above results forcefully illuminate that the proposed algorithm is more efficient than traditional single-objective GAs; moreover, with the increase of the number of target paths, the advantages become more obvious.

5.2. Computational Complexity. We will compare the computational complexity of our multipopulation genetic algorithm and those of traditional ones. Suppose that the program under test has l executable statements and there are n target paths $\{P_1, P_2, \ldots, P_n\}$. The population size is m. Because m can be set manually, we consider m as a constant. We take the number of executed statements for the calculation of individual fitness and individual sharing in a generation as a measure of the computational complexity of an algorithm.

If we use traditional multipopulation GAs to solve the problem, which means that there is no individual sharing among subpopulations, then the program under test will be run nm times, which is equal to the number of all individuals. Since each run of the program probably executes l statements, the number of executed statements for the run of the program under test will be lmn. Taking the computation of the fitness as one statement, then all these mn individuals need to execute mn statements. So the number of executed statements in a generation using traditional multipopulation GAs is $C(l, n) = lmn + mn$.

If we use the proposed method to solve the problem, which means that subpopulations share all individuals, in addition to the run of the program under test and the computation of the fitness function, we consider the computation

Table 3: Description of subject programs.

Program	Loc	Description	Number of targets
Insertion sort	16	Array sorting	4
Bubble sort	18	Array sorting	4
Triangle	20	Return the type of a triangle	4
Binary search	37	Key number searching	7
Gcd	55	Compute greatest common divisor	20
Look	135	Find words in the system dictionary or lines	30
Comm	145	Select or reject lines common	30
Cal	160	Print a calendar for a specified year or month	30
Controller	172	Internal states regulation	30
Tcas	173	Altitude separation	30
Col	275	Filter reverse paper motions	50
Spline	289	Interpolate smooth curve	50
Tot_info	365	Statistics computation	50
Schedule2	374	Priority scheduler	50
Printtok	400	Lexical analyzer	50
Schedule	412	Priority scheduler	50
Replace	564	Pattern replace	100
Barcode	672	Barcode maker	100

due to individual sharing among subpopulations. Taking the computation of the approach level as a statement, individual sharing needs to execute mn^2 sentences. So the number of executed sentences in a generation using the proposed method is $C'(l,n) = lmn + mn + mn^2$. Under normal circumstances, n is much smaller than l, so $lmn+mn+mn^2 \ll 2lmn + mn$. Thus

$$\frac{C'(l,n)}{C(l,n)} = \frac{lmn + mn + mn^2}{lmn + mn} \leq \frac{2lmn + mn}{lmn + mn} < 2. \quad (19)$$

On the other hand, each subpopulation has m individuals as possible solutions for each generation in traditional multipopulation GAs. But in our method, the possible solutions become mn for each generation via individual sharing, which is n times that of traditional methods. In other words, the population size is magnified to n times via individual sharing with the computation quantity almost doubling.

6. Experiments

A group of experiments are conducted so as to investigate the performance of the proposed method. In the following section, subject programs are first introduced. Afterwards, experimental design is characterized. Finally, empirical results are presented and discussed.

6.1. Subject Programs. In order to evaluate the proposed method, we select eighteen programs for experiments. Table 3 shows some basic information of each program, including its name, size, and description. Table 3 is sorted by the sizes of the programs. These test subjects include not only laboratory programs, but also nontrivial industry ones. In addition, their lengths and functions are different from each other. These programs have been thoroughly used by other researches in the literature of software testing and analysis [19, 32–34]. The number of target paths for each program is also listed in Table 3.

For each program under test, we just randomly choose a part of feasible paths to cover. If there are too many paths to be covered, we can divide them into several groups, so that the scale of paths is reasonable. In addition, if we choose infeasible paths as target ones, the performances of different methods will not be distinguished, because it is impossible for any method to generate test data covering infeasible paths. The prediction of the infeasibility of a program path is an undecidable problem, and heuristic techniques that automatically select likely feasible paths can be employed [32].

6.2. Experimental Design. When designing the experiment, we specially have concern about two issues that can be described as follows.

Proposition 1. *Can individual sharing improve the efficiency of the algorithm?*

In order to verify the first proposition, we conduct two groups of experiments. In the first group of experiments, we use the proposed multipopulation GA with individual sharing to generate test data, while in the second one, different populations do not implement individual sharing but evolve independently.

Proposition 2. *How is the overall performance of the proposed method?*

In order to validate the overall performance of the proposed method in this study (for short, our method), we compare it with other three methods, namely, Gong's method

Table 4: Parameter settings.

Parameter	Value
Population size	300
Selection operator	Roulette wheel
Crossover operator	One-point crossover
Crossover rate	0.9
Mutation operator	One-point mutation
Mutation rate	0.3
Maximum generation	50000
Encoding style	Binary
Variable range	[0, 1023]

[30], Ahmed's method [29], and random method. The reason why we adopt Gong's and Ahmed methods to compare is that they are also about the problem of generating test data for multiple paths. In addition, random method is a basic test technique and has been widely used, so we also adopt it as a consult object.

All methods (except random one) apply the same values of parameters, which are listed in Table 4. There are two termination criteria: one is that the test data for all target paths have been found; the other is that the number of generations has reached the maximum.

6.3. Experimental Results. In each group of experiments, we performed 30 runs for each program under test and record the time consumption of each run and each method, where the time consumption refers to the time needed to generate test data covering all target paths.

6.3.1. Experimental Results for Testing the Performance of Individual Sharing. The experimental results to test the performance of individual sharing are listed in Table 5, in which Ave. and S.D. are the sample average and standard deviation of time consumption for each program and method, respectively. Sh.R. means the radio of the number of test data obtained by individual sharing and the number of all test data.

It can be seen from Table 5 that, (1) for all subject programs, the average time consumption using the method of individual sharing is all less than that not implementing individual sharing. The least time consumption of the method applying individual sharing is 6.38 seconds (Bubble Sort), while that not implementing individual sharing for the same program is 11.03 seconds. The most time consumption of the method applying individual sharing is 183.53 seconds (barcode), while that of the second method is 265.72 seconds. (2) The sharing rates of all programs exceed 30% except schedule (29.8%). The average sharing rate of the eight programs is 36.87%, which means that approximately one of each three test data pieces is obtained by individual sharing. By this way, we can make more full use of individuals generated in evolutionary process, therefore improving the efficiency of generating test data.

We use hypothesis testing to give a more scientific analysis for the above experimental results. Let X_1 and X_2 denote the time consumption using and not using individual sharing, respectively (without confusion, we will use the same symbol for all programs under test). It can be verified that X_1 and X_2 are random variables obeying normal distribution. Suppose that $X_i \sim N(\mu_i, \sigma_i^2)$, $i = 1, 2$. Because the sample standard deviation is an unbiased estimate of the standard deviation of the population, we take the values of sample standard deviations as those of standard deviations. Let the significance level $\alpha = 0.01$. We will illustrate the performances

Table 5: Number of evaluations and success rate of different fitness functions.

Programs	Sharing			No sharing	
	Ave. (s)	S.D.	Sh.R (%)	Ave. (s)	S.D.
Insertion sort	17.62	4.24	34.6	24.74	5.91
Bubble sort	**6.38**	**3.72**	**45.6**	**11.03**	**4.93**
Triangle	12.92	5.72	41.3	20.83	7.73
Binary search	39.82	8.18	32.4	48.49	10.93
Gcd	32.74	7.25	36.7	43.83	8.89
Look	16.26	6.32	31.8	22.63	7.83
Comm	30.77	11.86	45.6	50.35	16.37
Cal	39.27	9.06	39.3	55.92	12.83
Controller	46.72	11.72	33.9	63.52	12.89
Tcas	38.62	9.29	32.7	46.83	10.94
Col	75.61	16.72	35.3	103.73	24.65
Spline	50.72	17.75	37.8	69.93	23.89
Tot_info	79.92	20.85	37.9	121.78	26.78
Schedule2	73.88	24.25	31.8	97.62	30.57
Printtok	36.83	15.73	33.7	48.29	19.82
Schedule	45.74	17.99	29.8	65.23	18.74
Replace	141.23	44.27	41.3	176.34	42.87
Barcode	183.53	39.83	39.4	265.72	47.31

Table 6: Values of statistic U_1 and U_2 of object programs.

Programs	Values of U
Insertion sort	−5.36
Bubble sort	−4.12
Triangle	−4.51
Binary search	−3.48
Gcd	−5.30
Look	−3.47
Comm	−5.31
Cal	−5.81
Controller	−5.28
Tcas	−3.13
Col	−5.17
Spline	−3.54
Tot_info	−6.76
Schedule2	−3.33
Printtok	−2.48
Schedule	−4.11
Replace	−3.12
Barcode	−7.28

Table 7: Number of evaluations and success rate of different fitness functions.

Programs	Our method		Gong's method		Ahmed's method		Random method		U_1	U_2	U_3
	Ave. (s)	S.D.	Ave. (s)	S.D.	Ave. (s)	S.D.	Ave. (s)	S.D.			
Insertion sort	19.52	5.07	24.72	6.08	27.93	6.86	42.89	10.82	−3.6	−5.40	−10.71
Bubble sort	5.85	4.29	8.79	4.15	9.85	3.93	12.32	4.17	−2.7	−3.77	−5.92
Triangle	13.58	5.29	17.94	5.12	20.82	7.73	28.73	8.72	−3.24	−4.23	−8.14
Binary search	36.04	10.82	43.24	9.08	54.84	10.93	59.61	13.36	−2.79	−6.70	−7.51
Gcd	34.82	9.93	42.79	9.67	51.37	8.72	72.71	13.49	−3.15	−6.86	−12.39
Look	18.02	7.88	22.94	7.26	25.72	9.79	31.82	10.9	−2.52	−3.36	−5.62
Comm	33.92	12.89	39.74	16.23	50.35	17.48	48.28	15.51	−1.54	−4.14	−3.90
Cal	35.92	10.02	60.93	13.19	60.52	13.07	82.73	15.33	−8.27	−8.18	−14.00
Controller	49.93	12.78	62.19	12.98	71.81	13.83	90.26	15.53	−3.69	−6.36	−10.98
Tcas	35.92	9.95	42.85	9.74	52.89	11.44	77.28	19.53	−2.73	−6.13	−10.34
Col	69.01	15.94	84.23	18.24	112.56	23.78	108.92	26.72	−3.44	−8.33	−7.03
Spline	53.88	17.23	63.99	21.43	78.81	21.84	89.51	20.84	−2.01	−4.91	−7.22
Tot_info	84.27	21.89	114.23	23.14	125.83	29.78	147.34	32.25	−5.15	−6.16	−8.86
Schedule2	78.83	22.51	93.66	26.69	110.73	35.75	135.37	46.31	−2.33	−4.14	−6.01
Printtok	30.72	16.92	35.74	17.83	56.83	20.83	72.75	21.56	−1.12	−5.33	−8.40
Schedule	44.93	15.84	56.84	19.23	70.72	23.82	107.32	21.78	−2.62	−4.94	−12.69
Replace	137.38	40.03	160.98	40.92	188.78	51.78	205.27	48.9	−2.26	−4.30	−5.89
Barcode	192.82	40.17	224.87	49.23	287.67	56.13	316.42	68.73	−2.76	−7.53	−8.50

of different methods by comparing $\mu_1(= E(X_1))$ and $\mu_2(= E(X_2))$.

Step 1. Establishing hypothesis:

$$H_0 : \mu_1 - \mu_2 \geq 0; \qquad H_1 : \mu_1 - \mu_2 < 0. \tag{20}$$

Step 2. Constructing statistics:

$$U = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}}. \tag{21}$$

Step 3. Giving rejection region:

$$U = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}} \leq -Z_\alpha, \tag{22}$$

where $\alpha = 0.01$, $n_1 = n_2 = n_3 = 30$.

Step 4. Calculating the value of statistics.

The values of statistics U of different programs are listed in Table 6; $Z_\alpha = 2.325$.

Step 5. Drawing conclusions

From Table 6 we conclude that the values of U are all less than $-Z_\alpha = -2.325$. Then we reject null hypothesis H_0 for all object programs, which means that the time consumption using individual sharing is significantly less than that not using it.

6.3.2. Experimental Results for Testing the Proposed Method. The experimental results of comparing different methods are listed in Table 7. The meanings of all symbols are the same with Table 5. We also use hypothesis testing to give a scientific analysis for the above experimental results. The value of U_1 shows the hypothesis testing results by comparing our method and Gong's method, that of U_2 shows the hypothesis testing results by comparing our method and Ahmed's method, and that of U_3 shows the hypothesis testing results by comparing our method and the random method.

It can be seen from Table 7 that, (1) for all subject programs, the average time consumption using our method is all less than that of Gong's, Ahmed's, and the random methods. The least time consumption of our method is 5.85 seconds (Bubble Sort), while that of Gong's, Ahmed's, and the random methods for the same program is 8.79, 9.85, and 12.32 seconds, respectively. The largest time consumption of our method is 192.82 seconds (Barcode), while that of Gong's, Ahmed's, and the random methods is 224.87, 289.67, and 316.42 seconds, respectively. (2) Gong's and Ahmed's methods have better results than the random method but are all poorer than ours. (3) The values of U_2 and U_3 are all less than $-Z_\alpha = -2.325$. The values of U_1 are all less than $-Z_\alpha = -2.325$ except three programs, that is, Comn, Splinge, and Printtok. For these three programs, the time consumption of our method is still all less than that of Gong's method. Then we conclude that the time consumption using our method is significantly less than that using Gong's, Ahmed's, and random methods.

7. Threats to Validity

The present study focuses on generating test data for multiple paths coverage. One possible threat to the validity of the proposed method may be related to parameter settings. The settings of parameters in GAs have an influence on the performance of generating test data. Appropriate choices of

these values can improve the performance of an algorithm and therefore enhance its efficiency in generating test data. However, how to set proper parameters is not the emphasis of this study; thus we just give the values of the parameters based on our experience. The second threat to the validity may have relation with the use of software systems. Thus, possible bugs or errors, different program conversions, and test objectives may also have influence on the obtained results. Additionally, the selection of target paths may have also influenced the obtained results.

8. Conclusion

We establish a mathematical model which is a rational reflection of the problem of generating test data for multiple paths coverage. On this basis, a multipopulation GA is presented to solve the problem in the model. The main idea of this algorithm, very different from traditional multipopulation GAs, is to improve the search efficiency by means of individual sharing among different subpopulations. In addition, we not only prove the efficiency of our method theoretically, but also apply it in various programs under test. The experimental results show that our method has more significant advantages than Ahmed's multiobjective method and random method. The proposed algorithm in this study enriches the theory and technique of GA-based test data generation and provides a new way to improve the efficiency of software testing.

Possible future researches are presented as follows: one is the method to generate test data when the number of target paths is very large; the other one is the establishment of test platform based on our method.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported by Natural Science Foundation of China (nos. 61203304, 61375067), Natural Science Foundation of Jiangsu Province (no. BK2012566), and the Fundamental Research Funds for the Central Universities (no. 2012QNA41).

References

[1] B. Beizer, *Software Testing Techniques*, International Thomson Computer Press, 1990.

[2] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, 1976.

[3] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[4] R. Malhotra and M. Khari, "Heuristic search-based approach for automated test data generation: a survey," *International Journal of Bio-Inspired Computation*, vol. 5, no. 1, pp. 1–18, 2013.

[5] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*, pp. 970–978, Wiley, 1994.

[6] P. M. S. Bueno, M. Jino, and W. E. Wong, "Diversity oriented test data generation using metaheuristic search techniques," *Information Sciences*, vol. 259, pp. 490–509, 2014.

[7] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[8] K. Ma, K. Y. Phang, J. S. Foster et al., *Static Analysis*, Springer, Berlin, Germany, 2011.

[9] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Software, Practice and Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[10] D. Zhang, C. Nie, and B. Xu, "Optimal allocation of test case considering testing-resource in partition testing," *Journal of Nanjing University (Natural Sciences)*, vol. 41, no. 5, pp. 553–561, 2005.

[11] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[12] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: overview and conceptual c omparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, 2003.

[13] P. McMinn, "Search-based software test data generation: a survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[14] D. W. Gong and X. J. Yao, "Testability transformation based on equivalence of target statements," *Neural Computing and Applications*, vol. 21, no. 8, pp. 1871–1882, 2012.

[15] X. J. Yao, D. W. Gong, Y. J. Luo, and M. Li, "Test data reduction based on dominance relations of ta rget statements," in *Proceedings of IEEE World Congress on Computational Intelligence*, pp. 2191–2198, Springer, 2012.

[16] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology*, vol. 48, no. 7, pp. 586–605, 2006.

[17] A. Baars, M. Harman, Y. Hassoun et al., "Symbolic search-based testing," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pp. 53–62, IEEE, Lawrence, Kan, USA, November 2011.

[18] M. Alshraideh, B. A. Mahafzah, and S. Al-Sharaeh, "A multiple-population genetic algorithm for branch coverage test data generation," *Software Quality Journal*, vol. 19, no. 3, pp. 489–513, 2011.

[19] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, 2001.

[20] O. Bühler and J. Wegener, "Evolutionary functional testing," *Computers and Operations Research*, vol. 35, no. 10, pp. 3144–3160, 2008.

[21] A. Watkins, E. M. Hufnagel, D. Berndt, and L. Johnson, "Using genetic algorithms and decision tree induction to classify software failures," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 2, pp. 269–291, 2006.

[22] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software—Practice and Experience*, vol. 42, no. 11, pp. 1331–1362, 2012.

[23] I. Hermadi, C. Lokan, and R. Sarker, "Genetic algorithm based path testing: challenges and key parameters," in *Proceedings of*

the 2nd WRI World Congress on Software Engineering (WCSE '10), pp. 241–244, Wuhan, China, December 2010.

[24] P. M. S. Bueno and M. Jino, "Automatic test data generation for program paths using genetic algorithms," *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, no. 6, pp. 691–709, 2002.

[25] A. Watkins and E. M. Hufnagel, "Evolutionary test data generation: a comparison of fitness functions," *Software Practice and Experience*, vol. 36, no. 1, pp. 95–116, 2006.

[26] J. Mei and S. Y. Wang, "An improved genetic algorithm for test cases generation oriented paths," *Chinese Journal of Electronics*, vol. 23, no. 3, pp. 494–498, 2014.

[27] I. Hermadi and M. A. Ahmed, "Genetic algorithm based test data generator," in *Proceedings of the Congress on Evolutionary Computation (CEC '03)*, pp. 85–91, Canberra, Australia, December 2003.

[28] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[29] M. A. Ahmed and I. Hermadi, "GA-based multiple paths test data generator," *Computers and Operations Research*, vol. 35, no. 10, pp. 3107–3124, 2008.

[30] D. W. Gong and Y. Zhang, "Novel evolutionary generation approach to test data for multiple paths coverage," *Acta Electronica Sinica*, vol. 38, no. 6, pp. 1299–1304, 2010.

[31] I. Hermadi, C. Lokan, and R. Sarker, "Dynamic stopping criteria for search-based test data generation for path testing," *Information and Software Technology*, vol. 56, no. 4, pp. 395–407, 2014.

[32] P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitorin g the search for test data," in *Proceedings of IEEE International Conference on Automated Software Engineering (ASE '00)*, pp. 209–218, Grenoble, France, 2000.

[33] R. L. Becerra, R. Sagarna, and X. Yao, "An evaluation of differential evolution in software test data generation," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '09)*, pp. 2850–2857, Trondheim, Norway, May 2009.

[34] A. Pachauri and G. Srivastava, "Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1191–1208, 2013.