Analyzing the Performance of the Multiple-Searching Genetic Algorithm to Generate Test Cases

Wanida Khamprapai 1,2, Cheng-Fa Tsai 2,3* and Paohsi Wang 3

1 Department of Tropical Agriculture and International Cooperation, National Pingtung University of Science and Technology, Pingtung 91201, Taiwan; p10522012@mail.npust.edu.tw
2 Department of Management Information Systems, National Pingtung University of Science and Technology, Pingtung 91201, Taiwan
3 Department of Food and Beverage Management, Cheng Shiu University, Kaohsiung 83347, Taiwan; k0627@gcloud.csu.edu.tw
* Correspondence: cftsai@mail.npust.edu.tw; Tel.: +886-08-770-3201 (ext. 7906)

Abstract: Software testing using traditional genetic algorithms (GAs) minimizes the required number of test cases and reduces the execution time. Currently, GAs are adapted to enhance performance when finding optimal solutions. The multiple-searching genetic algorithm (MSGA) has improved upon current GAs and is used to find the optimal multicast routing in network systems. This paper presents an analysis of the optimization of test case generations using the MSGA by defining suitable values of MSGA parameters, including population size, crossover operator, and mutation operator. Moreover, in this study, we compare the performance of the MSGA with a traditional GA and hybrid GA (HGA). The experimental results demonstrate that MSGA reaches the maximum executed branch statements in the lowest execution time and the smallest number of test cases compared to the GA and HGA.

Keywords: software testing; branch coverage; genetic algorithm; multiple-search genetic algorithm; network systems

1. Introduction

The software contains a set of statements that provide instructions for a computer to perform a task. These statements are divided into sequence or control statements that are executed in order. Control statements must consider conditions under which the next statements will be executed. The next statements then perform different statements depending on conditions. Control statements act to implement different sets of statements depending on if the initial condition is true or false. Therefore, control statements affect software outputs and can be the source of error. Software testing processes help verify mistakes that may happen in control statements by determining a set of inputs for a test, called test case, which can then reveal the error or execute as many statements as possible. There are many software testing techniques, including genetic algorithms (GAs), particle swarm optimization, and ant colony optimization, all of which are applied to determine a set of inputs or generate test cases.

GAs are utilized to solve optimization problems in many fields, such as network systems [1], image processing [2], or software testing [3–5]. GAs can search for an optimal solution to a complex system and locate the near global optimum [6]. However, traditional GAs may not be appropriate for solving some complex systems. Therefore, researchers have examined ways to improve GAs for applications to complex problems and have examined ways to enhance solution efficiencies. These efforts to improve GAs have relied in part on accurate configuration parameters, including the population size, crossover probability, and mutation probability.
The multiple-searching genetic algorithm (MSGA) [1] has been proposed as an improved GA for routing between source and destination in network systems by configuring parameters to receive an optimal solution to the given problem. An MSGA can find optimal solutions more quickly than traditional GAs and can be applied in different fields. In this study, MSGA was utilized in software testing, with suitable parameter values defined to enable the generation of optimal test cases. Following the definition of the parameter values, MSGA was compared to a traditional GA and hybrid GA (HGA) in terms of the execution time, number of test cases, and percentage of coverage [7] using the software program Siemens Suite.

2. Motivation and Related Work

GAs are used in numerous and diverse fields when optimal solutions are needed in reasonable amounts of time. In the case of software testing, GAs are used to search for appropriate test cases. Software testing evaluates the quality of software to satisfy user requirements or to prove the correctness of software. Testers wish to generate the smallest number of test cases that cover as many statements as possible and adequately test a given test criterion. For example, applying a GA for test case generation corresponds with du-path coverage to execute all paths where variables are defined and used [8]. GAs have been applied to evaluate test cases to satisfy mutation testing [9] and used to generate test cases in accordance with branch coverage [10–13].

In recent years, researchers have adjusted GAs to increase efficiency for application to new complex systems. For instance, researchers have focused on improving the selection operator to find optimal solutions to the multicast routing problem [1], performing a local search with the best offspring in a generation to quickly converge on the global optimum [7], improving the population initialization and crossover operator to segment magnetic resonance images [14], applying the Gaussian function with crossover and mutation to reduce computation time [15], and solving the traveling salesman problem by combining a GA and two local optimization strategies [16].

MSGAs improve upon traditional GAs by employing a selection operator with two types of chromosomes having different mutation probabilities. MSGAs successfully find optimal solutions and obtain global-minimum solutions. Although MSGAs have been previously applied in network systems, this is the first time that an MSGA is being used in software testing. In this study, we investigated the performance of an MSGA in software testing by considering the configuration of the important parameters of population size, crossover operator, and mutation operator. Subsequently, the MSGA performance in terms of test case generation and time was compared to that obtained using a traditional GA and an HGA. HGA [7] is a GA modified by adding a local search procedure after the processes of GA. The local search is only executed on the best offspring in current population. The HGA improves accuracy and efficiency to find a solution. Specifically, performance comparison investigates: Does the improvement of the selection operator in the MSGA affect in generating test cases when algorithms were assigned the same parameters?

3. Test Case Generation Using GAs

GAs have been used by researchers to generate test cases during software development [12,17–19]. GAs utilize chromosomes that represent test cases or a set of test data. The length of the chromosome corresponds to the number of input parameters in the software under test (SUT). The basic operators of GAs produce new test cases in the next generation, while a fitness function considers selecting suitable test cases.

3.1. Problem Definition

Test case generation is related to the SUT. Since a tester cannot test the SUT with all the possible values of input parameters, a test criterion is used to consider which inputs to apply. The tester defines the test criterion and methodology for generating test cases. Generating an optimal test case using a GA can be formally outlined as $ST = \{SUT, C, P_{GA}, G, Sc\}$, where $SUT$ denotes the software under
test, \( C \) denotes the test criterion, \( P_{GA} \) is the adopted control parameters in the GA, \( G \) indicates the number of generations, and \( Sc \) is the stopping criterion. The fitness function is associated with the test criterion and is defined as \( P_{GA} = \{ Ps, F, Sel, Cro, Mut \} \), where \( Ps \) is the population size, \( F \) indicates the fitness function, \( Sel \) denotes the selection operator, \( Cro \) denotes the crossover operator, and \( Mut \) is the mutation operator.

The test criterion \( C \) affects the test requirement \( TR \). This research uses a branch coverage criterion. Test case generation for the branch coverage criterion can be from the source code, a requirement specification, or a development model. When using the branch coverage criterion, each decision statement is executed to obtain all possible outcomes at least once. The decision statements of source code are obtained from if, switch, or loop statements. Let \( D \) represent a set of decision statements in \( SUT \) and \( P \) denote a set of parameters in \( D \). Each decision statement \( d \in D \), and the possible outcomes of \( d \) are true or false. Therefore, generating test cases according to the branch coverage criterion can be defined as \( TR = \{ d = \text{true}, d = \text{false} \} \).

The value of parameter \( p \in P \) in the decision statement determines the value of \( d \). The challenge lies in providing the value of the parameters to reveal the software’s behavior. A suitable value of a parameter will execute as many decision statements as possible. The percentage of executed decision statements can be computed as \( Coverage = \left( \frac{D_{exec}}{D_{SUT}} \right) \times 100 \), where \( D_{exec} \) represents the number of executed decision statements in \( SUT \), and \( D_{SUT} \) is the total number of decision statements in \( SUT \).

### 3.2. Representation of Chromosomes

Chromosomes are a set of possible solutions. A good chromosome design facilitates an efficient solution for a GA. The representation of a chromosome corresponds to a given problem. In software testing, chromosomes are represented as input parameters with length \( n \) [20–22]. Each gene is denoted as \( p_i \), where \( 1 \leq i \leq n \), relates to the data type and condition of the parameter. If the data type of the \( i \)th gene \( p_i \) is double, then \( p_i \) is assigned the value as double. Figure 1 shows the representation of chromosomes with examples of input parameters. The SUT has five input parameters, the chromosome contains five genes, and each gene relates to one input parameter.

![Chromosome Representation](image)

**Figure 1.** Chromosome representation to generate test cases using a genetic algorithm (GA). Each chromosome has five genes \((p)\) that each correspond to one input parameter of a specific data type.
Each gene in a chromosome consists of data to be tested. The configuration for each gene will differ depending on the tested software. For example, the data type of variable $a$ in Figure 1 is integer, the possible value of variable $a$ is based on the range of integer data type in the C programming. Variable $a$ affects the arithmetic expression ($x < a$) of the decision statement, causing variable $sc$ to have a different number of inputs depending on variable $a$. The length of each chromosome will be different. If the value of variable $a$ is 30, variable $sc$ is taken as 30 inputs. Therefore, the length of chromosome is 31, it is obtained from variable $a$, and the number of variable $sc$.

3.3. Fitness Function

This study focuses on branch coverage as the test criterion. This experiment designed a fitness function to measure that the test case generated by the MSGA covers the target decision statements. A fitness function is a summation of a branch coverage level and a branch distance. The branch coverage level counts the number of decision statements covered by each test case. The branch distance calculates how close the test case was to stay on a path leading to the target. The branch distance is computed according to Equation (1) [23].

$$\text{Branch distance} = \frac{1}{1.001^{\text{distance}}}$$  

where 1.001 is a constant for the normalized branch distance. A typical value is defined as 1.001 [24]. The obtained distance [25,26] is shown in Table 1.

| Decision Statement | Distance         |
|-------------------|-----------------|
| $a < b$           | $a - b$         |
| $a <= b$          | $a - b$         |
| $a > b$           | $b - a$         |
| $a >= b$          | $b - a$         |
| $a == b$          | $abs(a - b)$    |
| $a! == b$         | $abs(a - b)$    |
| $a && b$          | $a + b$         |
| $a || b$          | $\text{min}(a, b)$ |

The obtained distance is derived from the arithmetic expression of the decision statement being tested. Table 1 displays the calculation of the distance, where $a$ and $b$ are the values of the parameters in the decision statement. For example, the decision statement $a <= b$ must be measured as true. Then, the distance may be assigned as $a - b$. If the obtained distance is zero, parameters $a$ and $b$ are indicated as having the same values.

3.4. Genetic Operators

Problems are solved using GAs through three basic operators: Selection, crossover, and mutation (Figure 2). The selection operator chooses good chromosomes from the evaluation of fitness values. These chromosomes are selected as parent chromosomes. Then, the crossover and mutation operators are applied to produce offspring in the next generation. The crossover operator takes two chromosomes and exchanges some genes. The mutation operator changes the value of one or more genes. The performance of the crossover and mutation operator depends on the given functions and probabilities that can differ according to the nature of the problem.
4. Multiple-Searching Genetic Algorithms

MSGAs [1] are GAs that have been improved in 2004 to solve the problem in network systems. An MSGA establishes more chromosomes to avoid local optima traps. Different chromosomes are applied to different search methods to find the global optimum [1]. The MSGA utilizes multiple searches in many different directions to avoid local optima traps. The processes involved in MSGAs are similar to those of GAs but include an expanded selection mechanism with three sub-processes (Figure 3). Algorithm 1 shows the process utilized by an MSGA.

**Algorithm 1.** Pseudocode for MSGA.

1: Create initial chromosomes
2: Evaluate fitness value and order by descending
3: while not terminal condition do
4: Select half of chromosomes with the highest fitness value and discard the retain chromosomes //Conservative chromosomes
5: Create Explorer chromosomes
6: Combine Conservative and Explorer chromosomes Cross
7: Cross chromosomes
8: Mutate Conservative chromosomes with $M_c$
9: Mutate Explorer chromosomes with $M_e$
10: Evaluate fitness value
11: end while
12: return chromosomes

![Figure 3](image-url)  
*Figure 3. Flowcharts of traditional (a) GAs and (b) multiple-searching genetic algorithms (MSGAs).*
Similar to GAs, MSGAs create initial chromosomes and measure the fitness value of chromosomes to preserve half of the chromosomes with the highest fitness values (conservative chromosomes). The remaining half of chromosomes are discarded. After the selection of these chromosomes, a candidate tree is applied to create explorer chromosomes. Then, conservative and explorer chromosomes are combined (Figure 4). The combined population is performed crossover and mutation.

![Figure 4. Selection operators for MSGAs.](image)

The candidate tree used to generate explorer chromosomes involves each set of candidate genes collecting genes in the same position on the whole conservative chromosomes. Next, the gene set chooses one gene for each position on the conservative chromosome to create an explorer’s gene and then combines as the explorer’s chromosomes. This mechanism iterates until all genes are executed. The explorer’s gene can be selected by a roulette wheel or any similar selection method (Figure 5).

![Figure 5. Mechanism for generating explorer chromosomes. Only one example of a randomly generated explorer chromosome is shown in the middle panel. Red lines indicate which gene from each position was chosen.](image)

The crossover operation included in MSGAs is the same as in traditional GAs. The crossover operation selects two chromosomes from the set of all chromosomes and crosses according to a given crossover method and probability. Crossover methods are used to determine cut points in chromosomes to swap genes and can include one-point, two-point, and uniform methods. The crossover probability is the chance that two chromosomes will exchange genes. Algorithm 2 outlines the crossover operation step in an MSGA.
Two chromosomes, one from each set of conservative and explorer chromosomes, are selected as parent chromosomes for a crossover operation (Figure 6). Each chromosome is split corresponding to the given crossover method and the positions of cut points in the chromosomes are randomly created. Crossover operations are performed with the same crossover method and probability on both conservative and explorer chromosomes. After the crossover, one chromosome is kept to the conservative chromosome and another one is retained to the explorer chromosome.

The mutation operation involves changing some genes in the chromosome according to a mutation probability. The mutation probability determines the number of mutated chromosomes in one generation. The selected chromosomes are changed randomly in one or more genes. If the mutation probability is high, the algorithm becomes a random search. The conservative and explorer chromosomes are assigned with different mutation probabilities (Figure 7). The explorer chromosomes are assigned with higher mutation probability than the conservative chromosomes to prevent premature convergence [1].

5. Experimental Evaluation of the MSGA

This section presents the experimental platform for the ability of the MSGA to generate test cases and the details of Siemens Suite, the tested program.
5.1. Experimental Platform

The experiments were carried out on Windows 10 Enterprise (Seattle, WA, USA) ×64 with an Intel® Core i7 CPU 3.60 GHz and 4 GB of RAM. The testing tool used Gcov plug-in for Eclipse IDE. Gcov was used to collect the coverage information of the SUT until the end of the processes. Every new test case (chromosome) of each generation called Gcov to count the number of times each decision statement in the SUT was executed.

5.2. Software Under Test (SUT)

The performance of the MSGA was evaluated on Siemens Suite (Table 2). A test case was brought as input parameters for Siemens Suite and then the number of decision statements that were executed was observed. Siemens Suite is widely utilized as the benchmark test software. Siemens Suite has been previously used to evaluate the performance of various techniques [27–29] and contains seven programs that were written with the C language (Table 2). The programs contained in Siemens Suite were developed to study the fault detection of a given test criterion. Each program is related to the seeded faults that are inserting errors into the source code.

| Program Name   | No. of Lines | No. of Processes | No. of Decisions | Description                     |
|----------------|--------------|------------------|------------------|---------------------------------|
| print_tokens.c | 726          | 18               | 108              | Analyzer lexical                |
| print_tokens2.c| 570          | 19               | 161              | Analyzer lexical                |
| replace.c      | 564          | 21               | 163              | Pattern matching and replacing  |
| schedule.c     | 412          | 18               | 55               | Priority task scheduler         |
| schedule2.c    | 374          | 16               | 88               | Priority task scheduler         |
| tcas.c         | 173          | 9                | 50               | Aircraft collision avoidance    |
| tot_info.c     | 565          | 7                | 86               | Computes statistics             |

6. Parametric Analysis

The performance of a GA depends upon the value of the parameters. Different problems call for different optimal values of these parameters [30]. Previous research has determined the optimal values of the parameters for multicast routing in large networks. An MSGA has never been applied in software testing. In this study, parametric analysis of MSGA considered the parameters population size, crossover operation, and mutation operation to obtain the optimal values for generating test cases. The experiment in this part is conducted on one program, print_tokens.c, due to this, it is a median-scale program selected from seven programs. To analyze the stability of these parameters, we used non-parametric Mann–Whitney U tests (p-value) with a significance level of 0.05 to investigate the result of each given parameter. Each result was calculated to lie between parameters with the best average value and the other values. The empirical significance of the difference between the results of each parameter was evaluated using an independent parametric t-test.

6.1. Population Size

Researchers have struggled to find an accurate definition of optimal population size for GAs. A small population size could result in poor search solutions, while a large population size could result in optimal solutions only after a very time-consuming search [31,32]. In this study, the MSGA was executed 100 times in each population size. Small population sizes achieved higher average numbers of generations compared to large population sizes (Table 3). The small population sizes could quickly arrive at the maximum number of generations, whereas the large population sizes executed more slowly. The optimal population size of MSGA for generating test cases is recommended to be between 50 and 70.
Table 3. Effect of population size for test case generation using the MSGA.

| Population Size | Best Value | Average Value | Worst Value |
|-----------------|------------|---------------|-------------|
|                 | Time (s)   | No. of Generations | Time (s)   | No. of Generations | Time (s) | No. of Generations |
| No. of parameters |            |                |            |                       |          |                     |
| 30              | 30.80      | 3              | 119.59     | 54.96                  | 345.20   | 100                  |
| 50              | 35.40      | 3              | 88.73      | 18.29                  | 431.30   | 100                  |
| 70              | 49.10      | 3              | 108.09     | 10.56                  | 717.00   | 100                  |
| 100             | 72.80      | 2              | 145.05     | 7.71                   | 391.50   | 13                   |
| 150             | 106.60     | 3              | 327.04     | 5.92                   | 507.20   | 10                   |

We analyzed the stability of the population size of 50 parameters compared with those of the other sizes. The results in Table 4 are based on the t-test. It is observed that the population sizes of 50 and 70 produce no differences in their means. For the Mann–Whitney test ($p$-value), which was used to compare the performances for the population size of 50 and the other sizes, the results are all less than 0.05 and are significantly different. Both population sizes of 50 and 70 show the same satisfactory performance in generating test cases. When considering the best and the worst values, this study used a population size of 50 (Table 3).

Table 4. Statistical comparison between population size 50 and the others.

| Pair of the Values | t-Test       | Mann-Whitney ($p$-Value) |
|-------------------|--------------|--------------------------|
| 50 vs. no. of parameters | <0.00001     | <0.00001                 |
| 50 vs. 30         | 0.021425     | <0.00001                 |
| 50 vs. 70         | 0.124037     | <0.00001                 |
| 50 vs. 100        | 0.012744     | <0.00001                 |
| 50 vs. 150        | 0.013221     | <0.00001                 |

6.2. Selection Operation

The MSGA used in this study includes a selection operator that is expanded from a traditional GA to include the generation of conservative and explorer chromosomes. This experiment applied a selection operator where half of the population of chromosomes with the highest fitness were selected to be conservative chromosomes. Then, explorer chromosomes were generated using the roulette wheel selection for each gene (see Figure 5). Each gene is assigned a selection proportion according to the value of the fitness of each chromosome. Genes of chromosomes with higher fitness obtained the larger segment in a roulette wheel.

6.3. Crossover Operation

Crossover operations within MSGAs are used to solve for the optimal solution. The selection of a crossover operator depends on a given problem. Furthermore, the performance of a crossover operator depends on probability. Generally, crossover probability is configured at a high rate [33,34]. A low crossover probability results in slow convergence to the optimal solution [35]. In this experiment, the crossover operators tested three basic functions (one-point, two-point, and uniform). The crossover probability was specified to be between 0.6 and 1, with increments of 0.1. Each crossover operation was independently executed 100 times while using a mutation probability of zero. When different crossover operators were used to solve a given problem with different optimal crossover probabilities, it was found that an overall uniform crossover outperformed both one-point and two-point crossovers (Table 5). Therefore, the best crossover parameter for this MSGA was a uniform crossover operator with a 0.8 crossover probability.
Table 5. Effect of crossover operator for test case generation using the MSGA.

| Crossover Operation | Crossover Probability | Best Value | Average Value | Worst Value |
|---------------------|-----------------------|------------|---------------|-------------|
|                     |                       | Time (s)   | No. of Generations | Time (s)   | No. of Generations | Time (s)   | No. of Generations |
| One-point           | 0.6                   | 49.1 3     | 108.09 10.56      | 717.00 100 |
|                     | 0.7                   | 42.7 2     | 107.96 9.83       | 703.20 100 |
|                     | 0.8                   | 48.3 3     | 116.47 11.42      | 709.10 100 |
|                     | 0.9                   | 50.1 3     | 123.00 12.52      | 718.60 100 |
|                     | 1.0                   | 42.7 2     | 134.84 14.36      | 803.30 100 |
| Two-point           | 0.6                   | 47.7 3     | 89.08 10.15       | 555.20 100 |
|                     | 0.7                   | 44.5 3     | 81.18 8.06        | 633.20 100 |
|                     | 0.8                   | 44.0 2     | 90.36 10.03       | 609.50 100 |
|                     | 0.9                   | 43.3 2     | 94.20 10.89       | 571.40 100 |
|                     | 1.0                   | 43.4 2     | 88.70 9.85        | 597.90 100 |
| Uniform             | 0.6                   | 49.6 3     | 82.65 6.96        | 542.30 100 |
|                     | 0.7                   | 49.5 3     | 84.14 7.71        | 617.20 100 |
|                     | 0.8                   | 49.3 3     | 76.56 5.81        | 115.30 9 |
|                     | 0.9                   | 55.3 4     | 83.25 6.65        | 667.60 100 |
|                     | 1.0                   | 42.2 6     | 81.68 6.76        | 531.40 100 |

We compared the uniform crossover operator with 0.8 with the others (Table 6). As shown in Table 6, the \(t\)-test indicates that there are five crossover operator values that are greater than 0.05; however, the Mann–Whitney (\(p\)-value) results are all less than 0.05. In considering the worst value, we can still conclude that the uniform crossover operator with 0.8 is better than the others.

Table 6. Statistical comparison between the uniform with a 0.8 and the others.

| Pair of Values          | \(t\)-Test | Mann-Whitney (\(p\)-value) |
|------------------------|------------|----------------------------|
| uniform (0.8) vs. one-point (0.6) | 0.011437   | <0.00001                   |
| uniform (0.8) vs. one-point (0.7) | 0.016065   | <0.00001                   |
| uniform (0.8) vs. one-point (0.8) | 0.006913   | <0.00001                   |
| uniform (0.8) vs. one-point (0.9) | 0.003068   | <0.00001                   |
| uniform (0.8) vs. one-point (1.0) | 0.000960   | <0.00001                   |
| uniform (0.8) vs. two-point (0.6) | 0.029574   | <0.00001                   |
| uniform (0.8) vs. two-point (0.7) | 0.117917*  | <0.00001                   |
| uniform (0.8) vs. two-point (0.8) | 0.033398   | <0.00001                   |
| uniform (0.8) vs. two-point (0.9) | 0.020169   | <0.00001                   |
| uniform (0.8) vs. two-point (1.0) | 0.039893   | <0.00001                   |
| uniform (0.8) vs. uniform (0.6) | 0.115698*  | <0.00001                   |
| uniform (0.8) vs. uniform (0.7) | 0.078426*  | <0.00001                   |
| uniform (0.8) vs. uniform (0.9) | 0.191008*  | <0.00001                   |
| uniform (0.8) vs. uniform (1.0) | 0.161153*  | <0.00001                   |

\* is value greater than 0.05.

6.4. Mutation Operation

The mutation probabilities used in GAs are usually low [36,37]. Many researchers define the mutation probability as \(M_p = 1/n\), where \(n\) is the length of the chromosome [38–40]. However, high mutation probabilities increase the searchability in global optima. If the mutation probability is too high, the resulting chromosomes could be similar to those generated by a random search [41]. The mutation operation mechanism of the MSGA specifies that the conservative and explorer chromosomes have different mutation probabilities. Tsai et al. (2004) [1] assigned a higher mutation probability for explorer chromosomes to get the optimal solution. In this study, conservative chromosomes were given a mutation probability of \(1/n\) while explorer chromosomes had been tested a mutation probability of \(1/n\) and a high mutation probability, ranging from 0.1 to 0.5 with increments of 0.1 (Table 7). The conservative and explorer chromosomes were applied the mutation operator with one-point. The MSGA was executed 100 times per mutation probability. The optimal mutation probability for MSGA to generate test cases was found to be \(1/n\) and 0.1 (Table 7). The higher mutation probability provides a better global optimum. The different mutation probabilities of conservative and explorer chromosomes can avoid local optima.
Table 7. Effect of the mutation operator for test case generation using the MSGA.

| Mutation Probability | Best Value | Average Value | Worst Value |
|-----------------------|------------|---------------|-------------|
|                       | Time (s)   | No. of Generations | Time (s)   | No. of Generations | Time (s)   | No. of Generations |
| $1/n$                 | 50.5       | 2              | 97.25      | 5.71             | 136.40     | 9              |
| 0.1                   | 50.8       | 2              | 94.99      | 5.60             | 157.40     | 11             |
| 0.2                   | 64.9       | 3              | 100.36     | 6.43             | 141.50     | 11             |
| 0.3                   | 62.0       | 3              | 107.96     | 6.91             | 152.60     | 11             |
| 0.4                   | 62.9       | 6              | 111.17     | 7.15             | 173.50     | 15             |
| 0.5                   | 50.3       | 2              | 110.94     | 7.17             | 157.90     | 13             |

In this study, we more heavily focused on comparing the 0.1 mutation probability with the others (Table 8). In terms of both the $t$-test and the Mann–Whitney test ($p$-value), we can observe that there is only one value of the mutation probability ($1/n$) that is greater than 0.05. The explorer chromosomes were assigned with a higher mutation probability than the conservative chromosomes. Therefore, the optimal mutation probability for the explorer chromosome is defined as 0.1.

Table 8. Statistical comparison between the 0.1 mutation probability and the others.

| Pair of Values | $t$-Test | Mann-Whitney ($p$-Value) |
|---------------|---------|------------------------|
| 0.1 vs. $1/n$ | 0.316314 | 0.2776                 |
| 0.1 vs. 0.2   | $<0.00001$ | $<0.00001$            |
| 0.1 vs. 0.3   | $<0.00001$ | $<0.00001$            |
| 0.1 vs. 0.4   | $<0.00001$ | $<0.00001$            |
| 0.1 vs. 0.5   | $<0.00001$ | $<0.00001$            |

The parametric analysis indicates that an optimal solution for test case generation using MSGA should use a population size of 50, a uniform crossover operator, a crossover probability of 0.8, and mutation probabilities of $1/n$ and 0.1 for conservative and explorer chromosomes, respectively.

7. Performance Comparison

To evaluate the performance and robustness of the MSGA, the solution of the MSGA was compared with those of a traditional GA and an HGA in terms of execution time, number of test cases, and percentage coverage. To the stability of the MSGA compared to that of the GA and HGA, the Vargha-Delaney $\hat{A}_{12}$ effect size is calculated. Furthermore, each algorithm was reported with a 95% confidence interval of number of test cases achieved. The execution time includes the whole process for generating test cases, running test cases, and coverage acquisition. The percentage coverage was obtained from the number of executed branch statements. The parameters were defined according to the findings of the parametric analysis (Tables 3–8). All three algorithms were assigned the same parameters (Table 9).

Table 9. Parameter settings for performance comparison. Note: $M_p$ and $M_e$ indicate mutation probability of conservative and explorer chromosomes, respectively.

|                      | MSGA     | GA       | HGA      |
|----------------------|----------|----------|----------|
| Crossover operator   | Uniform  | Uniform  | Uniform  |
| Crossover probability| 0.8      | 0.8      | 0.8      |
| Mutation operator    | One-point| One-point| One-point|
| Mutation probability | $M_p = 1/n$, $M_e = 0.1$ | $1/n$       | $1/n$    |
| Population size      | 50       | 50       | 50       |
| Generations          | 100      | 100      | 100      |
| Stopping criteria    | After 100 generations or coverage > 85% | After 100 generations or coverage > 85% | After 100 generations or coverage > 85% |
The mutation operation of the GA and the HGA were the same as the mutation of conservative chromosomes. The processes of all three algorithms stopped after the maximum number of generations (100) was reached or the percentage coverage of the branch statements exceeded 85%. It is impossible to achieve 100% coverage. The maximum coverage level possible is 5/6 or 83% [42]. Thus, 85% coverage is enough to the stopping criterion. All seven programs in the SUT were independently executed 100 times for each algorithm.

The MSGA outperformed the GA and the HGA in the case of test case generation, except for schedule.c and schedule2.c (Table 10). The percentage of coverage is the most important performance indicator because it indicates that the test cases generated with each algorithm can execute an equal number of branch statements. The test cases of the MSGA can execute more branch statements than all seven programs with a coverage of over 90%.

Table 10. Results for generating test cases using MSGA, GA, and hybrid GA (HGA).

| Program Name          | Algorithm | No. of Test Cases | $\hat{A}_{12}$ (MSGA:Other) | Confidence Interval | Coverage (%) |
|-----------------------|-----------|-------------------|-----------------------------|---------------------|--------------|
| print_tokens.c        | MSGA      | 292.50            | -                           | (77.33, 313.67)     | 93.44        |
|                       | GA        | 405               | 0.72                        | (373.17, 436.83)    | 91.93        |
|                       | HGA       | 345.50            | 0.61                        | (320.71, 370.29)    | 91.93        |
| print_tokens2.c       | MSGA      | 373.50            | -                           | (357.39, 389.61)    | 98.00        |
|                       | GA        | 445               | 0.67                        | (417.84, 472.16)    | 96.00        |
|                       | HGA       | 425.50            | 0.63                        | (401.72, 449.26)    | 97.50        |
| replace.c             | MSGA      | 1710.50           | -                           | (1351.46, 2063.54)  | 92.21        |
|                       | GA        | 1899.50           | 0.52                        | (1505.39, 2293.61)  | 91.39        |
|                       | HGA       | 1894              | 0.52                        | (1492.33, 2295.67)  | 91.39        |
| schedule.c            | MSGA      | 50                | -                           | (50, 50)            | 96.86        |
|                       | GA        | 50                | 0.5                         | (50, 50)            | 96.86        |
|                       | HGA       | 50                | 0.5                         | (50, 50)            | 96.86        |
| Schedule2.c           | MSGA      | 50                | -                           | (50, 50)            | 97.12        |
|                       | GA        | 50                | 0.5                         | (50, 50)            | 97.12        |
|                       | HGA       | 50                | 0.5                         | (50, 50)            | 97.12        |
| tcas.c                | MSGA      | 174               | -                           | (164.39, 183.61)    | 96.92        |
|                       | GA        | 335               | 0.90                        | (307.13, 362.87)    | 90.77        |
|                       | HGA       | 218.50            | 0.73                        | (206.15, 230.85)    | 96.92        |
| tot_info.c            | MSGA      | 327               | -                           | (297.92, 356.08)    | 92.25        |
|                       | GA        | 1220              | 0.87                        | (1013.18, 1426.82)  | 89.15        |
|                       | HGA       | 327.50            | 0.52                        | (303.22, 351.78)    | 89.15        |

For four out of seven programs, the MSGA yielded fewer generations, fewer test cases, and a higher percentage of coverage than both the GA and the HGA (Table 10). For the program tcas.c, the MSGA and the HGA had an equal percentage coverage, but the MSGA had a lower number of generations and number of test cases. The programs schedule.c and schedule2.c demonstrated similar performance among the three algorithm types, generating the same number of generations and test cases and very similar percentages of coverage (Table 10). The MSGA took less time to achieve coverage than the GA and the HGA for all seven programs (Table 11). The $\hat{A}_{12}$ measure indicated that the MSGA performed better than the GA and the HGA because all values are more than 0.5. For example, 80% of the time, the programs schedule.c and schedule2.c took less time than the GA.

Table 11. Time performance of the MSGA, GA, and HGA for the seven programs in the software under test (SUT). Note: Bold values are the best values of each program.

| Program Name         | Average (s) | Standard Deviation | $\hat{A}_{12}$ (MSGA: Other) |
|----------------------|-------------|-------------------|-----------------------------|
| print_tokens.c       | 98.21       | 112.23            | 126.95                       |
| print_tokens2.c      | 155.15      | 173.02            | 163.37                       |
| replace.c            | 240.33      | 257.99            | 243.04                       |
| schedule.c           | 25.40       | 25.80             | 25.55                        |
| schedule2.c          | 25.00       | 25.24             | 25.17                        |
| tcas.c               | 30.95       | 35.50             | 31.42                        |
| tot_info.c           | 151.51      | 404.64            | 165.62                       |

For four out of seven programs, the MSGA yielded fewer generations, fewer test cases, and a higher percentage of coverage than both the GA and the HGA (Table 10). For the program tcas.c, the MSGA and the HGA had an equal percentage coverage, but the MSGA had a lower number of generations and number of test cases. The programs schedule.c and schedule2.c demonstrated similar performance among the three algorithm types, generating the same number of generations and test cases and very similar percentages of coverage (Table 10). The MSGA took less time to achieve coverage than the GA and the HGA for all seven programs (Table 11). The $\hat{A}_{12}$ measure indicated that the MSGA performed better than the GA and the HGA because all values are more than 0.5. For example, 80% of the time, the programs schedule.c and schedule2.c took less time than the GA.
As shown in Tables 10 and 11, the performance comparison indicates that the MSGA has higher performance than other algorithms when assigning the same parameters. The improvement of the selection operator affects and generating test cases in the aspect of the time performance and coverage. MSGA can produce test case generation quicker and obtain a higher percentage of coverage. Except for the results, schedule.c and schedule2.c are quite similar because both programs are very small-scale programs.

8. Discussion

According to our study, the MSGA in software testing can find two other related works but those MSGAs differ the full names and the improved processes, namely, multi-stage genetic algorithm [43,44]. It has optimized processes for testing the object-oriented programs. It includes two stages. First, finding test cases that satisfy a given test criterion. Second, generating test data. A population in the MSGA contains conservative and explorer chromosomes. Creating more chromosomes allows the MSGA to effectively achieve optimum quickly. Explorer chromosomes are produced from chromosomes with high fitness value (conservative chromosomes). Selecting genes from conservative chromosomes provides the explorer chromosomes a chance of high fitness value. Chromosomes in each generation of the MSGA have a greater chance of getting closer to the optimum solution of the desired problem. From the above performance comparison, the MSGA can achieve effective optimum quickly. Furthermore, the MSGA generated fewer test cases than the others while obtaining a greater percentage of coverage. This finding shows that the MSGA produced a solution that was as close as possible to the problem features. Creating more chromosomes using the mechanism of multiple-searching resulted in a diversified population and produced global-minimum solutions. Although the small-scale programs produced the same results for all algorithms, the larger-scale programs provided different results. Owing to the decreased number of test cases, the time and cost incurred by software testing resulted in a 50% reduction of software development costs [41].

9. Conclusions

In this study, we investigated parameter optimization for test case generation using an MSGA and compared the efficiency of the MSGA with that of a traditional GA and HGA. As the optimal configuration of parameters differs among problems, the MSGA was tested with seven programs in Siemens Suite to obtain the optimal solution of each parameter across different programs. The parametric analysis yielded three main results. First, the MSGA should use a population size of 50. A smaller population size results in an inadequate solution that requires more generations, while a larger population size requires more time to solve the problem. Second, the analysis of the crossover operator demonstrated that the crossover function should be uniform and the crossover probability should be 0.8. Finally, the mutation operator for conservative and explorer chromosomes should be 1/n and 0.1, respectively.

The results of comparison of the MSGA to the GA and HGA in terms of efficiency indicated that the MSGA requires fewer generations and fewer test cases than the traditional GA and HGA. Moreover, the MSGA provides a higher percentage of coverage and requires less time to generate test cases than the traditional GA and HGA. Although Siemens Suite consists of a set of small-scale to medium-scale programs and is written with language C, the concept of MSGA can be applied to generate test cases in complex programs and programs developed with other languages. Moreover, we believe that the MSGA still has many potential possibilities to improve performance through integration with other methods.

Author Contributions: Conceptualization, W.K. and C.-F.T.; methodology, W.K. and C.-F.T.; software, W.K.; validation, P.W.; formal analysis, W.K.; investigation, W.K. and C.-F.T.; resources, W.K.; data curation, P.W.; writing—original draft preparation, W.K.; writing—review and editing, W.K. and C.-F.T.; visualization, W.K.; supervision, C.-F.T.; project administration, C.-F.T.; funding acquisition, C.-F.T. All authors have read and agreed to the published version of the manuscript.
**Funding:** This research was funded by the Ministry of Science and Technology, Republic of China, Taiwan, grant numbers MOST-107-2637-E-020-006, MOST-108-2637-E-020-003, MOST-108-2321-B-020-003, and MOST-107-2321-B-020-005.

**Acknowledgments:** The authors would like to express their sincere gratitude to the anonymous reviewers for their useful comments and suggestions for improving the quality of this paper, and we thank the Department of Tropical Agriculture and International Cooperation, Department of Management Information Systems, National Pingtung University of Science and Technology, Taiwan, and the Ministry of Science and Technology, Republic of China, Taiwan, for supporting this research.

**Conflicts of Interest:** The authors declare no conflict of interest.

**References**

1. Tsai, C.-F.; Tsai, C.-W.; Chen, C.-P. A novel algorithm for multimedia multicast routing in a large scale network. *J. Syst. Softw.* 2004, 72, 431–441. [CrossRef]
2. Cao, X.; Miao, J.; Xiao, Y. Medical Image Segmentation of Improved Genetic Algorithm Research Based on Dictionary Learning. *World J. Eng. Technol.* 2017, 5, 90–96. [CrossRef]
3. Koleejan, C.; Xue, B.; Zhang, M. Code coverage optimisation in genetic algorithms and particle swarm optimisation for automatic software test data generation. In Proceedings of the 2015 IEEE Congress on Evolutionary Computation (CEC), Sendai, Japan, 25–28 May 2015; pp. 1204–1211.
4. Chuaychoo, N.; Kansomkeat, S. Path Coverage Test Case Generation using Genetic Algorithm. *J. Telecommun. Electron. Comput. Eng.* 2017, 9, 115–119.
5. Khan, R.; Amjad, M. Introduction to Data Flow Testing with Genetic Algorithm. *Int. J. Comput. Appl.* 2017, 170, 39–45. [CrossRef]
6. Ayad, A.; Awad, H.; Yassin, A. Parametric analysis for genetic algorithms handling parameters. *Alex. Eng. J.* 2013, 52, 99–111. [CrossRef]
7. Wan, W.; Birch, J.B. An Improved Hybrid Genetic Algorithm with a New Local Search Procedure. *J. Appl. Math.* 2013, 2013, 1–10. [CrossRef]
8. Khan, R.; Amjad, M.; Srivastava, A.K. Optimization of Automatic Generated Test Cases for Path Testing Using Genetic Algorithm. In Proceedings of the 2016 Second International Conference on Computational Intelligence & Communication Technology (CICT), Ghaziabad, India, 12–13 February 2016; pp. 32–36. [CrossRef]
9. Jatana, N.; Suri, B. Particle Swarm and Genetic Algorithm applied to mutation testing for test data generation: A comparative evaluation. *J. King Saud Univ.—Comput. Inf. Sci.* 2020, 32, 514–521. [CrossRef]
10. Pachauri, A.; Srivastava, G. Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism. *J. Syst. Softw.* 2013, 86, 1191–1208. [CrossRef]
11. Aleti, A.; Grunske, L. Test data generation with a Kalman filter-based adaptive genetic algorithm. *J. Syst. Softw.* 2015, 103, 343–352. [CrossRef]
12. Yang, S.; Man, T.; Xu, J.; Zeng, F.; Li, K. RGA: A lightweight and effective regeneration genetic algorithm for coverage-oriented software test data generation. *Inf. Softw. Technol.* 2016, 76, 19–30. [CrossRef]
13. Bahaweres, R.B.; Zawawi, K.; Khairani, D.; Hakiem, N. Analysis of statement branch and loop coverage in software testing with genetic algorithm. In Proceedings of the 2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI), Yogyakarta, Indonesia, 19–21 September 2017; pp. 1–6. [CrossRef]
14. Das, S.; De, S. A Modified Genetic Algorithm Based FCM Clustering Algorithm for Magnetic Resonance Image Segmentation. In Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications (FICTA 2016), Bhubaneswar, India, 16–17 September 2016; pp. 435–443. [CrossRef]
15. Pravesjit, S.; Kantawong, K. An improvement of genetic algorithm for optimization problem. In Proceedings of the 2017 International Conference on Digital Arts, Media and Technology (ICDAMT), Chiang Mai, Thailand, 1–4 March 2017; pp. 226–229. [CrossRef]
16. Borna, K.; Hashemi, V.H. An Improved Genetic Algorithm with a Local Optimization Strategy and an Extra Mutation Level for Solving Traveling Salesman Problem. *Int. J. Comput. Sci. Eng. Inf. Technol.* 2014, 4, 47–53. [CrossRef]
17. Khurana, N.; Chillar, R. Test Case Generation and Optimization using UML Models and Genetic Algorithm. *Procedia Comput. Sci.* 2015, 57, 996–1004. [CrossRef]

18. Ramaiah, A.; Basu, A. Basis Path Based Test Suite Minimization Using Genetic Algorithm. *Int. J. Intell. Syst. Appl.* 2018, 10, 36–49. [CrossRef]

19. Esfandyari, S.; Rafe, V. A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy. *Inf. Softw. Technol.* 2018, 94, 165–185. [CrossRef]

20. Thi, D.N.; Hieu, V.D.; Ha, N.V. A Technique for Generating Test Data Using Genetic Algorithm. In Proceedings of the 2016 International Conference on Advanced Computing and Applications (ACOMP), Can Tho, Vietnam, 23–25 November 2016; pp. 67–73. [CrossRef]

21. Sabharwal, S.; Aggarwal, M. Test Set Generation for Pairwise Testing Using Genetic Algorithms. *J. Inf. Process. Syst.* 2015, 13, 1089–1102. [CrossRef]

22. Akhter, N.; Singh, A.; Singh, G. Automatic Test Case Generation by using Parallel 3 Parent Genetic Algorithm. *Int. J. Res. Appl. Sci. Eng. Technol.* 2018, 6, 114–121. [CrossRef]

23. Pachauri, A.; Gursaran; Mishra, G. A path and branch based approach to fitness computation for program test data generation using genetic algorithm. In Proceedings of the 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), Noida, India, 25–27 February 2015; pp. 49–55. [CrossRef]

24. Wegener, J.; Baresel, A.; Sthamer, H. Evolutionary test environment for automatic structural testing. *Inf. Softw. Technol.* 2001, 43, 841–854. [CrossRef]

25. Garg, D.; Garg, P. Basis Path Testing Using SGA & HGA with ExLB Fitness Function. *Procedia Comput. Sci.* 2015, 70, 593–602. [CrossRef]

26. Latiu, G.I.; Cret, O.A.; Vacariu, L. Automatic Test Data Generation for Software Path Testing Using Evolutionary Algorithms. In Proceedings of the 2012 Third International Conference on Emerging Intelligent Data and Web Technologies, Bucharest, Romania, 19–21 September 2012; pp. 1–8. [CrossRef]

27. Zhou, Z.Q.; Sinaga, A.; Susilo, W.; Zhao, L.; Cai, K.-Y. A cost-effective software testing strategy employing online feedback information. *Inf. Sci.* 2018, 422, 318–335. [CrossRef]

28. Gou, X.; Huang, T.; Yang, S.; Su, M.; Zeng, F. Optimized Differential Evolution Algorithm for Software Testing. *Int. J. Comput. Intell. Syst.* 2018, 12, 215–226. [CrossRef]

29. Lin, H.; Wang, Y.; Gong, Y.; Jin, D. Domain-RIP Analysis: A Technique for Analyzing Mutation Stubbornness. *IEEE Access* 2018, 7, 4006–4023. [CrossRef]

30. Arcuri, A.; Fraser, G. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empir. Softw. Eng.* 2013, 18, 594–623. [CrossRef]

31. Roeva, O.; Fidanova, S.; Paprzycki, M. Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling. In Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, Krakow, Poland, 8–11 September 2013; pp. 371–376.

32. Mora-Melià, D.; Martínez-Solano, F.J.; Iglesias-Rey, P.L.; Gutiérrez-Bahamondes, J.H. Population Size Influence on the Efficiency of Evolutionary Algorithms to Design Water Networks. *Procedia Eng.* 2017, 186, 341–348. [CrossRef]

33. Shimin, L.; Zhangang, W. Genetic Algorithm and its Application in the path-oriented test data automatic generation. *Procedia Eng.* 2011, 15, 1186–1190. [CrossRef]

34. Chehouri, A.; Younes, R.; Khoder, J.; Perron, J.; Ilincà, A. A Selection Process for Genetic Algorithm Using Clustering Analysis. *Algorithms* 2017, 10, 123. [CrossRef]

35. Guo, H.; Feng, Y.; Hao, F.; Zhong, S.; Li, S. Dynamic Fuzzy Logic Control of Genetic Algorithm Probabilities. *J. Comput.* 2014, 9, 22–27. [CrossRef]

36. Croitoru, N.-E. High-Probability Mutation in Basic Genetic Algorithms. In Proceedings of the 2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 22–25 September 2014; pp. 301–305. [CrossRef]

37. Dassanayake, P. Effect of Mutation and Effective Use of Mutation in Genetic Algorithm. In Proceedings of the ITRU Research Symposium, University of Moratuwa, Moratuwa, Sri Lanka, 16–17 November 2015; pp. 21–24.

38. Abdioun, O.; Abouchabaka, J.; Tajani, C. Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem. *Problem. Int. J. Emerg. Sci* 2012, 2, 61–77.
39. Deb, K.; Deb, A. Analysing mutation schemes for real-parameter genetic algorithms. *Int. J. Artif. Intell. Soft Comput.* **2014**, *4*, 1–28. [CrossRef]

40. Aston, E.; Channon, A.; Belavkin, R.V.; Gifford, D.R.; Krašovec, R.; Knight, C.G. Critical Mutation Rate has an Exponential Dependence on Population Size for Eukaryotic-length Genomes with Crossover. *Sci. Rep.* **2017**, *7*, 15519. [CrossRef]

41. Kuehn, M.; Severin, T.; Salzwedel, H. Variable Mutation Rate at Genetic Algorithms: Introduction of Chromosome Fitness in Connection with Multi-Chromosome Representation. *Int. J. Comput. Appl.* **2013**, *72*, 31–38. [CrossRef]

42. Ammann, P.; Offutt, J. *Introduction to Software Testing*, 2nd ed.; Cambridge University Press (CUP): New York, NY, USA, 2016; pp. 18–19.

43. Ghiduk, A.S. Automatic Generation of Object-Oriented Tests with a Multistage-Based Genetic Algorithm. *J. Comput.* **2010**, *5*, 1560–1569. [CrossRef]

44. Ghiduk, A.S. Testing the Object-Oriented Programs Using a Multi-Stage Genetic Algorithm. In Proceedings of the 2nd International Conference on Computer Science and Its Applications, Jeju, Korea, 1 July 2010. [CrossRef]

**Publisher’s Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).