electronics

MDPI

Article

Multiple-Searching Genetic Algorithm for Whole Test Suites

Wanida Khamprapai [1,2], Cheng-Fa Tsai [2,*], Paohsi Wang [3] and Chi-En Tsai [4]

[1] Department of Tropical Agriculture and International Cooperation, National Pingtung University of Science and Technology, Pingtung 91201, Taiwan; wanida.kpp@gmail.com
[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
[4] Department of Multimedia Business Unit II, Realtek Semiconductor Corporation, Hsinchu 30076, Taiwan; t82327@gmail.com
* Correspondence: cftsai@mail.npust.edu.tw; Tel.: +886-08-770-3201 (ext. 7906)

Abstract: A test suite is a set of test cases that evaluate the quality of software. The aim of whole test suite generation is to create test cases with the highest coverage scores possible. This study investigated the efficiency of a multiple-searching genetic algorithm (MSGA) for whole test suite generation. In previous works, the MSGA has been effectively used in multicast routing of a network system and in the generation of test cases on individual coverage criteria for small- to medium-sized programs. The performance of the algorithms varies depending on the problem instances. In this experiment were generated whole test suites for complex programs. The MSGA was expanded in the EvoSuite test generation tool and compared with the available algorithms on EvoSuite in terms of the number of test cases, the number of statements, mutation score, and coverage score. All algorithms were evaluated on 14 problem instances with different corpus to satisfy multiple coverage criteria. The problem instances were Java open-source projects. Findings demonstrate that the MSGA generated test cases reached greater coverage scores and detected a larger number of faults in the test class when compared with the others.

Keywords: search-based software engineering; software testing; genetic algorithm; EvoSuite tool; multiple coverage

Citation: Khamprapai, W.; Tsai, C.-F.; Wang, P.; Tsai, C.-E. Multiple-Searching Genetic Algorithm for Whole Test Suites. *Electronics* **2021**, *10*, 2011. https://doi.org/10.3390/electronics10162011

Academic Editors: Stefanos Kollias, Juan M. Corchado and Javid Taheri

Received: 12 July 2021
Accepted: 18 August 2021
Published: 19 August 2021

1. Introduction

In software testing, genetic algorithms (GAs) have successfully achieved the maximum coverage criterion in object-oriented classes [1–4]. The coverage criterion is a definition for which parts of the program are to be executed and whether the set of test cases can be executed as defined [5,6]. There are a variety of coverage criteria for testing (e.g., branch, line, mutation). EvoSuite [7] is an automatic tool to generate whole test suites for Java classes using GA. The whole test suite is a set of generated test cases to cover various coverage criteria at a time. Traditionally, a test case is generated to satisfy the individual coverage criterion [8].

A multiple-searching genetic algorithm (MSGA) [9] is an improved GA that has various chromosomes to increase the performance of search solutions. The MSGA has been successfully applied in the multicast routing of a network system [9] and in generating test cases in software testing [10]. The MSGA has been utilized in software testing to generate test cases for testing small- to medium-sized programs. Those programs are written in C language. The set of test cases was created satisfying the branch coverage criterion. Their results indicated that the MSGA generates fewer cases but achieves a better percentage of coverage than the traditional GA. Nevertheless, the question remains whether the MSGA would be suitable for programs developed in another language or other complex programs. The techniques are sufficient to generate test cases for small-sized programs but may

not succeed with complex programs [11,12]. Consequently, the aim of this study was to evaluate test case generation of the MSGA for whole test suites of Java classes. The MSGA was extended on EvoSuite. The MSGA was evaluated with 14 open-source Java projects developed by Google and the Apache Software Foundation, and the results were compared to five other algorithms available in EvoSuite: traditional GA, steady-state GA, breeder GA, cellular GA, and random search.

The paper is structured as follows. Section 2 discusses the whole test suite generation including the tool used, the experimental methods, the representation of chromosomes, and the fitness function. Section 3 presents the problem instances and the parameter setting. The experimental results are described in Section 4. The results are discussed in Section 5. Finally, Section 6 concludes the paper.

2. Whole Test Suites

The target of the whole test suite is to optimize the generated test cases to cover a variety of coverage criteria and minimize the number of test cases while maximizing coverage [13,14]. This means the generated test cases can execute more statements in the test class to satisfy various criteria. EvoSuite supports whole test suite generation using the GA. Basically, the GA begins with initial chromosomes and applies genetic operators (selection, crossover, and mutation). The GA repeats until criteria are satisfied, for example, timeout or 100% coverage is reached. However, the timeout criterion is inapplicable for determining which algorithm can generate the fewest number of test cases while achieving maximum coverage [15]; because test case generation is stopped at a predetermined timeout, the number of generated test cases may be insufficient to provide maximum coverage. Therefore, it will focus on generating test cases to obtain maximum coverage and to find as many faults as possible. This section describes the representation of chromosomes, the fitness function, and the genetic algorithm used in this research.

2.1. Genetic Algorithm

The GA is a popular search-based method that utilizes a process of natural selection. The GA has recently undergone many improvements to suit optimization problems. Basically, the GAs [16,17] simulate the possible solutions of the given problem, called chromosomes. Each chromosome consists of genes. The genes denote characteristics of chromosomes that can be transmitted to the next generation. Each generation comprises chromosomes, called populations. The mechanism of the GA ensures appropriate chromosomes through genetic operators such as selection, crossover, and mutation. The suitability of each chromosome is determined by its fitness value. The chromosomes with high fitness values have an opportunity to be selected as the parent chromosomes [18,19]. Then, the parent chromosomes are processed by crossover and mutation operators. The mechanism of the GA repeats until the optimal solution is found.

The selection operator [20,21] is the process that chooses at least two chromosomes to complete the next steps. Selection has a variety of methods such as roulette wheel, ranking, tournament, and elitism. The crossover operator [22] creates two new chromosomes from the selected chromosome in the previous step. Typically, the crossover point chooses at random within the chromosome. Offspring are created by swapping the genes from the crossover point onwards. The new offspring are added to the population. The mutation operator [18,23] helps to reduce iteration searches and produce various chromosomes. The mutation changes the random element, thus resulting in a new chromosome. Chromosomes are selected, as the mutation operator depends on the probability of mutation.

2.2. Genetic Algorithm for Software Testing

The GA is one of the most widespread methods used in software testing and includes both the traditional GA and the improved GAs. For example, test case generation with hybrid GA, path testing, and mutation testing leads to fewer redundant test cases and the greatest mutation score [24]. To minimize the number of test cases while increasing

test coverage, a population aging process was applied to a typical GA without affecting any of the GA's original parameters [25]. GA is used to improve test case generation for multiple paths, for which GA can find solutions quickly [26]. GA is used to reform formal specifications to generate test data that can detect as many faults as possible [27]. The evidence accumulated suggested that GAs enhance the performance of test cases, allowing them to examine more source code and detect more faults. Furthermore, the usage of GA in software testing decreases testing time because GA can produce test cases automatically.

The processes of a GA for software testing are similar to the process of a general GA, in that it consists of main three processes: selection, crossover, and mutation. The chromosome representation and fitness function differ in each problem. This experiment tested the java source code and used EvoSuite to evaluate the performance of algorithms. Therefore, the chromosome representation and fitness functions are shown in Sections 2.3 and 2.4, respectively.

2.3. Chromosome Representation

The chromosome representation determines the structure of a potential solution in the GA [28,29]. A solution is called a chromosome, and a set of chromosomes is a population. In generating test suites, the chromosome represents a list of statements to be executed in software or the test class, called test cases. A set of test cases is a test suite. The length of each test case depends on the number of statements. A statement [30] denotes a new variable declaration or an existing method involving the test class. The mentioned variables of method calls must be in the same chromosome. Test cases involve five types of statements [14]:

- Primitive statements are the variable declarations with values, include declaring array with size.

```
int defaultParser0 = 24;
String[] stringArray0 = new String[6];
```

- Field statements are the statements that refer to variables of objects.

```
SourceMapFormat sourceMapFormat0 = SourceMapFormat.DEFAULT;
```

- Constructor statements involve creating new objects and calling the existing constructor in the test class.

```
Options options0 = new Options();
```

- Method statements concern the existing method involving the test class.

```
Options options1 = options0.addOption("", true, (String) null);
Precision.round(1.0, Integer.MIN_VALUE);
```

- Assignment statements assign values to variables of objects or array elements.

```
stringArray0[0] = "q";
```

Figure 1 displays the chromosome representation for the Grade class. From the source code, the array of float objects is declared and called in the next line. This means that the float array variable is declared in the test case to test some statements of the Grade class. Therefore, the statement in the test case can be a primitive statement as follows:

```
float[] floatArray0 = new float[6];
```

The generated test suite for the Grade class is composed of two test cases or two chromosomes. Each test case has a different list of statements depending on the number of variables and values used in each test. For example, Test case 1 contains five statements but EvoSuite only counts three statements because it counts declared variables and ignores the assertions. As for Test case 2, it has one statement. The assertions are added after the search-based methods test case generation process [31]. Therefore, the assertions are not counted in the number of statements.

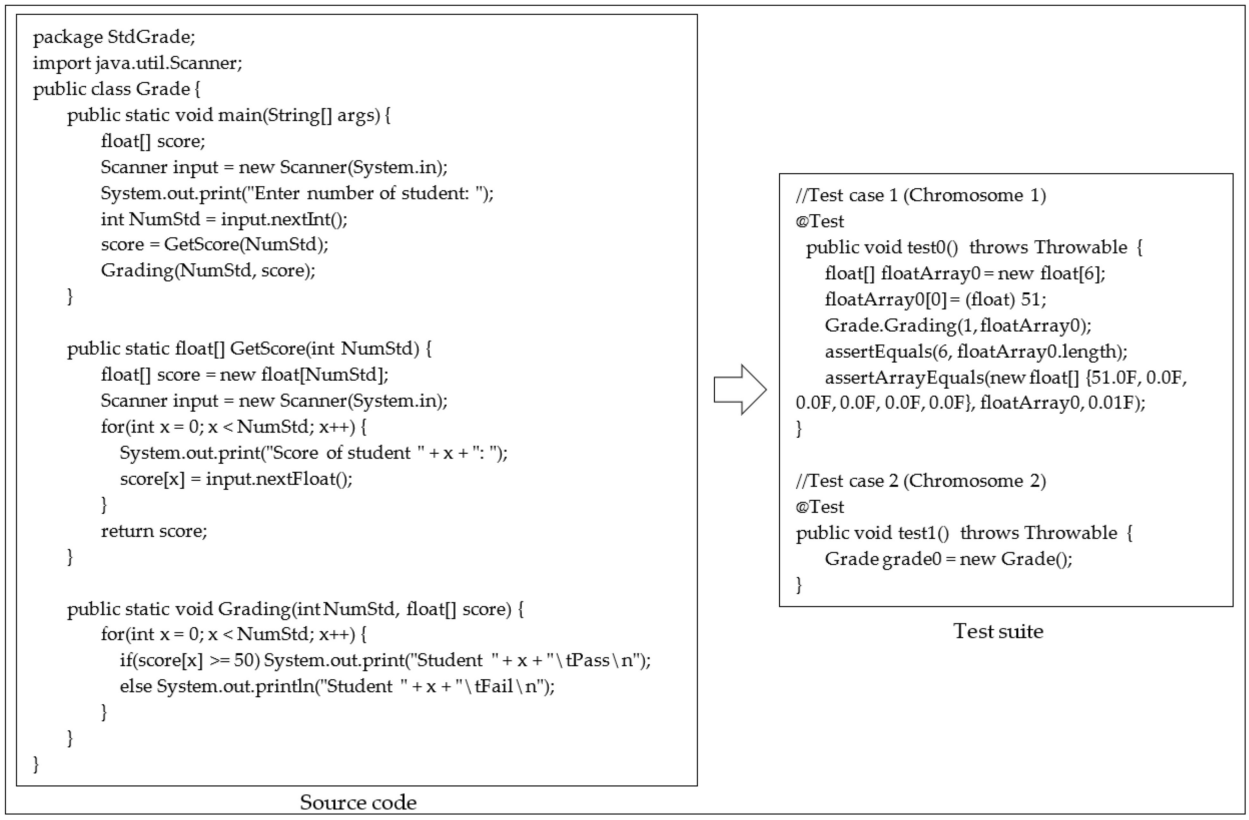


Figure 1. Chromosome representation.

2.4. Fitness Function

The fitness function is a guide to selecting optimal test suites. The fitness function corresponds to a given coverage criterion [32]. The coverage criterion [8] is the rule that defines which parts of the software are tested in test suites. The fitness function is a measurement as to whether test suites satisfy a given coverage criterion. This experiment focused on multiple coverage criteria that are the existing criteria in EvoSuite. There are eight coverage criteria that are differentiated by the fitness function [33,34].

- Line coverage is a basic criterion used to measure how many test suites can execute statement lines. The statements must be reached at least once. These statement lines exclude the comment lines. As a consequence, the fitness function for line coverage is computed according to Equation (1).

$$f(T_{LC}) = |SL| - |SL_T| + \sum_{b \in B} d_{min}(b, T) \quad (1)$$

where SL is the number of statement lines excluding comment lines, SL_T represents the number of statement lines executed in test suite T, and $d_{min}(b,T)$ denotes the minimum branch distance of branch b in a set of branches B that is executed on test suite T. The branch distance [4] demonstrates how close the test suite was to the desired outcome.

- Branch coverage aims to cover control statements, be it the decision-making or loop statements. The control statements are executed to obtain outcomes both true and false. This means at least one test suite executes the control statement to obtain a true result, and at least one more time to obtain a false result. Therefore, the fitness function measures how many test suites can reach control statements. The fitness function for branch coverage is defined as Equation (2).

$$f(T_{BC}) = \sum_{b \in B} d(b, T) \quad (2)$$

where *d(b,T)* indicates the branch distance of branch b in a set of branches B that is reached with test suite T. Equation (3) is the condition to obtain the value of the branch distance *d(b,T)*. In the equation, dmin is a minimal branch distance covered by T.

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ d_{min}(b,T) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \quad (3)$$

- Direct branch coverage includes the control statements in methods that are called directly by test suites. With the execution of the control statements through an indirect method call, it is difficult to cover those control statements. The indirect method call is a calling method within another method. The fitness function of direct branch coverage uses the same $f(T_{BC})$ as that of branch coverage but only focuses on the control statements in methods directly called.
- Exception coverage involves handling exceptions in the test class. The exception is some problems that occur at runtime. Therefore, generated test suites must create exceptions in the test class and throw them. The fitness function of exception coverage cannot be computed as a percentage because some exceptions are unintended, undeclared, or thrown to the external method. Equation (4) defines the fitness function of exception coverage:

$$f(T_{EC}) = \frac{1}{1 + N_E} \quad (4)$$

where N_E is the number of all unique exceptions that test suites can only discover in the test class.

- Weak mutation coverage involves modifying a location in the test class (called mutant) and observing the outcomes of the original and mutant versions. In the event that the outcomes of both are the same, this indicates that the test suites are unable to execute faults or that the mutant is never executed [35,36]. The fitness function of weak mutation coverage is computed using Equation (5):

$$f(T_{WM}) = \sum_{\mu \in M} d(\mu, T) \quad (5)$$

where *d(μ,T)* represents the infection distance of mutant μ in a set of mutants M that is executed with test suite T. The infection distance of the mutant can obtain the value as Equation (6). In the equation, d_{min} denotes a minimal infection distance executed by T.

$$d(\mu, T) = \begin{cases} d_{min}(\mu, T) & \text{if the mutant is reached,} \\ 1 & \text{otherwise.} \end{cases} \quad (6)$$

- Output coverage is used to cover the returned value of the method by means of mapping a return type to abstract values. The abstract values are possible returned values based on the given return data type of method. There is at least one test suite that calls a method in the test class to return a value that corresponds to each abstract value. Equation (7) shows a list of abstract values:

$$V_{abst}(type(M)) = \begin{cases} \{-, 0, +\} & \text{if } type \equiv Number, \\ \{alphabetical, digit, *\} & \text{if } type \equiv Char, \\ \{true,\ false\} & \text{if } type \equiv Boolean, \\ \{null, \neq & \text{otherwise.} \end{cases} \quad (7)$$

where $V_{abst}(type(M))$ is abstract values of a method based on a given return type. The output coverage utilizes abstract values to calculate fitness value as Equation (8):

$$f(T_{OC}) = \sum_{g \in V_{abst}} d(g, T) \quad (8)$$

where g represents an abstract value in a set of abstract values of method that is reached at least once by test suite T.

- Method coverage relates to creating test suites to execute all methods in the test class. The fitness function of method coverage is defined as Equation (9):

$$f(T_{MC}) = |M_T| - |M_E| \tag{9}$$

where M_T is the total number of methods in the test class, and M_E indicates the number of executed methods by test suites in the test class.

- Method coverage (no exception) aims to cover all methods in the test class without throwing an exception. When the method calls and receives the invalid parameters or invalid states, the method will throw the exception. This results in test suites of the method coverage achieving a high fitness value. Consequently, method coverage (no exception) requires that all methods are directly called through test suites and that executions are terminated when they occur. The fitness function of method coverage (no exception) is the same as the fitness function of method coverage.

2.5. EvoSuite

EvoSuite [31] is an automated tool used to generate test cases for Java code. EvoSuite is available for free download at http://www.evosuite.org (accessed on 12 December 2019). EvoSuite can be run from the command line or as plugins in an integrated development environment [37]. In this experiment, EvoSuite was accessed via its Eclipse plugin. Therefore, EvoSuite requires the java bytecode from the test class as input, and test cases are automatically generated. EvoSuite applies several techniques, for example, GAs, chemical reaction optimization, linearly independent path-based search, and random search. Furthermore, EvoSuite supports eight coverage criteria. Testers can choose whether generated test cases satisfy individual or multiple coverage criteria. EvoSuite is a tool that can be extended or modified, such as by adding new techniques and increasing/adjusting genetic operators or fitness functions. Several researchers have utilized EvoSuite to generate whole test suites [14,38–40]. EvoSuite can successfully generate whole test suites using search-based techniques.

In this study, five available algorithms in EvoSuite (four GAs and the random search) were utilized to compare the efficiency for test suite generation of the MSGA. The four GAs are the traditional GA, steady-state GA, breeder GA, and cellular GA. The traditional GA is an unmodified GA that performs only three basic genetic operators. The steady-state GA [41,42] is an improved GA with added processes after the mutation operator is employed. The added processes involve removing the chromosome with the lowest fitness value from the current population. The breeder GA [43,44] chooses the fittest chromosomes with the principle of breeding before performing genetic operators. The cellular GA [45] performs the mutation operator on only one of the best chromosomes, which is selected from the crossed chromosomes.

2.6. Multiple-Searching Genetic Algorithm

The MSGA [9] has improved the GA for multicast routing in network systems. The MSGA adjusts the selection operator of the GA by producing two types of chromosomes: conservative and explorer chromosomes. The conservative chromosomes are similar to the initial chromosomes in GA, but the conservative chromosomes keep only half of the highest fitness chromosomes. The remaining chromosomes are removed. The conservative chromosomes are used to produce the explorer chromosomes. Thereafter, both types of chromosomes are merged. Figure 2 displays the method of creating two types of chromosomes for the MSGA.

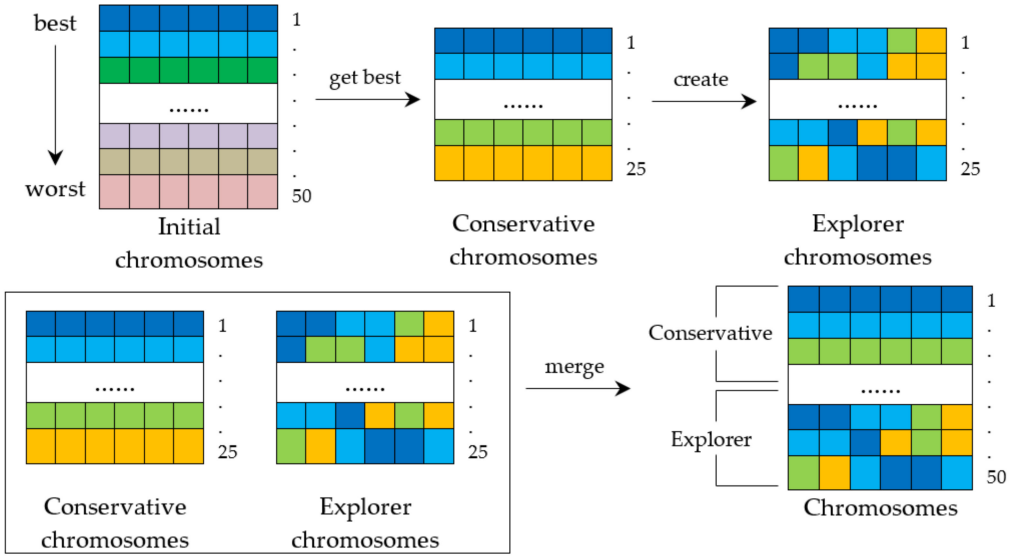


Figure 2. Mechanism for generation of the conservative and explorer chromosomes.

The explorer chromosomes are created using the candidate sets to gather genes of all conservative chromosomes that are in the same position. This means genes in the same position of all conservative chromosomes are collected in the same candidate set. For example, first position genes of all conservative chromosomes are kept in the first candidate set. The explorer chromosome's gene is chosen from each candidate set by selecting only one gene. Gene selection in the candidate can involve selection methods such as roulette wheel or tournament. The selected gene is assigned in the same position on explorer chromosomes. Algorithm 1 outlines the explorer chromosome creation.

Algorithm 1 Pseudocode for explorer chromosomes

```
//Collect genes in each position of conservative chromosomes
for i = 1 to the number of conservative chromosomes
    for j = 1 to the length of chromosomes
    Select ith gene of jth conservative chromosome
    Keep the selected gene to ith candidate set
    end for
end for
//Create explorer chromosomes
for i = 1 to the number of conservative chromosomes
    for j = 1 to the number of candidate set
    Select one gene in jth candidate set
    Keep the selected gene to jth gene of ith explorer chromosome
    end for
end for
return explorer chromosomes
```

The mechanisms for crossover and mutation operators of the MSGA are similar to GA. Typically, the crossover operator involves exchanging genes between two-parent chromosomes, and two offspring chromosomes are obtained for the next generation. The mechanism for the crossover operator of the MSGA is swapping genes between conservative and explorer chromosomes at the position cut points. Thereafter, one offspring is retained in the explorer chromosome, and another is kept in the conservative chromosome. The mutation operator relates to modifying one or more genes. The mutation operator is executed after the crossover operator. The mutation operator of the MSGA separates the mutation probability of conservative and explorer chromosomes. This means the conservative and explorer chromosomes are mutated differently depending on the mutation

probability. Figure 3 illustrates the method of crossover and mutation operators of the MSGA, in which the mutation probability of conservative chromosomes is represented by M_c, and the mutation probability of the explorer chromosomes is indicated by M_e.

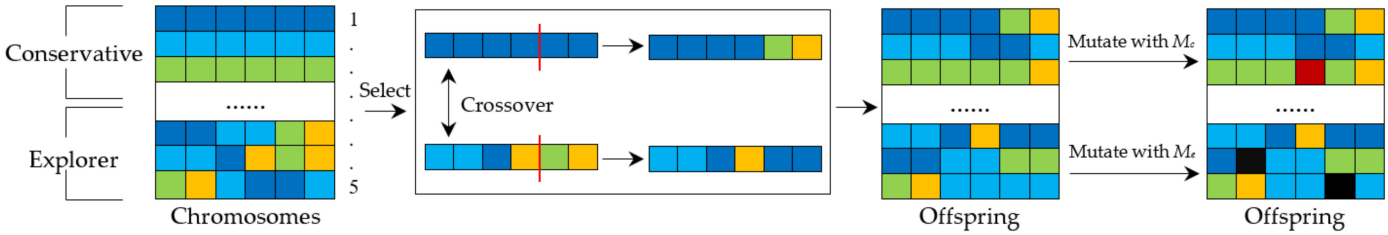


Figure 3. Mechanism for the crossover and mutation operators of the MSGA.

3. Experimental Evaluation

The capability of the MSGA to generate whole test suites was compared with algorithms in EvoSuite. This section details the problem instances and the parameter setting.

3.1. Problem Instance

When evaluating algorithms, the real-world problem instances would be used to minimize threats to external validity. The efficiency of the MSGA was measured by 14 java open-source programs and libraries (Table 1). Some of these libraries and programs were developed by Google and the Apache Software Foundation, because Google and the Apache Software Foundation have a large number of open-source software projects available for download by developers. Furthermore, libraries and program from another corpus were used to ensure that classes were diverse: JGraphT, Joda Time, NanoXML, and Parallel Colt. The 14 problem instances were chosen at random to have a variety of sizes and functionalities, as well as to be drawn from multiple studies [7,46,47]. The largest problem had 65,389 lines, 761 classes, and 66,671 branches (Parallel Colt), whereas the smallest problem consisted of 955 lines, 30 classes, and 178 branches (Java Certificate Transparency).

Table 1. Detail of tested programs.

Case Study	No. of Lines	No. of Classes	No. of Branches
Closure Compiler	102,535	816	15,357
Commons CLI	1480	22	961
Commons Codec	5545	68	3050
Commons Email	1505	20	209
Commons Jelly	4688	95	636
Commons Math3	65,389	918	28,450
Commons Numbers	317	5	225
Guava	52,884	578	16,859
Java Certificate Transparency	955	30	178
JGraphT	26,401	368	12,039
Joda Time	19,441	166	9924
NanoXML	1882	26	738
Parallel Colt	122,923	761	66,671
Truth	4117	58	223
Total	410,062	3931	155,520

Note: The number of non-commenting source lines of code was reported by JavaLOC (https://sourceforge.net/p/locjava/wiki/Home/ (accessed on 28 August 2020)). The number of classes and branches was reported by EvoSuite.

3.2. Parameter Tuning

The parameter setting can influence the efficiency of the GA. The optimal values of parameters depend on problems that cannot define the optimal values for all problems [48]. This experiment used the EvoSuite tool to apply the MSGA to generate whole test suites

of Java classes. The MSGA was extended to EvoSuite and the results compared with the existing algorithms in EvoSuite. Consequently, the parameters of algorithms in this experiment tuned the default values of EvoSuite, whether they be population size, chromosome length, selection, crossover, mutation, fitness function, etc. As Arcuri and Fraser (2013) [49] pointed out, the default values of EvoSuite are sufficient to test the performance of the algorithms. However, the MSGA has two types of chromosomes, and the mutation probabilities for both chromosomes have to differ. Tsai et al. (2004) [9] indicated that obtaining the optimal solution of the MSGA should define a higher mutation probability of the explorer chromosomes than that of the conservative chromosomes. The default of the mutation probabilities in EvoSuite is 0.75. Several researchers [18,50] demonstrated that $1/l$ is an efficient mutation probability, where l is the length of chromosomes. Therefore, the mutation probability of conservative chromosomes was defined as $1/l$ and that of explorer chromosomes was assigned the default value (0.75) because $1/l$ has a smaller value than the default value.

3.3. Evaluation Metrics

The efficiency of algorithms for software testing is the maximum source code reachable and reveals the maximum faults possible of the generated test cases. All algorithms measured the number of test cases, the number of statements, mutation scores, and coverage scores. The metrics of the number of test cases and the number of statements showed how much each algorithm obtained in 60 s. The coverage score counted the total number of source codes accessed by the test cases. The mutation score reported the number of detected faults in the test class. All the experimental findings were examined using a 95% confidence interval for the coverage score and the mutation score, non-parametric Mann–Whitney U tests with a significance level (p-value) of 0.05, the Vargha–Delaney $\hat{A}_{12}$ effect size, and the standard deviation.

4. Experimental Results

In this study, the MSGA was extended on EvoSuite version 1.0.6 to create all of the test suites for the above problem instances. The experimental results are summarized in the dot plot and marginal distribution plots that were created using RStudio version 1.1.383. The experiments were run on Windows 10 Professional (Seattle, WA, USA) 64 with an Intel® Core i7 3.40 GHz CPU and 16 GB of RAM. All six algorithms were independently executed 30 times for each test class. Each run used 60 s timeout. The experiment had $3931 \times 6 \times 30 = 707{,}580$ runs of EvoSuite by executing 3931 classes in the above problem instances and six aforementioned algorithms. Therefore, the computation time of this experiment was at least $707{,}580/(60 \times 24) = 491.375$ days.

The competing algorithms were compared in terms of the number of test cases (#T), the number of statements (#S), mutation score, and coverage score. The generated test suites satisfied multiple criteria: line, branch, direct branch, exception, weak mutation, output, method, and method (no exception). The multiple criteria were the default coverage criteria of EvoSuite. The experimental results were investigated with the Vargha–Delaney $\hat{A}_{12}$ effect size and non-parametric Mann–Whitney U tests with a 5% level of significance (p-value). The relationships between output variables were measured using the correlation coefficients. Furthermore, the experimental results are presented with a 95% confidence interval (CI) and the standard deviation (σ). The experimental results are tabulated in Table 2.

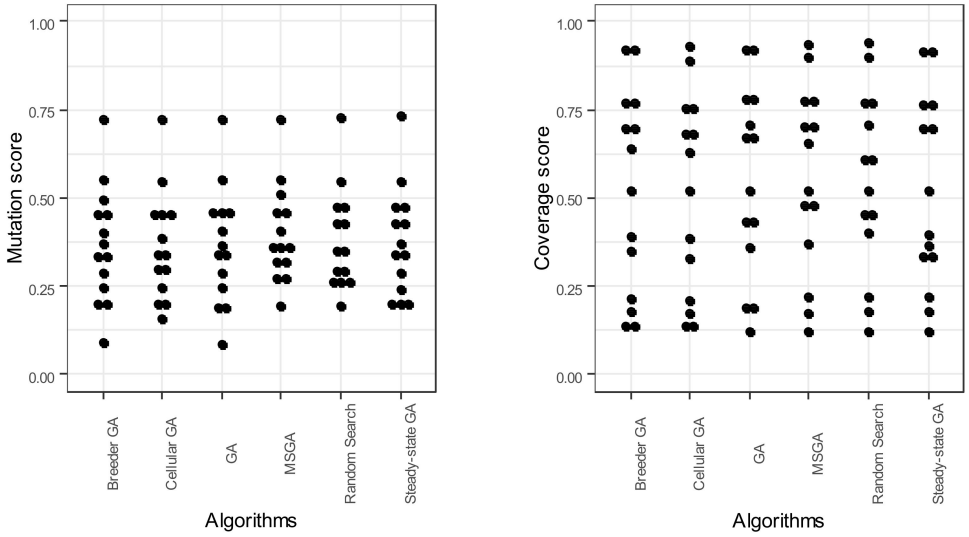
Table 2. Results of generating a whole test suite using each algorithm.

Algorithm	#T	#S	Mutation Score			Coverage Score			*p*-Value	$\hat{A}_{12}$ (MSGA:Others)
			Avg.	CI	σ	Avg.	CI	σ		
MSGA	29,063.5000	115,118.8667	0.3958	(0.3944, 0.3972)	0.0038	0.5567	(0.5556, 0.5578)	0.0029	-	-
Standard GA	19,665.5667	71,463.5000	0.3644	(0.3622, 0.3666)	0.0059	0.5142	(0.5125, 0.5158)	0.0044	<0.00001	1
Steady-State GA	25,156.8000	95,555.2667	0.3761	(0.3745, 0.3777)	0.0042	0.5258	(0.5247, 0.5268)	0.0029	<0.00001	1
Breeder GA	19,896.5667	72,086.6000	0.3668	(0.3657, 0.3680)	0.0031	0.5137	(0.5128, 0.5146)	0.0023	<0.00001	1
Cellular GA	19,722.2000	67,968.6667	0.3633	(0.3614, 0.3652)	0.0051	0.5044	(0.5032, 0.5056)	0.0032	<0.00001	1
Random Search	28,553.0667	119,981.8333	0.3809	(0.3796, 0.3823)	0.0036	0.5404	(0.5392, 0.5416)	0.0032	<0.00001	0.9944

The aim of this study was to assess the efficiency of the whole test suites generated with MSGA compared to the others. The MSGA outperformed all algorithms, which is evident in the number of test cases. The MSGA can generate more test cases than the other algorithms in 60 s. According to Fraser (2018) [31], presenting the number of statements provides a better evaluation than just presenting the number of test cases. The number of statements indicates the total number of instructions that were used to test the test class. Each test case may have a different number of statements. The MSGA produced fewer statements than the random search. When considering the mutation score and coverage score, the MSGA achieved the highest scores. The mutation score [51,52] is a measure of efficiency for the test suites of how many faults can be revealed in the test class. The coverage score [53,54] is the ratio of executed statements based on the given criteria using the test suites. The average mutation score of the MSGA was 0.3958, which shows that the test cases of the MSGA revealed faults 39.58% out of the total number of mutants. With 95% confidence, the average fault detection of the MSGA was between 39.44% and 39.72%. For the coverage score, the MSGA scored the highest average of 0.5567, which means that approximately 55.67% of the source code in the test class was executed using test cases of the MSGA.

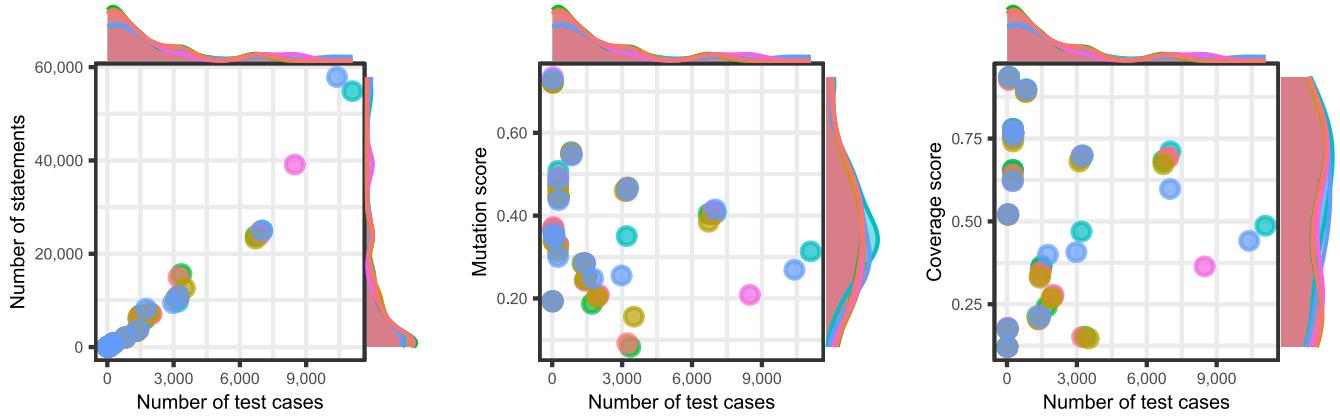
The distributions of the average mutation score and average coverage score achieved by each algorithm are shown in Figure 4. Each dot represents the average scores that each open-source library or program scored. The dot plots of the mutation score and coverage score display very little difference of each algorithm—it is hard to tell how each algorithm differs significantly. Therefore, this study considered the *p*-value and $\hat{A}_{12}$ (Table 2) to compare the efficiency of the MSGA with the others. The results of the comparison indicate all *p*-values < 0.05, which means a statistically significant difference between the MSGA and the others. The $\hat{A}_{12}$ indicates the number of times that the MSGA can perform better than the others. The $\hat{A}_{12}$ of all algorithms had values of more than 0.95. This means that the MSGA beat each algorithm more than 95% of the time. $\hat{A}_{12}$ = 0.9944 demonstrates that 99.44% of the time, the MSGA outperformed the random search.

Figure 5 and Table 3 represent the association between pairs of output variables, the number of test cases, the number of statements, mutation score, and coverage score. The highest statistical correlation (0.9755) was found between the number of test cases and the number of statements, taking into account six algorithms and 14 open-source programs and libraries. In general, the number of test cases was highly correlated with the number of statements. The trend in Figure 5a clearly demonstrates that the number of test cases is higher, and a higher number of statements is obtained. The relationships between pairs of other variables are negligible, with a coefficient of <0.3. This means that when the number of test cases or the number of statements increases, there is no tendency for the mutation score or the coverage score to increase or decrease. Some scattering can be observed from the dots in Figure 5b–e. The association between the mutation score and the coverage score exhibits a low correlation because coefficients of >0.3 and <0.5 (see Figure 5f and Table 3).

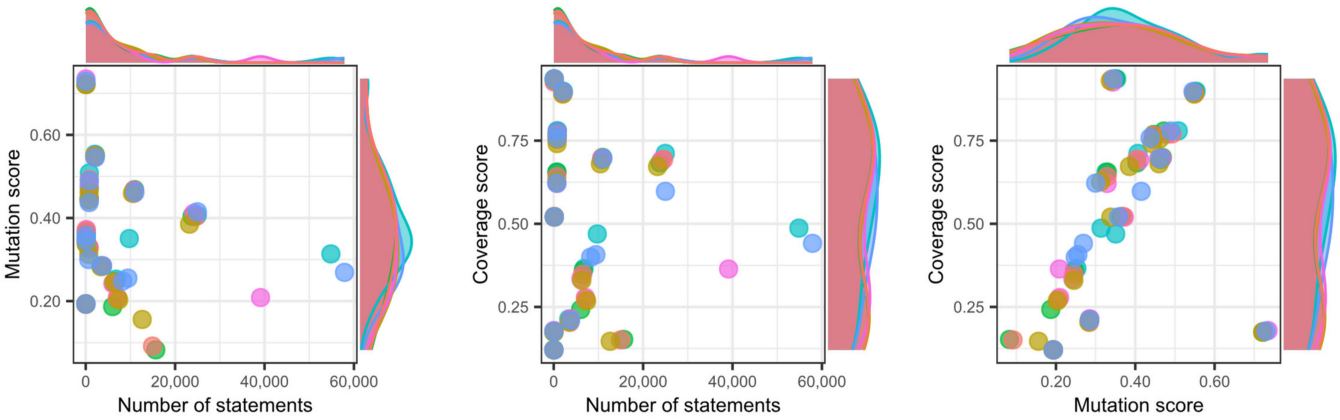


(**a**) Mutation score (**b**) Coverage score

Figure 4. Average scores of mutations and coverages for each algorithm.



(**a**) Number of test cases and number of statements (**b**) Number of test cases and mutation score (**c**) Number of test cases and coverage score



(**d**) Number of statements and mutation score (**e**) Number of statements and coverage score (**f**) Mutation score and coverage score

Figure 5. Correlation of pairs of output variables.

Table 3. Correlation coefficient matrix for output variables.

Correlation Matrix	#T	#S	Mutation Score	Coverage Score
#T	1.0000	0.9755	−0.1786	−0.0149
#S	-	1.0000	−0.2050	−0.0487
Mutation score	-	-	1.0000	0.3552
Coverage score	-	-	-	1.0000

5. Discussion

The purpose of the study was to examine the efficiency of the MSGA for generating whole test suites. The MSGA was extended into the EvoSuite tool, and the performance was compared with five existing algorithms in EvoSuite. The findings indicate that the MSGA outperforms the competition in terms of mutation score and coverage score. As regards the number of test cases, the MSGA obtained the highest number. Many researchers stress decreasing the number of test cases to reduce the time for the overall testing [55–57]. In this experiment, the test cases of all algorithms were automatically produced through the EvoSuite tool in a limited time of 60 s. In addition, Gay (2017) [58] claims test cases are generated with an automatic tool and that the time is reduced. This means that the MSGA generates test cases faster than the aforementioned algorithms, which decreases the time for the overall testing. In addition, these findings confirm previous results [59,60] on the relationship between the output variables. They indicated that the number of test cases is not correlated with the mutation score or coverage score, though the mutation score and the coverage score are slightly related.

There were several threats to the validity of this experiment. First, threats to internal validity relate to parameter tuning and the result collection. In this study, all algorithms were assigned parameters with the default values of EvoSuite. Each algorithm was independently executed 30 times with the same tool to investigate the accuracy of the results. All experimental results were collected using EvoSuite: number of test cases, number of statements, mutation score, and coverage score. Second, threats to external validity involve a definition of instances and the scope of experimental analysis. The problem instances in this study were chosen from programs and libraries developed by Google and the Apache Software Foundation. A total of 14 instances were selected based on previous research studies. These instances are open-source Java projects that are widely available for programmers to download. In this experiment, a total of 3931 classes were applied. The results are limited to the algorithms utilized in the experiment.

6. Conclusions

Test suite generation has been a major topic in software testing research in creating efficient test cases. By investigating the performance of the MSGA to generate whole test suites for Java classes and complex programs, this experiment established that test cases of the MSGA revealed a larger number of faults and executed more instructions in the test class than other algorithms. These experimental results obtained values from defining the same parameter setting of all algorithms. In addition, all algorithms were run on the same tool and the same computer to avoid bias of the results. This suggests that the MSGA is an improved GA for finding routes in the network system but can also be reused to generate whole test suites in software testing. In general, the same parameter tuning may limit the performance of the algorithm. Each algorithm has different optimal parameter values depending on the problem instances. The results in this experiment are just a preliminary efficacy test for generating a whole test suite. Consequently, MSGA should be assigned with appropriate values to generate test cases that are more efficient at finding faults and testing the source code. Furthermore, alternative coverage criterion selections and correlations with software system use resulted in different outcomes. Future research into whole test suite generation using the MSGA should focus on improving the performance to detect more faults and reach more statements in the test class by adding

processes to choose the best chromosomes for the next generation or integration with the other methods.

Author Contributions: Conceptualization, W.K. and C.-F.T.; methodology, W.K. and C.-F.T.; software, W.K.; validation, P.W. and C.-E.T.; formal analysis, W.K.; investigation, W.K. and C.-F.T.; resources, W.K.; data curation, P.W. and C.-E.T.; 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-108-2637-E-020-003, MOST-108-2321-B-020-003, and MOST-109-2637-E-020-003.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The proposed algorithm in this study including source code and results are available upon request from the corresponding author.

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. We also thank the staff of 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. It is their kind help and support that have made to complete this research.

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

References

1. Jaffari, A.; Yoo, C.J.; Lee, J. Automatic Test Data Generation Using the Activity Diagram and Search-Based Technique. *Appl. Sci.* **2020**, *10*, 3397. [CrossRef]
2. Sato, V. Specification-based Test Case Generation with Constrained Genetic Programming. In Proceedings of the 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, 11–14 December 2020; pp. 98–103. [CrossRef]
3. Vats, P.; Mandot, M.; Mukherjee, S.; Sharma, N. Test Case Prioritization & Selection for an Object Oriented Software using Genetic Algorithm. *Int. J. Eng. Adv. Technol.* **2020**, *9*, 349–354. [CrossRef]
4. Shamshiri, S.; Rojas, J.M.; Gazzola, L.; Fraser, G.; McMinn, P.; Mariani, L.; Arcuri, A. Random or Evolutionary Search for Object-Oriented Test Suite Generation? *Softw. Test. Verif. Reliab.* **2017**, *28*, e1660. [CrossRef]
5. Marijan, D.; Gotlieb, A. Software Testing for Machine Learning. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; pp. 13576–13582. [CrossRef]
6. Salman, Y.D.; Hashim, N.D.; Rejab, M.M.; Romli, R.; Mohd, H. Coverage Criteria for Test Case Generation Using UML State Chart Diagram. *AIP Conf. Proc.* **2017**, *1891*, 020125. [CrossRef]
7. Fraser, G.; Arcuri, A. Whole test suite generation. *IEEE Trans. Softw. Eng.* **2013**, *39*, 276–291. [CrossRef]
8. Amman, P.; Offutt, J. *Introduction to Software Testing*, 2nd ed.; Cambridge University Press: New York, NY, USA, 2016; pp. 17–19.
9. Tsai, C.F.; Tsai, C.W.; Chen, C. A novel algorithm for multimedia multicast routing in a large scale network. *J. Syst. Softw.* **2004**, *72*, 431–441. [CrossRef]
10. Khamprapai, W.; Tsai, C.F.; Wang, P. Analyzing the Performance of the Multiple-Searching Genetic Algorithm to Generate Test Cases. *Appl. Sci.* **2020**, *10*, 7264. [CrossRef]
11. Fraser, G.; Arcuri, A. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* **2014**, *24*, 1–42. [CrossRef]
12. Arcuri, A.; Iqbal, M.Z.; Briand, L. Random Testing: Theoretical Results and Practical Implications. *IEEE Trans. Softw. Eng.* **2012**, *38*, 258–277. [CrossRef]
13. Rojas, J.M.; Vivanti, M.; Arcuri, A.; Fraser, G. A detailed investigation of the effectiveness of whole test suite generation. *Empir. Softw. Eng.* **2017**, *22*, 852–893. [CrossRef]
14. Fraser, G.; Arcuri, A.; McMinn, P. A Memetic Algorithm for whole test suite generation. *J. Syst. Softw.* **2015**, *103*, 311–327. [CrossRef]
15. Fraser, G.; Arcuri, A. EvoSuite: On The Challenges of Test Case Generation in the Real World. In Proceedings of the sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–22 March 2013; pp. 362–369. [CrossRef]

16. Cui, X.; Yang, J.; Li, J.; Wu, C. Improved Genetic Algorithm to Optimize the Wi-Fi Indoor Positioning Based on Artificial Neural Network. *IEEE Access* **2020**, *8*, 74914–74921. [CrossRef]
17. Rivera, G.; Cisneros, L.; Sanchez-Solis, P.; Rangel-Valdez, N.; Rodas-Osollo, J. Genetic Algorithm for Scheduling Optimization Considering Heterogeneous Containers: A Real-World Case Study. *Axioms* **2020**, *9*, 27. [CrossRef]
18. Hassanat, A.; Almohammadi, K.; Alkafaween, E.; Abunawas, E.; Hammouri, A.; Prasath, V.B.S. Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach. *Information* **2019**, *10*, 390. [CrossRef]
19. Li, X.; Wang, Z.; Sun, Y.; Zhou, S.; Xu, Y.; Tan, G. Genetic algorithm-based content distribution strategy for F-RAN architectures. *ETRI J.* **2019**, *41*, 348–357. [CrossRef]
20. Drachal, K.; Pawłowski, M. A Review of the Applications of Genetic Algorithms to Forecasting Prices of Commodities. *Economies* **2021**, *9*, 6. [CrossRef]
21. Chiesa, M.; Maioli, G.; Colombo, G.I.; Piacentini, L. GARS: Genetic Algorithm for the identification of a Robust Subset of features in high-dimensional datasets. *BMC Bioinform.* **2020**, *21*, 54. [CrossRef]
22. Hardi, S.M.; Zarlis, M.; Effendi, S.; Lydia, M.S. Taxonomy Genetic Algorithm for Implementation Partially Mapped Crossover in Travelling Salesman Problem. *J. Phys. Conf. Ser.* **2020**, *1641*, 012104. [CrossRef]
23. Wang, J.; Zhang, M.; Ersoy, O.K.; Sun, K.; Bi, Y. An Improved Real-Coded Genetic Algorithm Using the Heuristical Normal Distribution and Direction-Based Crossover. *Comput. Intel. Neurosc.* **2019**, *2019*, 4243853. [CrossRef] [PubMed]
24. Mishra, D.B.; Mishra, R.; Acharya, A.A.; Das, K.N. Test Data Generation for Mutation Testing Using Genetic Algorithm. In *Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing*; Bansal, J., Das, K., Nagar, A., Deep, K., Ojha, A., Eds.; Springer: Cham, Switzerland, 2019; Volume 817, pp. 857–867. [CrossRef]
25. 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]
26. Zhu, Z.; Xu, X.; Jiao, L. Improved evolutionary generation of test data for multiple paths in search-based software testing. In Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC), Donostia, Spain, 5–8 June 2017; pp. 612–620. [CrossRef]
27. Wang, R.; Sato, Y.; Liu, S. Mutated Specification-Based Test Data Generation with a Genetic Algorithm. *Mathematics* **2021**, *9*, 331. [CrossRef]
28. Albadr, M.A.; Tiun, S.; Ayob, M.; AL-Dhief, F. Genetic Algorithm Based on Natural Selection Theory for Optimization Problems. *Symmetry* **2020**, *12*, 1758. [CrossRef]
29. Wang, Y.M.; Zhao, G.Z.; Yin, H.L. Genetic algorithm with three dimensional chromosome for large scale scheduling problems. In Proceedings of the 10th World Congress on Intelligent Control and Automation, Beijing, China, 6–8 July 2012; pp. 362–367. [CrossRef]
30. Fraser, G.; Zeller, A. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Trans. Softw. Eng.* **2012**, *38*, 278–292. [CrossRef]
31. Fraser, G. A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator. In *Search-Based Software Engineering*; Colanzi, T., McMinn, P., Eds.; Springer: Cham, Switzerland, 2018; Volume 11036, pp. 106–130. [CrossRef]
32. Salahirad, A.; Almulla, H.; Gay, G. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Softw. Test. Verif. Reliab.* **2019**, *29*, e1701. [CrossRef]
33. Rojas, J.M.; Campos, J.; Vivanti, M.; Fraser, G.; Arcuri, A. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Search-Based Software Engineering*; Barros, M., Labiche, Y., Eds.; Springer: Cham, Switzerland, 2015; Volume 9275, pp. 93–108. [CrossRef]
34. Gay, G. The Fitness Function for the Job: Search-Based Generation of Test Suites that Detect Real Faults. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, 13–17 March 2017; pp. 345–355. [CrossRef]
35. Shin, K.W.; Lim, D.J. Model-Based Test Case Prioritization Using an Alternating Variable Method for Regression Testing of a UML-Based Model. *Appl. Sci.* **2020**, *10*, 7537. [CrossRef]
36. Hariri, F.; Shi, A. SRCIROR: A toolset for mutation testing of C source code and LLVM intermediate representation. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 860–863. [CrossRef]
37. Fraser, G.; Rojas, J.M.; Campos, J.; Arcuri, A. EVOSUITE at the SBST 2017 Tool Competition. In Proceedings of the 10th International Workshop on Search-Based Software Testing, Buenos Aires, Argentina, 22–23 May 2017; pp. 39–41. [CrossRef]
38. Almasi, M.M.; Hemmati, H.; Fraser, G.; Arcuri, A.; Benefelds, J. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Buenos Aires, Argentina, 20–28 May 2017; pp. 263–272. [CrossRef]
39. Fraser, G.; Arcuri, A. Achieving scalable mutation-based generation of whole test suites. *Empir. Softw. Eng.* **2015**, *20*, 783–812. [CrossRef]
40. Rojas, J.M.; Fraser, G.; Arcuri, A. Seeding strategies in search-based unit test generation. *Softw. Test. Verif. Reliab.* **2016**, *26*, 366–401. [CrossRef]
41. Agapie, A.; Wright, A.H. Theoretical analysis of steady state genetic algorithms. *Appl. Math.* **2014**, *59*, 509–525. [CrossRef]
42. Corus, D.; Oliveto, P. Standard Steady State Genetic Algorithms Can Hillclimb Faster Than Mutation-Only Evolutionary Algorithms. *IEEE Trans. Evol. Comput.* **2018**, *22*, 720–732. [CrossRef]

43. Stoica, F.; Boitor, C.G. Using the Breeder genetic algorithm to optimize a multiple regression analysis model used in prediction of the mesiodistal width of unerupted teeth. *Int. J. Comput. Commun.* **2014**, *9*, 62–70. [CrossRef]
44. Yusran, Y.; Rahman, Y.A.; Gunadin, I.C.; Said, S.M.; Syafaruddin, S. Mesh grid power quality enhancement with synchronous distributed generation: Optimal allocation planning using breeder genetic algorithm. *Prz. Elektrotech.* **2020**, *1*, 84–88. [CrossRef]
45. Osaba, E.; Martinez, A.D.; Lobo, J.L.; Ser, J.D.; Herrera, F. Multifactorial Cellular Genetic Algorithm (MFCGA): Algorithmic Design, Performance Comparison and Genetic Transferability Analysis. In Proceedings of the 2020 IEEE Congress on Evolutionary Computation (CEC), Glasgow, UK, 19–24 July 2020; pp. 1–8. [CrossRef]
46. Grano, G.; Titov, T.V.; Panichella, S.; Gall, H.C. Branch coverage prediction in automated testing. *J. Softw. Evol. Process.* **2019**, *31*, e2158. [CrossRef]
47. Vera-Pérez, O.L.; Danglot, B.; Monperrus, M.; Baudry, B. A comprehensive study of pseudo-tested methods. *Empir. Softw. Eng.* **2019**, *24*, 1195–1225. [CrossRef]
48. Mosayebi, M.; Sodhi, M. Tuning genetic algorithm parameters using design of experiments. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, Cancún, Mexico, 8–12 July 2020; pp. 1937–1944. [CrossRef]
49. 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]
50. 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] [PubMed]
51. Bashir, M.B.; Nadeem, A. Improved Genetic Algorithm to Reduce Mutation Testing Cost. *IEEE Access* **2017**, *5*, 3657–3674. [CrossRef]
52. Li, Y.B.; Sang, H.B.; Xiong, X.; Li, Y.R. An Improved Adaptive Genetic Algorithm for Two-Dimensional Rectangular Packing Problem. *Appl. Sci.* **2021**, *11*, 413. [CrossRef]
53. Lee, J.; Kang, S.; Jung, P. Test coverage criteria for software product line testing: Systematic literature review. *Inf. Softw. Technol.* **2020**, *122*, 106272. [CrossRef]
54. Masri, W.; Zaraket, F.A. Chapter Four—Coverage-Based Software Testing: Beyond Basic Test Requirements. In *Advances in Computers*; Memon, A.M., Ed.; Elsevier: Amsterdam, The Netherlands, 2016; Volume 103, pp. 79–142. [CrossRef]
55. Alian, M.; Suleiman, D.; Shaout, A. Test Case Reduction Techniques—Survey. *Int. J. Adv. Comput. Sci. Appl.* **2016**, *7*, 264–275. [CrossRef]
56. Flemström, D.; Potena, P.; Sundmark, D.; Afzal, S.; Bohlin, M. Similarity-based prioritization of test case automation. *Softw. Qual. J.* **2018**, *26*, 1421–1449. [CrossRef]
57. Jung, P.; Kang, S.; Lee, J. Efficient Regression Testing of Software Product Lines by Reducing Redundant Test Executions. *Appl. Sci.* **2020**, *10*, 8686. [CrossRef]
58. Gay, G. Generating Effective Test Suites by Combining Coverage Criteria. In *Search-Based Software Engineering*; Menzies, T., Petke, J., Eds.; Springer: Cham, Switzerland, 2017; Volume 10452, pp. 65–82. [CrossRef]
59. Antinyan, V.; Derehag, J.; Sandberg, A.; Staron, M. Mythical Unit Test Coverage. *IEEE Softw.* **2018**, *35*, 73–79. [CrossRef]
60. Inozemtseva, L.; Holmes, R. Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 435–445. [CrossRef]