mathematics

MDPI

Article

Performance of Enhanced Multiple-Searching Genetic Algorithm for Test Case Generation in Software Testing

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: Test case generation is an important process in software testing. However, manual generation of test cases is a time-consuming process. Automation can considerably reduce the time required to create adequate test cases for software testing. Genetic algorithms (GAs) are considered to be effective in this regard. The multiple-searching genetic algorithm (MSGA) uses a modified version of the GA to solve the multicast routing problem in network systems. MSGA can be improved to make it suitable for generating test cases. In this paper, a new algorithm called the enhanced multiple-searching genetic algorithm (EMSGA), which involves a few additional processes for selecting the best chromosomes in the GA process, is proposed. The performance of EMSGA was evaluated through comparison with seven different search-based techniques, including random search. All algorithms were implemented in EvoSuite, which is a tool for automatic generation of test cases. The experimental results showed that EMSGA increased the efficiency of testing when compared with conventional algorithms and could detect more faults. Because of its superior performance compared with that of existing algorithms, EMSGA can enable seamless automation of software testing, thereby facilitating the development of different software packages.

Keywords: search-based test case generation; genetic algorithm; branch coverage; object-oriented

check for updates

Citation: Khamprapai, W.; Tsai, C.-F.; Wang, P.; Tsai, C.-E. Performance of Enhanced Multiple-Searching Genetic Algorithm for Test Case Generation in Software Testing. *Mathematics* **2021**, *9*, 1779. https://doi.org/10.3390/math9151779

Academic Editor: Vassilis C. Gerogiannis

Received: 22 June 2021
Accepted: 23 July 2021
Published: 27 July 2021

1. Introduction

Software testing is an important process in the software development life cycle. It is performed to investigate the quality of software and to evaluate the risks in software implementation. Software testing involves both valid and invalid inputs and includes the processes of executing the developed software and checking for the expected responses. Several techniques can be used to automatically produce inputs that conform to the behavior of the software being tested, and these techniques provide high coverage in a given branch, line, condition, or path. Various techniques have been proposed to reduce the cost, resources, and time involved in the testing process.

The genetic algorithm (GA) is a popular and efficient search-based technique for test case generation. GAs have been widely used to create suitable test cases [1–4]. Suitable test case generation helps to reduce costs in software testing given the huge cost of creating test cases, which accounts for more than 50% of the total cost of developing a program [5]. Researchers have investigated methods to enhance the solution efficiencies of GAs. Multiple-searching genetic algorithm (MSGA) [6] is a successfully solved optimal solution with high probability for routing in network system. MSGA is attractive to utilize in other fields. From previous work [7], MSGA can generate test cases for small to medium scale software but cannot increase the percentage of coverage for complex software. This means test cases generated with MSGA cannot increase the number of executed statements or source code in complex software. Therefore, while MSGA may be suitable for generating test cases for small to medium scale software, it may not be flexible enough for test case generation for complex software. Some algorithms may be suitable for generating test cases for small to medium scale software but may not succeed in complex cases. For this reason, we present a new algorithm for improving MSGA to make it suitable for generating test cases. We expect that the test case generation using our algorithm will also detect more errors or faults in the software and therefore reduce the cost of software testing by creating the minimum number of test cases while getting the maximum coverage. Further, our algorithm can create test cases for complex software. In this study, we used MSGA to generate test cases for software testing because MSGA can reach the global optimum faster than a traditional GA [7]. In addition, we refactored the algorithm to solve the problem of executing the source code for more access to the statements.

In this study, a new algorithm called the enhanced multiple-searching genetic algorithm (EMSGA), which is an improved MSGA incorporating some additional processes, was developed. The genetic operators constitute the basic mechanism of the GA, namely selection, crossover, and mutation. Additional processes in EMSGA include the evaluation of chromosomes and selection of the best chromosomes to add to the next generation. In the original MSGA, all the chromosomes that are executed with the genetic operators are added to the next generation. EMSGA was expanded in EvoSuite, and its effectiveness was compared with that of MSGA and seven other techniques available in EvoSuite. The SF110 corpus and nine open-source Java projects developed by Google and the Apache Software Foundation were employed as case studies for generating test cases using the aforementioned algorithms.

The remainder of this paper is organized as follows. Section 2 discusses previous research works related to this study. Section 3 describes search-based techniques for generating test cases, including representation and fitness functions. The proposed algorithm is also introduced in this section. Section 4 presents the problem instances and tools used to evaluate EMSGA. Section 5 presents the experimental results. Section 6 reports threats to the validity of the algorithm. A discussion of the results is presented in Section 7. Finally, Section 8 concludes the paper.

2. Related Work

In software engineering, GA has been successful in many areas, such as software design, effort estimation, and maintenance. For software design [8], GA can help migration from structure programming to object-oriented programming, and the results are better than greedy algorithm and Monte Carlo. In software effort estimation, GA is stable, has higher accuracy than a random approach, and consists of an exhaustive framework [9]. Furthermore, GA is utilized to manage maintenance packages taking into account the cost-effectiveness of the package and to reduce human bias [10].

Various search-based techniques are available for test case generation. GA is one of the most widely used techniques. Many GAs have been remodeled for increased search efficiency. For example, a population aging process was added in a traditional GA without modifying any original parameters of the GA to reduce the number of test cases and increase the test coverage [4]. The features of GA and ant colony optimization (ACO) were combined to increase the efficiency and health of test cases [11]. GA and negative selection algorithms were merged to reduce the generation of duplicate test cases [12]. The results of the studies indicate that these improved algorithms are capable of efficiently generating test cases, even though the algorithms were originally improved for other applications. MSGA is an improved GA for network systems. Even though it was improved for and utilized in another field, we believe that an enhanced version of MSGA can increase the efficiency of test case generation.

EMSGA reuses and refactors existing algorithms. The reusable nature of this algorithm [13] helps to increase the reliability of results, provides faster algorithm development, and reduces costs. Algorithm refactoring is caused by insufficient existing algorithms to perform certain tasks. Consequently, algorithms are improved to suit the task. Algorithm refactoring is challenging in terms of selecting some parts of an algorithm to improve the performance or adding some processes to make it suitable for solving a given problem. Several studies have examined refactoring. For example, Liu et al. (2020) [14] studied automated refactoring for real-time systems to help reduce the effort required by programmers to isolate portions for the execution of real-time systems under limitations. Several researchers have used the SF110 corpus and EvoSuite to compare newly developed algorithms and existing algorithms. For example, the EvoTLBO algorithm was extended into EvoSuite to compare the results with traditional GA and monotonic GA using 50 random classes from SF110 [15]; EvoSuite and SF110 were utilized to compare the performance of memetic algorithm with traditional GA [16]; and nontrivial classes were selected from SF110 to compare the efficiency of the DynaMOSA algorithm with the many objective sorting algorithm (MOSA), the whole suite approach with archive (WSA), and the traditional whole suite approach (WS) [17]. The SF110 corpus is considered as a benchmark for test generation [18]. The SF110 corpus contains 110 Java projects from SourceForge, 100 random projects, and the 10 most popular projects in SourceForge. EvoSuite is an automatic test generation tool for Java classes based on GA. In the present study, the SF110 corpus and EvoSuite were considered sufficient to measure the effectiveness of the proposed algorithm for test case generation. EMSGA was tested using SF110, and its effectiveness was compared with that of seven algorithms available in EvoSuite.

3. Search-Based Test Case Generation

The search-based technique is widely used for test case generation [19–22]. The following subsections describe some of the most well-known search-based techniques before introducing the proposed EMSGA.

3.1. Representation

A population of candidate solutions is represented as a test suite [17,22], which is a collection of test cases $T = \{t_1, t_2, \ldots, t_n\}$. Each test case is composed of various statements $t = \langle s_1, s_2, \ldots, s_l \rangle$, where l is the total number of statements. A statement [23] can be a variable declaration or a method call and can be of several different types, namely a primitive, a constructor, a method, an array, or an assignment.

Figure 1 presents the generated test cases from Java code by considering the required variables and methods to generate statements for testing the class under test. When considering Java code, the integer array variable is a required variable to maintain the numbers for sorting. Therefore, the integer array variable is declared in the test case. The number of statements depends on the instruction to be used for each test. The length of either the test case or the chromosome depends on the number of statements. The population evolves iteratively to yield better solutions. The processes are repeated until a stopping criterion is satisfied.

3.2. Fitness Function

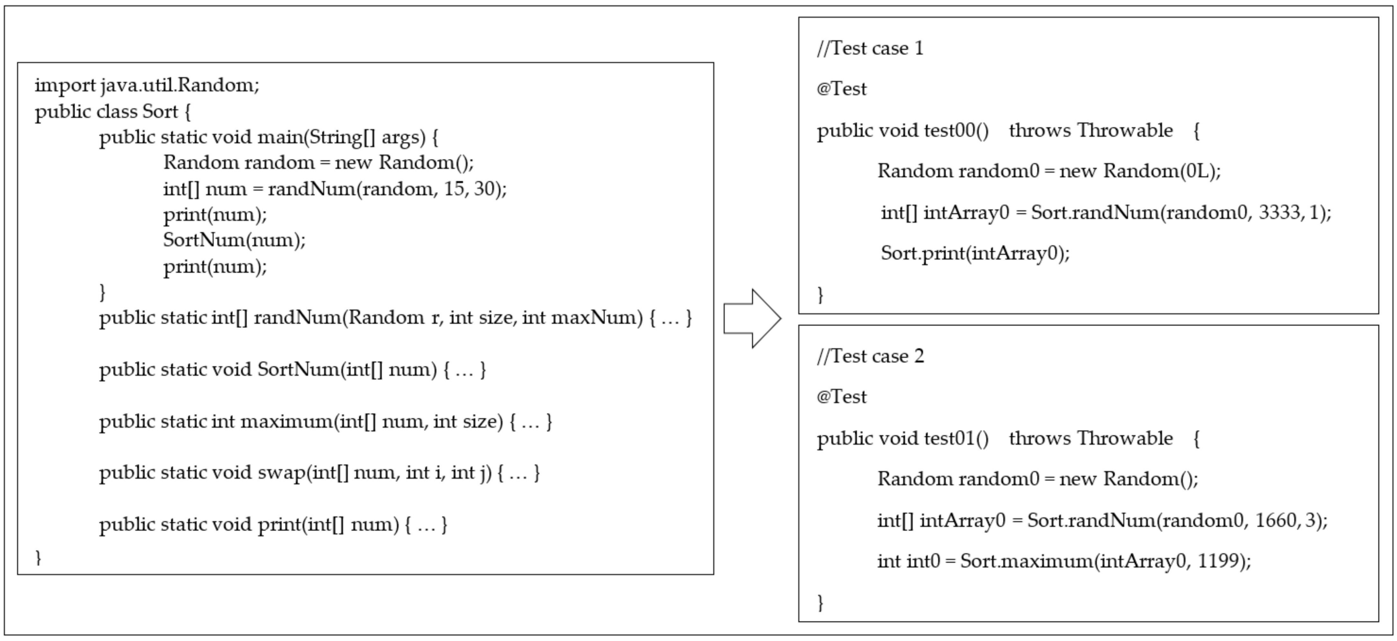
In software testing, a fitness function is used to evaluate the ability of the generated test suites to execute the source code of the program. Typically, fitness functions are assessed based on the branch coverage metric. Complete branch coverage refers to all control structures being executed and all lines of code being tested. This metric is defined as follows [24,25]:

$$f(T) = |M| - |M_T| + \sum_{b \in B} d(b, T), \tag{1}$$

where $|M|$ denotes the total number of methods, $|M_T|$ is the number of methods executed in test suite T, and $d(b, T)$ represents the branch distance for each branch b on test suite

T that b is an element of in a set of branches B. The branch distance $d(b,T)$ is defined as follows:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ d_{min}(b,T) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$



```
import java.util.Random;
public class Sort {
        public static void main(String[] args) {
                Random random = new Random();
                int[] num = randNum(random, 15, 30);
                print(num);
                SortNum(num);
                print(num);
        }
        public static int[] randNum(Random r, int size, int maxNum) { … }

        public static void SortNum(int[] num) { … }

        public static int maximum(int[] num, int size) { … }

        public static void swap(int[] num, int i, int j) { … }

        public static void print(int[] num) { … }
}
```

```
//Test case 1
@Test
public void test00()   throws Throwable   {
        Random random0 = new Random(0L);
        int[] intArray0 = Sort.randNum(random0, 3333, 1);
        Sort.print(intArray0);
}
```

```
//Test case 2
@Test
public void test01()   throws Throwable   {
        Random random0 = new Random();
        int[] intArray0 = Sort.randNum(random0, 1660, 3);
        int int0 = Sort.maximum(intArray0, 1199);
}
```

Figure 1. Generated test cases from source code.

3.3. Genetic Algorithms

GAs [4,26] solve problems through the use of three basic operators: selection, crossover, and mutation. In GA, a chromosome is defined as a set of parameters that represent a proposed solution to the problem that the GA is being used to solve. The selection operator selects certain chromosomes as parent chromosomes. Chromosomes are selected on the basis of their fitness values. Chromosomes with higher fitness values have a higher chance of being selected. The crossover and mutation operators are applied to the parent chromosomes to produce offspring for the next generation. The crossover operator exchanges certain genes of two chromosomes. The mutation operator changes the value of some genes in a few chromosomes.

Several researchers have proposed techniques to improve the traditional GA for enhancing its solution efficiency and enabling its application in complex problems. These efforts have relied on adjustments of factors or integration of GAs with other strategies. For example, the monotonic GA [26] involves additional processes after the mutation process in the traditional GA. These additional processes measure the fitness values to determine the best offspring or the best parent for the next population; in contrast, the traditional GA increases the number of mutated offspring in the next population and then calculates the fitness values of all chromosomes. Another improved version of GA is the steady-state GA [27,28], in which the fitness values of the mutated offspring are determined and then the offspring is compared with the parent. If the offspring is better than the best parent, the offspring replaces the parent in the current population. The advantages of monotonic GA and steady-state GA are similar, namely removing duplicate chromosomes and ensuring the best chromosome is not discarded. A breeder GA [29] differs from the traditional GA in that it uses the principle of breeding, which involves selecting the fittest chromosomes and reproducing using those chromosomes. The breeder GA is more precise as it utilizes the science of breeding [30]. A cellular GA [31] is an improved GA that selects the best offspring after the crossover operator has been applied. The best offspring is mutated, and the fitness value is determined. The selection of cellular GA is restricted to the overlapping neighborhood producing slow solutions [32,33]. Table 1 summarizes the characteristic of each GA.

Table 1. Comparison of GA-based characteristics.

Algorithm	Characteristic of Algorithm
Traditional GA	Applies only three basic operators: selection, crossover, and mutation
Monotonic GA	Still applies three basic operators but adds some processes to select the best chromosome for the next generation.
Steady-state GA	Adds some processes to select the best chromosome. Similar to the monotonic GA but replaces the best chromosome in the current population.
Breeder GA	Applies the principle of breeding to select chromosomes before performing the basic operators.
Cellular GA	Performs mutation operator on only one crossed chromosome. Chromosomes are selected for mutation by choosing at random.

3.4. Chemical Reaction Optimization (CRO)

Chemical reaction optimization (CRO) [34] is a search-based technique that combines the advantages of GA and simulated annealing. CRO solves problems using a set of molecules. Each molecule possesses a molecular structure, potential energy, and kinetic energy. The molecular structure represents a possible solution that does not have any specific format. The potential energy is the fitness value of the corresponding molecule. The kinetic energy quantifies the tolerance of the worst solution. The iterative processes of CRO are similar to those of GA. A basic CRO involves four types of reactions: on-wall ineffective collision and decomposition are reactions where a single molecule hits a wall of the surface, and intermolecular ineffective collision and synthesis are reactions where multiple molecules collide with each other.

On-wall ineffective collision represents a local search. There is minimal change in the structure or properties of the molecule during this process. Decomposition is a type of collision that produces two or more new molecules. This process represents a global search. Intermolecular ineffective collision is the collision of multiple molecules, which produces minimal changes in the structure or properties of the molecules, similar to on-wall ineffective collision. Two or more collided molecules undergo small changes in structure or properties. Synthesis is a reaction that represents a global search. In this reaction, multiple colliding molecules fuse into a single molecule.

3.5. Random Search

Random search is the simplest search-based technique. It involves iterative searches until an optimal solution is obtained. In each iteration, the solution is incremented with a random vector. The fitness value of the modified solution is determined. If the modified solution is better than the previous solution, the former replaces the latter. Otherwise, the previous solution is retained. Random search is often utilized for comparison with other techniques [35]. This technique can effectively solve large-scale problems [36].

4. Proposed Algorithm: Enhanced Multiple-Searching Genetic Algorithm (EMSGA)

In the multiple-searching genetic algorithm (MSGA) introduced by Tsai et al. [6], two types of chromosomes are created to prevent the search from falling into a local optimum. The MSGA utilizes the candidate mechanism to create more chromosomes with the same features, resulting in better chromosomes. The MSGA has been successfully used to find the optimal multicast route in network systems. We believe that the MSGA can also be integrated with other strategies to increase search ability. Therefore, we propose EMSGA, a regeneration MSGA with the addition of a feature-selection strategy. After the mutation operator is employed and the fitness value is determined, only chromosomes from the best offspring or the best parent will be selected to be included in the next-generation population. If the mutated offspring are better than the parents, then they replace the parents in the next generation. Otherwise, the parents are retained. Choosing the best chromosome increases the chances of reaching the optimal solution. Generally, two mutated offspring are added to the next-generation population, and the parents are discarded. The processes involved in EMSGA are similar to those in MSGA, with the exception of the aforementioned best chromosome selection mechanism after the mutation process (Figure 2). Algorithm 1 shows the pseudocode of EMSGA.

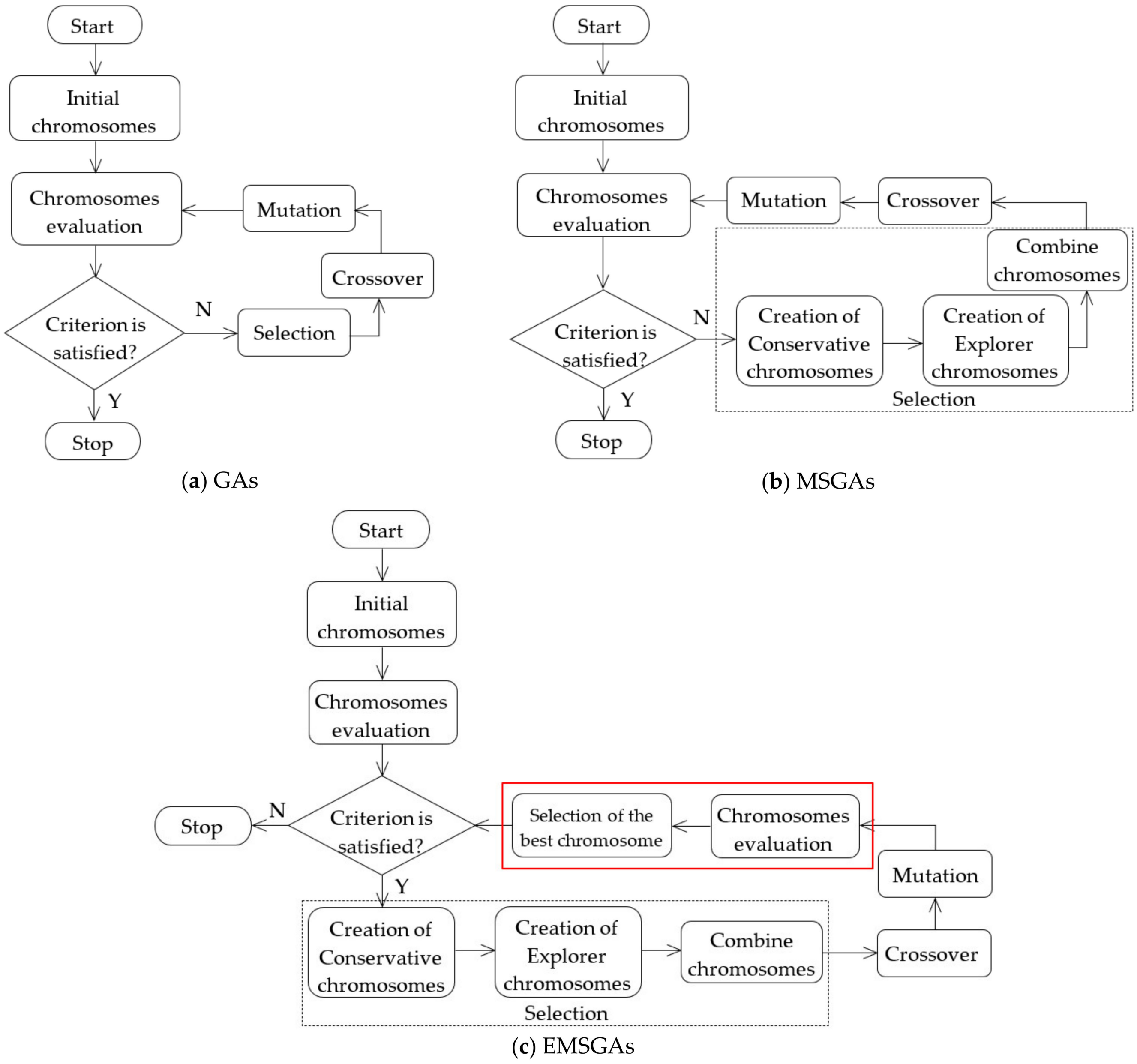


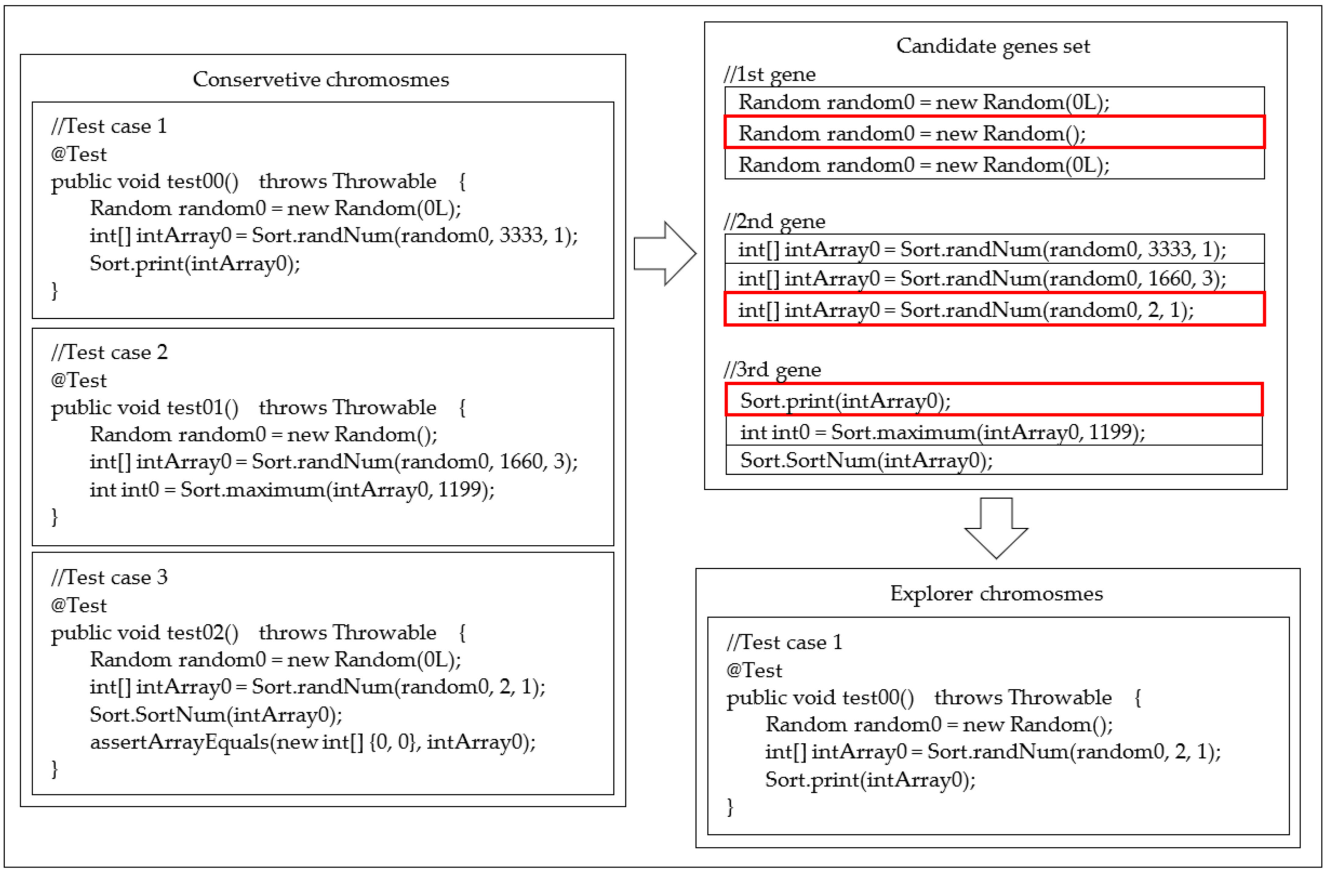
Figure 2. Flowcharts of GA (**a**) [7], MSGA (**b**) [7], and EMSGA (**c**). The red box indicates the additional processes in EMSGA. The black dashed box displays the additional processes in MSGA.

Algorithm 1 Pseudocode for EMSGA

```
1:  Procedure EMSGA()
2:  Create initial chromosomes
3:  Evaluate fitness value of initial chromosomes and order by descending
4:  while not terminal condition do
5:      Select half chromosomes with the highest fitness value //Conservative chromosomes
6:      Call procedure CreateExplorerChromosomes(Conservative chromosomes)
7:      Evaluate fitness value of explorer chromosomes
8:      Combine Conservative and Explorer chromosomes
9:      Call procedure Crossover(all chromosomes)
10:     //Mutation of EMSGA is the same as traditional GA
11:     Mutate Conservative chromosomes with $M_1$
12:     Mutate Explorer chromosomes with $M_2$
13:     Evaluate fitness value of the mutated chromosomes
14:     if the offspring is better than the best parent then
15:         Add offspring in the next population
16:     else
17:         Add parent in the next population
18:     end if
19: end while
20: return chromosomes
21: end procedure
22:
23: Procedure CreateExplorerChromosomes(Conservative chromosomes)
24: for each conservative chromosome $i$
25:     for each gene of conservative chromosome $j$ of $i$
26:         Keep $j$th gene of $i$th conservative chromosome to $j$th candidate gene set
27:     end for
28: end for
29: Creates explorer chromosomes with a number equal to the number of conservative
    chromosomes
30: for each explorer chromosome $i$
31:     for each candidate gene set $j$
32:         Select one gene from $j$th candidate gene set
33:         Preserve the selected gene in $j$th gene of $i$th explorer chromosome
34:     end for
35: end for
36: return explorer chromosomes
37: end procedure
38:
39: Procedure Crossover(all chromosomes)
40: Set a random number $r$
41: if $r$ is less than crossover probability then
42:     for half of all chromosomes from $i = 1$ to (population size / 2)
43:         Select $i$th chromosome and (population size $- i + 1$)th chromosome
44:         Split the selected chromosomes with crossover method
45:         Cross both chromosomes
46:     end for
47: end if
48: return chromosomes
49: end procedure
```

The EMSGA process starts with the creation of initial chromosomes. Then, the fitness value of the population is determined, and half of the chromosomes with the highest fitness values are retained. The rest of the chromosomes are discarded. The preserved chromosomes are called the conservative chromosomes. Next, the candidate mechanism is utilized to build the explorer chromosomes by selecting the genes of the conservative chromosomes.

The candidate mechanism is created to gather genes of all conservative chromosomes that are in the same position into the same candidate gene set. Each candidate gene set selects only one gene to create as a gene of explorer chromosome. Figure 3 illustrates the method for creating an explorer chromosome. Thereafter, crossover and mutation are performed on the conservative and explorer chromosomes separately. Both types of chromosomes are assigned the same crossover probability. The mutation probabilities are defined differently. At the end of each iteration, the chromosomes are evaluated in terms of the fitness value, and the best chromosomes are selected and added to the next-generation population.



```
//Test case 1
@Test
public void test00()   throws Throwable   {
    Random random0 = new Random(0L);
    int[] intArray0 = Sort.randNum(random0, 3333, 1);
    Sort.print(intArray0);
}
```

```
//Test case 2
@Test
public void test01()   throws Throwable   {
    Random random0 = new Random();
    int[] intArray0 = Sort.randNum(random0, 1660, 3);
    int int0 = Sort.maximum(intArray0, 1199);
}
```

```
//Test case 3
@Test
public void test02()   throws Throwable   {
    Random random0 = new Random(0L);
    int[] intArray0 = Sort.randNum(random0, 2, 1);
    Sort.SortNum(intArray0);
    assertArrayEquals(new int[] {0, 0}, intArray0);
}
```

```
//1st gene
Random random0 = new Random(0L);
Random random0 = new Random();
Random random0 = new Random(0L);

//2nd gene
int[] intArray0 = Sort.randNum(random0, 3333, 1);
int[] intArray0 = Sort.randNum(random0, 1660, 3);
int[] intArray0 = Sort.randNum(random0, 2, 1);

//3rd gene
Sort.print(intArray0);
int int0 = Sort.maximum(intArray0, 1199);
Sort.SortNum(intArray0);
```

```
//Test case 1
@Test
public void test00()   throws Throwable   {
    Random random0 = new Random();
    int[] intArray0 = Sort.randNum(random0, 2, 1);
    Sort.print(intArray0);
}
```

Figure 3. Mechanism of creating explorer chromosome. Red boxes demonstrate which one gene from each candidate gene set was chosen.

5. Experimental Evaluation

The aim of this study was to evaluate the capability of EMSGA to generate test cases and to compare the feasibility and effectiveness of EMSGA with those of other algorithms.

5.1. Problem Instances

The selection of problem instances is important for any empirical study on automatic test case generation. This study utilized the SF110 corpus (the details of SF110 are available online: https://www.evosuite.org/experimental-data/sf110/ (accessed on 4 March 2020)) [18] and nine open-source Java projects developed by Google and the Apache Software Foundation to evaluate EMSGA. The SF110 corpus is widely used as a benchmark [17,24,37]. It contains 110 projects that were written with the Java language. Not all classes in the SF110 corpus were employed in this experiment. Only 203 classes were chosen based on the selection in a previous study [38]. Furthermore, nine problem instances from Google and the Apache Software Foundation were chosen uniformly and at random based on their sizes and functionalities (Table 2), consisting of a total of 1382 classes. EvoSuite was applied to a total of 203 + 1382 = 1585 classes.

Table 2. Details of open-source Google and Apache projects. Note: the second column lists the number of non-commenting source lines of code reported by JavaNCSS (http://www.kclee.de/clemens/java/javancss/ (accessed on 10 December 2020)). The fourth column lists the number of branches reported by EvoSuite.

Problem Instances	No. of Lines	No. of Classes	No. of Branches
Java Certificate Transparency	955	30	178
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
Joda-Time	19,441	166	9924
Truth	4117	58	223
Total	103,437	1382	43,856

5.2. Test Generation Tool

The testing tool employed EvoSuite (EvoSuite can be downloaded from http://www.evosuite.org (accessed on 20 February 2020)) [24] to generate test cases for Java code. EvoSuite is widely used in software testing [3,39,40]. It utilizes search-based methods, including genetic algorithms, to generate test cases using Java bytecode. Furthermore, EvoSuite supports various coverage criteria to determine the quality of a solution.

In the experiment, the proposed algorithm was implemented as an extension to the EvoSuite. To extend the new algorithm in Evosuite, a developer must create a new class in the client module and extend the abstract class *GeneticAlgorithm*. The EMSGA class implemented the basic methods for GA that EvoSuite prepares. In addition, the EMSGA class added some processes for creating two types of chromosomes and selected the best chromosome. Test cases of each algorithm were automatically generated, and problem instances were executed through EvoSuite. The performance of EMSGA was compared with that of the MSGA, traditional GA, monotonic GA, steady-state GA, breeder GA, cellular GA, CRO, and random search. These search-based methods are provided in EvoSuite. The coverage achieved by the algorithms was assessed in terms of the branch coverage metric. Search budget configuration uses EvoSuite's default of 60 s [41]. Search budget is the time for generating test cases of the algorithm each time. The experiment was independently repeated 10 times.

The parameter settings influence the performance of search-based methods. The EvoSuite guides the default values (e.g., selection function, crossover function, crossover probability, mutation function, mutation probability, population size, and chromosome length) for test case generation. The default values of EvoSuite are the approximate values that are suitable for generating test cases that are based on GA. Table 3 shows the default values in EvoSuite. The same parameter setting may not be enough to fully extract the efficiency of the algorithm [42]. As Arcuri and Fraser (2013) [43] pointed out, the default values of EvoSuite are sufficient to evaluate the performance of algorithms for test case generation, whereas the suitable parameter setting is time-consuming and may or may not produce good results for algorithms. In addition, Črepinšek et al. (2014) [44] perceptively stated that all algorithms should be examined under the same conditions.

Therefore, the default values for all nine algorithms were used in the experiment. EMSGA assigns different mutation probabilities to the conservative and explorer chromosomes. If the explorer chromosomes are defined as having a higher mutation probability than the conservative chromosomes, the optimal solution can be obtained [6]. Several researchers have set the probability as $1/l$ for the mutation operator, where l is the chromosome length [43,45,46]. Accordingly, mutation probabilities of $1/l$ and 0.75 (default) were used for the conservative and the explorer chromosomes, respectively, in this study.

Table 3. Default values of parameters in Evosuite.

Parameters	Default Values
Population size	50
Chromosome length	40
Selection function	Rank
Crossover function	Single point relative
Crossover probability	0.75
Mutation function	Uniform
Mutation probability	0.75
Search budget	60 s

The experiment involved 1585 $\times$ 9 $\times$ 10 = 142,650 runs of EvoSuite with the aforementioned settings. The search in each run was limited to 60 s. The experiment required at least 142,650/(60 $\times$ 24) = 99.0625 days of computational time. It was conducted on a Windows 10 Professional (Seattle, WA, USA) $\times$64 system having an Intel® Core i7 CPU with 3.40 GHz and 16 GB of RAM.

5.3. Experimental Analysis

The coverage achieved was evaluated based on the branch criterion, number of test cases (#T), and mutation score. All the experimental results were analyzed via nonparametric Mann–Whitney U tests with a significance level (p-value) of 0.05, the Vargha–Delaney $\hat{A}_{12}$ effect size, and a 95% confidence interval for the branch coverage achieved. Boxplots and marginal distribution plots were created using RStudio Version 1.1.383.

6. Experimental Results

The experimental results for EMSGA and the competing algorithms are presented and analyzed in this section. The experimental results are tabulated in Table 4, which shows the standard deviation (σ), a 95% confidence interval (CI) of the branch coverage, the p-value for the Mann–Whitney U tests, and the Vargha-Delaney $\hat{A}_{12}$ effect size.

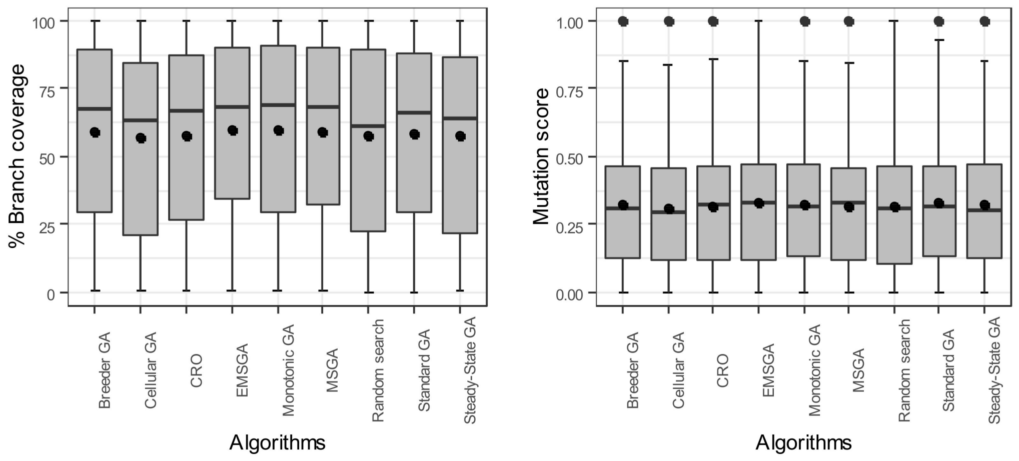
Table 4. Results of test case generation using each algorithm.

Algorithm	Branch Coverage			Mut. Score			#T	p-Value	$\hat{A}_{12}$ (EMSGA:Others)
	Avg.	σ	CI	Avg.	σ	CI			
EMSGA	0.5900	0.0032	(0.5877, 0.5923)	0.4174	0.0038	(0.4146, 0.4201)	180.49351	-	-
MSGA	0.5846	0.0033	(0.5823, 0.5870)	0.4166	0.0043	(0.4135, 0.4196)	181.5325	0.00578	0.87
GA	0.5829	0.0040	(0.5801, 0.5858)	0.4159	0.0046	(0.4127, 0.4192)	177.8818	0.00168	0.92
Monotonic GA	0.5855	0.0063	(0.5810, 0.5901)	0.4162	0.0050	(0.4127, 0.4198)	182.3091	0.03752	0.74
Steady-State GA	0.5699	0.0036	(0.5673, 0.5725)	0.4168	0.0023	(0.4152, 0.4185)	178.7455	0.00018	1
Breeder GA	0.5821	0.0059	(0.5779, 0.5864)	0.4167	0.0040	(0.4138, 0.4195)	180.0545	0.00466	0.88
Cellular GA	0.5588	0.0034	(0.5563, 0.5612)	0.4056	0.0044	(0.4024, 0.4087)	174.2039	0.00018	1
CRO	0.5717	0.0040	(0.5688, 0.5746)	0.4120	0.0062	(0.4076, 0.4164)	177.9416	0.00018	1
Random search	0.5683	0.0036	(0.5657, 0.5709)	0.4127	0.0025	(0.4109, 0.4144)	179.5857	0.00018	1

EMSGA achieved the highest branch coverage (0.5900). This means test cases of EMSGA can execute 59% of the source code of the class test. The branch coverage of EMSGA obtained that similar to the monotonic GA. However, EMSGA generated fewer test cases than the monotonic GA due to the limited search budget. Each algorithm had 60 s to search for the optimal test cases for each class. Although EMSGA generated fewer cases, the branch coverage of EMSGA was higher. This means that EMSGA is more efficient than monotonic GA. In terms of the mutation score, EMSGA achieved the best performance. The mutation score represents the number of faults that can be detected in the test cases, which is a measure of the quality of the test cases generated by each algorithm [47]. The $\hat{A}_{12}$ measure is a comparison of effect size between the EMSGA and the others; if $\hat{A}_{12} > 0.5$, it means EMSGA can beat that algorithm more than 50% of the time. For example, $\hat{A}_{12} = 0.74$ means EMSGA can beat the monotonic GA 74% of the time. The values of this metric for all the algorithms were found to be greater than 0.5. This means that the EMSGA can generate higher-quality test cases than the other algorithms.

Considering the values of all the metrics, EMSGA clearly outperformed MSGA in most categories. Furthermore, specifically in terms of the $\hat{A}_{12}$ measure, EMSGA performed significantly better than MSGA (average $\hat{A}_{12}$ effect size was 0.93). In the Mann–Whitney U tests, EMSGA exhibited a p-value of less than 0.05. From a comparison between EMSGA and MSGA, it can be concluded that EMSGA possesses a more effective best chromosome selection process due to the addition of genetic operators and is hence more efficient than the traditional MSGA. The higher mutation score implies that EMSGA is better at detecting faults than the other algorithms.

The distributions of the average branch coverage and average mutation scores obtained from the 1585 classes during the execution of the test cases generated by each algorithm are shown in Figure 4. The length of the box indicates the distribution of values between the 25% and 75% quantiles. The horizontal line in the box represents the median value. The dot in the box represents the mean value. The vertical lines indicate the smallest and largest values outside the middle 50%. The dots outside the box denote the outlier values. Despite the similar distributions of coverage and mutation score for all the algorithms, outliers of mutation score were observed across all the algorithms (see Figure 4b) except EMSGA and random search. This suggests that EMSGA and random search can detect up to 100% of the faults, while the other algorithms can detect approximately 80–90% of the faults (the outliers represent the undetected faults). Considering the distribution of coverage (see Figure 4a), EMSGA exhibited a higher average coverage than random search. Furthermore, EMSGA presented a narrower distribution, that is, less scattered data.

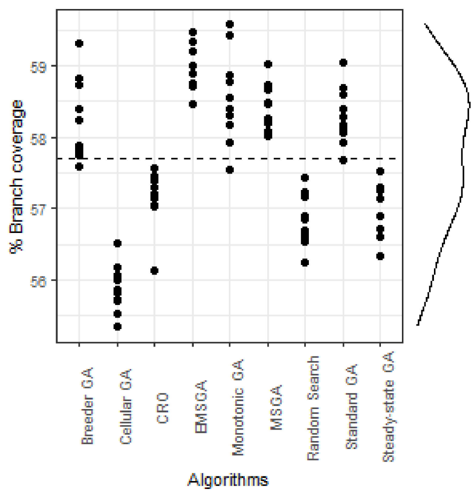


(**a**) Branch coverage criterion (**b**) Mutation score

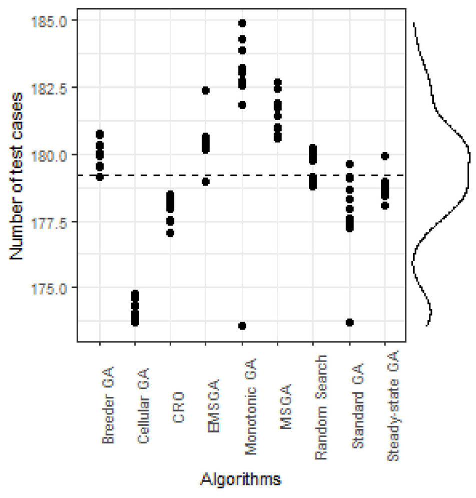
Figure 4. Coverage and mutation scores achieved by each algorithm.

Figure 5 presents the distributions of the branch coverage, number of test cases, and mutation score achieved by each algorithm. Each marginal distribution displays the average of each metric (dashed line) and the marginal density. The marginal density is the solid line on the right side of each marginal distribution plot that indicates the distribution of results. The average branch coverage of all the algorithms was 57.71% (Figure 5a). Five algorithms achieved values exceeding the average, namely EMSGA, MSGA, standard GA, monotonic GA, and breeder GA. In terms of the number of test cases (Figure 5b) as well, four algorithms achieved values better than the average (179.19 test cases), namely EMSGA, MSGA, monotonic GA, breeder GA, and random search. All algorithms exhibited mutation scores above the average (0.41). Thus, EMSGA achieved values exceeding the average for all three evaluation metrics. The ratio of classes reached branch coverage within each 10% branch coverage interval, as shown in Figure 6. For example, 35% of all classes that were tested in the test cases generated by EMSGA achieved a branch coverage between 81% and

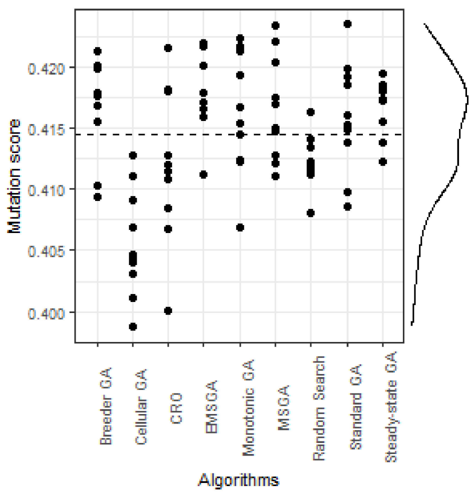
100%. From the experimental results, it is evident that EMSGA is feasible and effective for generating test cases.



(**a**) Branch coverage criterion



(**b**) Number of test cases



(**c**) Mutation score

Figure 5. Average values of metrics for each algorithm.

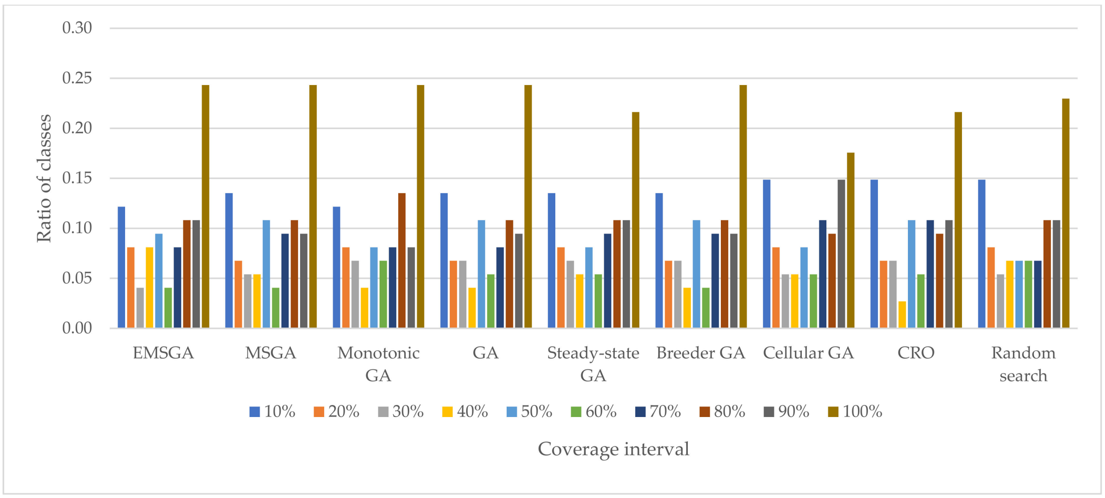
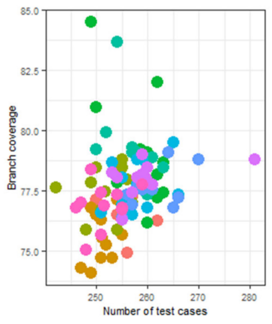
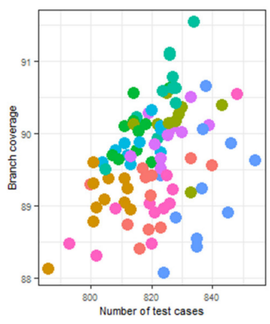


Figure 6. Proportion of classes for different branch coverage intervals.

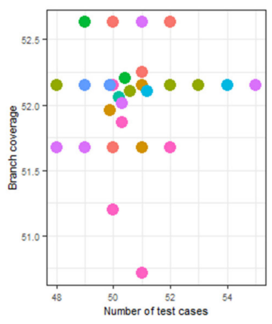
Figure 7 displays the association between the number of test cases and the achieved branch coverage when problem instances were executed using test cases of each algorithm. Several problem instances indicated the EMSGA achieved greater or equal branch coverage while the number of test cases was less than the others. The problem instance Truth is a small-scale program, and the test cases of all algorithms executed a similar number of source code.



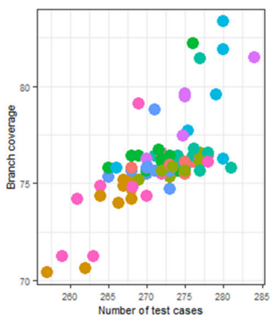
(**a**) Commons CLI



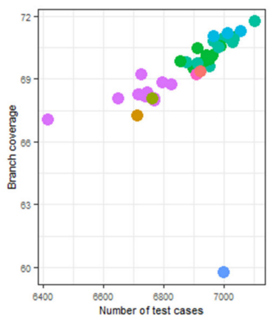
(**b**) Commons Codec



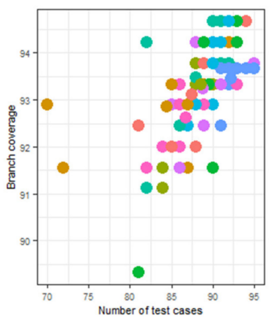
(**c**) Commons Email



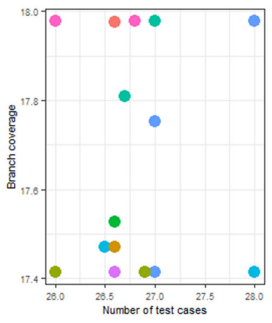
(**d**) Commons Jelly



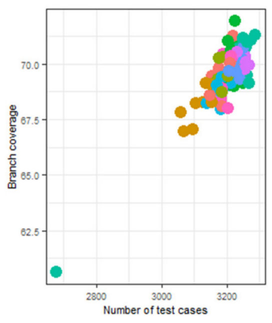
(**e**) Commons Math3



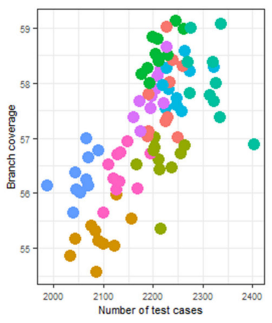
(**f**) Commons Numbers



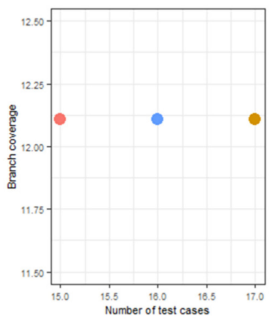
(**g**) Java Certificate Transparency



(**h**) Joda-Time



(**i**) SF110



(**j**) Truth

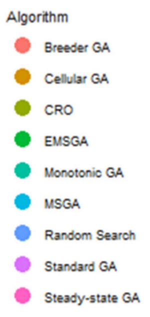


Figure 7. Problem instances that were evaluated with each algorithm.

7. Threats to Validity

Based on the results obtained, threats to internal validity are related to factors affecting the behavior of the software under test [48]. One such factor observed in the experiment was the number of test cases generated by all algorithms. Single testing might be inadequate for summarizing the performance of the algorithms in terms of generating test cases. In this experiment, each algorithm was run 10 times with the same tools. Furthermore, all parameters were defined with the same default values.

Threats to external validity are related to the generalization of the results beyond the scope of experimental analysis [22]. The SF110 corpus and nine open-source Java projects developed by Google and the Apache Software Foundation were utilized as case studies, which required a large number of experiments to be conducted. In this study, a total of 1585 classes were used, which included 203 classes from the SF110 corpus chosen based on previous studies [37] and all classes of the nine open-source Java projects. The reported results are limited to the search-based techniques employed in the experiments.

8. Discussion

EMSGA modifies the MSGA processes by comparing the parent and offspring and choosing the better chromosomes for the next generation. The selection of the better chromosome as input to the next generation allows for approaching the optimal solution. Our experimental results are in accordance with the results of previous experiments, which indicates that the branch coverage increases when a better chromosome is selected. For example, the monotonic GA achieved better results than the traditional GA [15,22]. Our results show that EMSGA can achieve a higher branch coverage, generate more test cases, and obtain a higher mutation score than MSGA.

One of the contributions of this research is our examination of the efficiency of EMSGA by extending it to EvoSuite, which is an automatic tool for generating test cases. The results of this application provide the number of test cases, the percentage of coverage, and mutation score. The results also indicate that EMSGA achieves a similar coverage with fewer test cases compared with monotonic GA. This is probably because the population of EMSGA contains two types of chromosomes, namely conservative and explorer chromosomes. The explorer chromosomes are created from high-fitness chromosomes. The main objective of software testing is to minimize the number of test cases and increase the coverage. The number of test cases affects the software development cost [5,49]. Although EMSGA produces fewer test cases than monotonic GA does in 60 s, the former achieves a higher coverage for the same number of test cases. A comparison of the efficiency between the existing algorithms in EvoSuite and EMSGA suggests that, in test case generation, the branch coverage may not be enough to clearly demonstrate the difference between results. The finding is consistent with Campos et al. (2018) [21], who indicated that the efficiencies of algorithms in EvoSuite may provide little difference in results for generating test cases. This could be due to a limitation on setting parameters, such as population size, basic function, or timing. In particular, as Fraser and Arcuri (2015) [50] pointed out, achieving a certain percentage of branch coverage and mutation score for a limited time may lead to higher mutation scores, but the coverage may be lower. The above experimental results also show that we can obtain higher mutation scores while having coverage very close to other algorithms. These findings lead us to believe that EMSGA has the potential to generate more test cases within a limited time and increase its coverage. Arcuri and Fraser [47] reported that the performance of a search-based technique depends on the parameter settings. A possible alternative is to find the best value of the parameters suitable for generating test cases [22], although the default values of EvoSuite are sufficient for evaluating algorithms in terms of test case generation. Therefore, appropriate values for EMSGA should be determined to generate the maximum number of test cases. Furthermore, EMSGA should be examined for other test coverage criteria.

9. Conclusions

This paper proposes an enhanced MSGA (EMSGA) to generate test cases for software testing. In EMSGA, the selection process involves creating two types of chromosomes to obtain better chromosomes before performing crossover and mutation operations. The performance of EMSGA on the basis of branch coverage, number of test cases, and mutation score was compared with that of other algorithms available in EvoSuite. The results show that EMSGA is more efficient than MSGA as well as the other algorithms. In addition, EMSGA can detect more faults in programs than the other algorithms. Therefore, because of its superior performance, EMSGA is expected to enable seamless automation of software testing, thereby facilitating the development of different software packages in the future.

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, 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 on 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, Taiwan; Department of Management Information Systems, Taiwan; National Pingtung University of Science and Technology, Taiwan; and the Ministry of Science and Technology, 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. Khan, R.; Amjad, M.; Srivastava, A.K. Optimization of Automatic Generated Test Cases for Path Testing Using Genetic Algorithm. In Proceedings of the 2nd International Conference on Computational Intelligence & Communication Technology, Ghaziabad, India, 12–13 February 2016; pp. 32–36. [CrossRef]
2. 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]
3. Aleti, A.; Grunske, L. Test data generation with a Kalman filter-based adaptive genetic algorithm. *J. Syst. Softw.* **2015**, *103*, 343–352. [CrossRef]
4. 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]
5. Kumar, D.; Mishra, M.M. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Comput. Sci.* **2016**, *79*, 8–15. [CrossRef]
6. Tsai, C.F.; Tsai, C.W.; Wu, H.C. A novel algorithm for multimedia multicast routing in a large scale network. *J. Syst. Softw.* **2004**, *72*, 431–441. [CrossRef]
7. 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]
8. Selim, M.; Siddik, M.S.; Gias, A.U.; Abdullah-Al-Wadud, M.; Khaled, S.M. A Ge-netic Algorithm for Software Design Migration fromStructured to Object Oriented Paradigm. In Proceedings of the 8th International Conference on Computer Engineering and Application (CEA 2014), Tenerife, Spain, 10–12 January 2014; pp. 187–192.
9. Murillo-Morera, J.; Quesada-López, C.; Castro-Herrera, C.; Jenkins, M. A genetic algorithm based framework for software effort prediction. *J. Softw. Eng. Res. Dev.* **2017**, *5*, 4. [CrossRef]
10. Bennett, T.E.; Brown, M.S.; Pelosi, M. A Genetic Algorithm for the Generation of Software Maintenance Release Plans without Human Bias. *J. Softw. Eng. Practice* **2015**, *1*, 6–21.

11. Khari, M.; Kumar, P.; Shrivastava, G. Enhanced approach for test suite optimisation using genetic algorithm. *Int. J. Comput. Aided Eng. Technol.* **2019**, *11*, 653–668. [CrossRef]
12. Mohi-Aldeen, S.M.; Mohamad, R.; Deris, S. Optimal path test data generation based on hybrid negative selection algorithm and genetic algorithm. *PLoS ONE* **2020**, *15*, e0242812. [CrossRef]
13. Rathee, A.; Chhabra, J.K. A multi-objective search based approach to identify reusable software components. *J. Comput. Lang.* **2019**, *52*, 26–43. [CrossRef]
14. Liu, Y.; An, K.; Tilevich, E. RT-Trust: Automated refactoring for different trusted execution environments under real-time constraints. *J. Comput. Lang.* **2020**, *56*, 100939. [CrossRef]
15. Shahabi, M.M.D.; Badiei, S.P.; Beheshtian, S.E.; Akbari, R.; Moosavi, M.R. EVOTLBO: A TLBO based Method for Automatic Test Data Generation in EvoSuite. *Int. J. Adv. Comput. Sci. Appl.* **2017**, *8*, 214–226. [CrossRef]
16. Fraser, G.; Arcuri, A.; McMinn, P. A Memetic Algorithm for whole test suite generation. *J. Syst. Softw.* **2015**, *103*, 311–327. [CrossRef]
17. Panichella, A.; Kifetew, F.M.; Tonella, P. A large scale empirical comparison of state-of-the-art search-based test case generators. *Inf. Softw. Technol.* **2018**, *104*, 236–256. [CrossRef]
18. Fraser, G.; Arcuri, A. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodo* **2014**, *24*, 1–42. [CrossRef]
19. Wang, R.; Sato, Y.; Liu, S. Mutated Specification-Based Test Data Generation with a Genetic Algorithm. *Mathematics* **2021**, *9*, 331. [CrossRef]
20. Rani, S.; Suri, B.; Goyal, R. On the Effectiveness of Using Elitist Genetic Algorithm in Mutation Testing. *Symmetry* **2019**, *11*, 1145. [CrossRef]
21. Tian, T.; Gong, D.; Kuo, F.C.; Liu, H. Genetic algorithm based test data generation for MPI parallel programs with blocking communication. *J. Syst. Softw.* **2019**, *155*, 130–144. [CrossRef]
22. Campos, J.; Ge, Y.; Albunian, N.; Fraser, G.; Eler, M.; Arcuri, A. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Inf. Softw. Technol.* **2018**, *104*, 207–235. [CrossRef]
23. Fraser, G.; Zeller, A. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Trans. Softw. Eng.* **2012**, *38*, 278–292. [CrossRef]
24. Fraser, G.; Arcuri, A. Whole test suite generation. *IEEE Softw. Eng.* **2012**, *39*, 276–291. [CrossRef]
25. Aleti, A.; Moser, I.; Grunske, L. Analysing the fitness landscape of search-based software testing problems. *Autom. Softw. Eng.* **2017**, *24*, 603–621. [CrossRef]
26. Whitley, D. Next Generation Genetic Algorithms: A User's Guide and Tutorial. In *Handbook of Metaheuristics*, 3rd ed.; Gendreau, M., Potvin, J.Y., Eds.; Springer: Cham, Switzerland, 2019; Volume 272, pp. 245–274. [CrossRef]
27. Sundar, S. A Steady-State Genetic Algorithm for the Dominating Tree Problem. In Proceedings of the 10th International Conference on Simulated Evolution and Learning, Dunedin, New Zealand, 15–18 December 2014; pp. 48–57. [CrossRef]
28. Agapie, A.; Wright, A.H. Theoretical analysis of steady state genetic algorithms. *Appl. Math.* **2014**, *59*, 509–525. [CrossRef]
29. Muhlenbein, H.; Schlierkamp-Voosen, D. Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization. *Evol. Comput.* **1996**, *1*, 25–49. [CrossRef]
30. Mühlenbein, H.; Schlierkamp-Voosen, D. The Science of Breeding and Its Application to the Breeder Genetic Algorithm. *Evol. Comput.* **1994**, *1*, 335–360. [CrossRef]
31. Dorronsoro, B.; Alba, E. A Simple Cellular Genetic Algorithm for Continuous Optimization. In Proceedings of the IEEE International Conference on Evolutionary Computation, Vancouver, BC, Canada, 16–21 July 2006; pp. 2838–2844. [CrossRef]
32. Pedemonte, M.; Panizo-LLedot, A.; Bello-Orgaz, G.; Camacho, D. Exploring Multi-objective Cellular Genetic Algorithms in Community Detection Problems. In *Intelligent Data Engineering and Automated Learning*; Analide, C., Novais, P., Camacho, D., Yin, H., Eds.; Springer: Cham, Switzerland, 2020; Volume 12490, pp. 223–235. [CrossRef]
33. Whitley, D. A genetic algorithm tutorial. *Stat. Comput.* **1994**, *4*, 65–85. [CrossRef]
34. Lam, A.Y.S.; Li, V.O.K. Chemical Reaction Optimization: A tutorial. *Memetic. Comp.* **2012**, *4*, 3–17. [CrossRef]
35. Marrison, C.I.; Stengel, R.F. The use of random search and genetic algorithms to optimize stochastic robustness functions. In Proceedings of the 1994 American Control Conference, Baltimore, MD, USA, 29 June–1 July 1994; pp. 1484–1489. [CrossRef]
36. Zabinsky, Z.B. Random search algorithms. In *Wiley Encyclopedia of Operations Research and Management Science*; Cochran, J.J., Cox, L.A., Keskinocak, P., Kharoufeh, J.P., Smith, J.C., Eds.; John Wiley & Sons: New York, NY, USA, 2010; pp. 1–13. [CrossRef]
37. Rojas, J.M.; Fraser, G.; Arcuri, A. Seeding strategies in search-based unit test generation. *Softw. Test. Verif. Reliab.* **2016**, *26*, 366–401. [CrossRef]
38. Panichella, A.; Kifetew, F.M.; Tonella, P. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Softw. Eng.* **2016**, *44*, 122–158. [CrossRef]
39. Grano, G.; Palomba, F.; Nucci, D.D.; Lucia, A.D.; Gall, H.C. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *J. Syst. Softw.* **2019**, *156*, 312–327. [CrossRef]
40. Ma, P.; Cheng, H.; Zhang, J.; Xuan, J. Can This Fault Be Detected: A Study on Fault Detection via Automated Test Generation. *J. Syst. Softw.* **2020**, *170*, 110769. [CrossRef]
41. Fraser, G. A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator. In Proceedings of the 10th International Symposium, Montpellier, France, 8–9 September 2018; pp. 106–130. [CrossRef]

42. Hansen, N.; Auger, A.; Finck, S.; Ros, R. Real-Parameter Black-Box Optimization Benchmarking 2010: Experimental Setup. 2010. Available online: https://hal.inria.fr/inria-00462481 (accessed on 31 May 2021).
43. 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]
44. Črepinšek, M.; Liu, S.; Mernik, M. Replication and comparison of computational experiments in applied evolutionary computing: Common pitfalls and guidelines to avoid them. *Appl. Soft. Comput.* **2014**, *19*, 161–170. [CrossRef]
45. 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*, 1–12. [CrossRef] [PubMed]
46. Deb, K.; Deb, D. Analysing mutation schemes for real-parameter genetic algorithms. *Int. J. Artif. Intell. Soft Comput.* **2014**, *4*, 1–28. [CrossRef]
47. Jia, Y.; Merayo, M.; Harman, M. Introduction to the special issue on Mutation Testing. *Softw. Test. Verif. Reliab.* **2015**, *25*, 461–463. [CrossRef]
48. Luo, Q.; Moran, K.; Poshyvanyk, D.; Penta, M.D. Assessing Test Case Prioritization on Real Faults and Mutants. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Madrid, Spain, 23–29 September 2018; pp. 240–251. [CrossRef]
49. Ammann, P.; Offutt, J. *Introduction to Software Testing*, 2nd ed.; Cambridge University Press: New York, NY, USA, 2016; pp. 18–19.
50. Fraser, G.; Arcuri, A. Achieving scalable mutation-based generation of whole test suites. *Empir. Softw. Eng.* **2015**, *20*, 783–812. [CrossRef]