symmetry

MDPI

Article

On the Effectiveness of Using Elitist Genetic Algorithm in Mutation Testing

Shweta Rani, Bharti Suri and Rinkaj Goyal *

University School of Information, Communication and Technology, Guru Gobind Singh (GGS) Indraprastha University, New Delhi 110078, India

* Correspondence: rinkajgoyal@gmail.com

Received: 8 July 2019; Accepted: 5 September 2019; Published: 9 September 2019

Abstract: Manual test case generation is an exhaustive and time-consuming process. However, automated test data generation may reduce the efforts and assist in creating an adequate test suite embracing predefined goals. The quality of a test suite depends on its fault-finding behavior. Mutants have been widely accepted for simulating the artificial faults that behave similarly to realistic ones for test data generation. In prior studies, the use of search-based techniques has been extensively reported to enhance the quality of test suites. Symmetry, however, can have a detrimental impact on the dynamics of a search-based algorithm, whose performance strongly depends on breaking the "symmetry" of search space by the evolving population. This study implements an elitist Genetic Algorithm (GA) with an improved fitness function to expose maximum faults while also minimizing the cost of testing by generating less complex and asymmetric test cases. It uses the selective mutation strategy to create low-cost artificial faults that result in a lesser number of redundant and equivalent mutants. For evolution, reproduction operator selection is repeatedly guided by the traces of test execution and mutant detection that decides whether to diversify or intensify the previous population of test cases. An iterative elimination of redundant test cases further minimizes the size of the test suite. This study uses 14 Java programs of significant sizes to validate the efficacy of the proposed approach in comparison to Initial Random tests and a widely used evolutionary framework in academia, namely Evosuite. Empirically, our approach is found to be more stable with significant improvement in the test case efficiency of the optimized test suite.

Keywords: mutation testing; search-based software engineering; genetic algorithm; test data generation

1. Introduction

Test data generation is a critical, labor-intensive, and time-consuming process that significantly affects software quality. However, automation can minimize the effort and may produce effective test cases satisfying specific objectives. Owing to a combinatorial problem, which is computationally intractable, different search-based algorithms have been proposed and used for generating the test suite [1–5]. These algorithms include Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Ant Colony Optimization (ACO), among others. Initially conceived by Holland [6], GA [7,8] is frequently adapted by the researchers and provides an evolved test suite through iterative searching of the search space. In each iteration, the fitness of the test suite is measured, and for convergence, it must satisfy some test requirements, i.e., branch coverage, statement coverage, and path coverage. Mutation testing is a type of structural software testing that inserts the fault in the source code of a program and makes it faulty. These faults exhibit the mistakes a programmer can make while writing the program, and the faulty version is known as a mutant. Test data is required to reveal a fault in the program. Here, test data are the inputs to the program and executing them against a mutant

indicates whether the fault is exposed or not. Fault exposition is also known as mutation coverage. Prior studies [9,10] suggest that mutation coverage can be considered to generate a superior test suite than other coverage measures and better guides the selection mechanism of test cases for evolution.

Mutants are widely accepted and simulated artificial faults that behave similarly to realistic ones [11,12] for test data generation [9,13]. These are created by systematic injection of faults using predefined mutation operators [14]. Mutation testing was initially suggested by DeMillo [15] and later explored by different researchers [16–18]. Execution of a test case (test inputs) against these faults results in the adequacy score of that test case. This score is also known as mutation score (%), which is measured using the killed mutants (covered faults, *KM*) and the total non-equivalent mutants (*M*) and is expressed as $[|KM| \times 100/|M|]$. However, some of the mutants are not recognized by any of the test cases because they do not differ from the original program. Therefore, such mutants are referred to as equivalent and adversely impact the test suite performance [19,20]. The problem of identification of equivalent mutants has also been addressed in prior studies [21–23]. Apart from the several benefits including a reduction in the search space and providing the framework for test suite quality assessment, mutation testing also suffers from a high computational cost. However, in the last few decades, researchers have tried to minimize this cost by using various techniques which usually follow do-fewer, do-smarter, or do-faster kinds of strategies [17,18,24–33] and more studies can be traced in a recent survey [34]. Selective mutation, which is used in the current study, is one of these approaches that generate mutants by applying some operators from the large set of mutation operators [14] (Section 2.1).

Search-based techniques with mutation testing have also been extensively researched and can be traced in the relevant surveys [35–39]. According to these studies, GA is more preferred by the research community. Literature in the test data generation field is reviewed and summarized in Table 1 in chronological order. It contains the records of publication details, tool availability, type of mutants created, search technique implemented for test generation, population size, and fitness function.

Initially, Baudry et al. [40,41] reported the natural killing of 60% mutants using the first set of component test cases. However, to detect more hard-to-kill mutants (90% mutants), they suggested to use GA to iteratively evolve the test cases. The same fitness function (mutation score) was used by other researchers as well to find a mutation adequate test suite in [42–45]. Application of GA with mutation testing has also been applied in finite state machines (FSM) by Molinero et al. [46] and Nilsson et al. [47].

Later, a new idea for mutant identification and formulating fitness function was suggested by Bottaci [48] and implemented by [49,50]. Following the related idea of fitness evaluation, Fraser [9,10] employed GA with weak mutation testing in a tool known as Evosuite. It automatically generates test cases using assertions. The performance of this tool is further demonstrated and compared in other prominent studies [51–55].

Considering the object's state, Bashir and Nadeem [56] proposed a novel fitness function, which restricts the search process by reviewing the tests that have either obtained the desired state or requires more method calls. The authors in [57–59] further extended the work that resulted in the development of a tool named eMuJava. They also compared the relative performances of their proposed variation of GA with traditional GA using ten Java programs of total 1028 LOC. In their study, improved GA converged in 373 iterations and created 9325 test cases that detected 93.5% mutants for triangle program. C++ mutation operators were also used by Perez et al. [60–62] for mutant selection and test suite improvement. Higher-order mutants were also examined for test creation using GA by Ghiduk [63].

Table 1. Summary of related work in test data generation.

Authors	Year	Tool Available?	S	M	Approach	P	Fitness Function
Baudry [40,41]	2000	No	1	TM	GA	-	Mutation Score
Masud [49]	2005	No	-	TM	GA	-	Botacci Fitness function

Table 1. *Cont.*

Authors	Year	Tool Available?	S	M	Approach	P	Fitness Function
Molinero [46]	2009	No	-	-	EGA	50	Mutation Score
Mishra [50]	2010	No	-	TM	EGA	-	Botacci Fitness function
Fraser [9]	2012	Yes	10	TM	GA	100	Branch, Mutation Distance & Mutation Impact
Fraser [10]	2015	Yes	40	TM	GA	100	Branch, Mutation Distance & Mutation Impact
Subramanian [43]	2011	No	21	TM	GA	-	Mutation Score
Haga [44]	2012	No	1	TM	GA	-	Mutation Score
Louzada [42]	2012	No	3	TM	EGA	-	Mutation Score
Bashir [57]	2017	Yes	10	TM, CM	IGA	50	State-Based Fitness Function
Ghiduk [63]	2018	No	4	HOM	GA	10–20	Mutation Score
Delgado-Perez [60]	2018	No	8	-	GA	5% *	Mutation Quality Based Fitness Function
This study	-	-	14	Delete Mutation Operators	EGA	$10 \times I$	Test Case Effectiveness and its Complexity in terms of time-steps

Here, S: Number of Subject Programs, M: Type of Mutants, TM: Traditional Mutants, CM: Class Level Mutants, HOM: Higher-Order Mutants, P: Population Size, EGA: Elitist GA, IGA: Improved GA, 5%; *: population size is 5% of number of mutants.

The study in this paper expands our previous work [45,64], dealing with GA and mutation testing. However, this work presents an improved fitness evaluation, incorporation of elitism, and performance comparison with the existing techniques. It implements a variant of GA by effectively blending the benefits of mutation testing for non-redundant test suite generation, followed by a novel fitness function that considers test case complexity in terms of time-steps with high fault exposure. Test case complexity impacts the process of testing at a more considerable extent for finding the faults. In this exposition, the performance of the proposed approach is compared with a popular testing tool, i.e., Evosuite [10] as well as with Initial Random tests. The contributions of this study are:

- Implementation of GA using the idea of diversification and intensification along with the integration of elitism and mutation-based fitness function. It addresses the problem of costly test suite with fault revealing abilities.
- Comparison of the effectiveness, efficiency and cost of the proposed approach with the state-of-the-art techniques on 14 Java programs on different evaluation metrics.
- Analyzing the impact of other artificial faults on the effectiveness of generated test suite.

This paper is organized as follows: Section 2 presents the basic terminologies with the proposed approach and an illustration of its execution. Section 3 describes the experimental setup and information of mutants used and Section 4 discusses the results of the evaluation with some limitations. Section 5 concludes with the significant findings of this study.

2. Methods and Materials

2.1. Terminologies and Illustrations

(a) Software Testing: It is the process of executing a program with the intent of finding the faults [65]. Actual output and expected output of executing a test case are compared and if they differ then it is said that fault is present.

(b) Test Case: A test case is an input to the program with its expected output and is used for testing the functionality of the program [65]. A collection of test cases is called a test suite, e.g., for a single input problem, a test case can be $\{T_1 = 7\}$, while for two input problem, $\{T_1 = (8, 4)\}$.

(c) Mutation Testing: It is a method of software testing that seeds the faults or errors in the program with a precondition of the syntactical correctness of the altered program [16,18].

(d) Mutants: The faulty version of a program is known as mutants. A mutant with a single fault is characterized as a single order mutant while those with more than one fault are higher-order mutants.

(e) Mutation Operators: Mutants are generated using some metagenic rules [14,66] which seeds the fault in the program systematically. These metagenic rules are termed as mutation operators in mutation testing (Figure 1). Figure 2 illustrates a few examples of such mutants.

(f) Killing a Mutant: A test $t \in T$ (Test Suite) kills a mutant $m \in M$ (set of Mutants) if the execution of t can distinguish the behavior of the original program s and mutant program m. It can be expressed as: t kills $m := m(t) \neq s(t)$

(g) Mutation Score: A test t that kills KM mutants out of M mutants, for t, mutation score (%) is calculated as $MS[t] := |KM| \times 100/|M|$

(h) GA and its Operators: GA is an evolutionary algorithm based on the concept of natural genetics of reproduction [6–8]. In an iteration of execution, it starts with the random initial population P, fitness evaluation of P, selection, reproduction (crossover and mutation) and stops re-iterating when an optimal solution is found (Figure 3). Each individual in the population is represented as chromosome (a sequence of genes) and encoded in binary for a binary-encoded GA, which is used in this study. For the evolution of individuals, crossover combines two individuals and produces two new individuals (offspring); on the other hand, mutation flips a bit in the gene of a chromosome [67]. In this work, a population of GA is mapped to the set of test cases, and the chromosome is mapped to the concatenated value of test inputs.

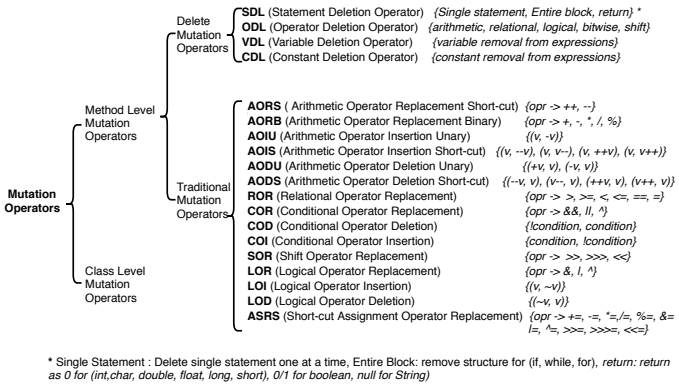

Figure 1. Details of mutation operators (adapted from [23]).

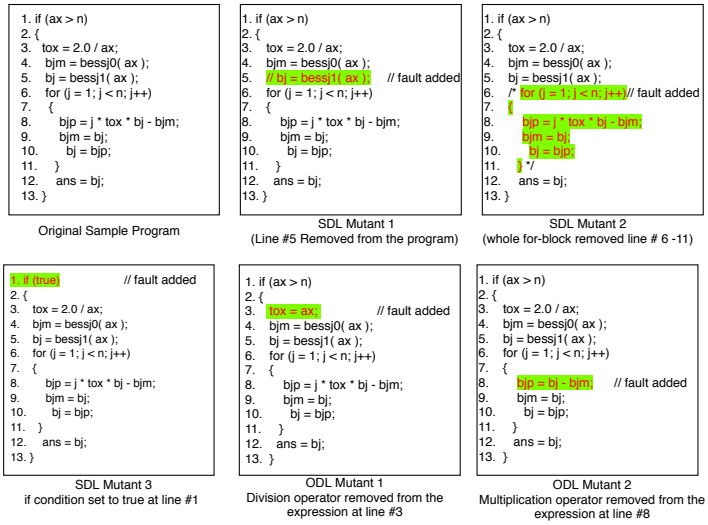

Figure 2. Illustration of some mutants on a sample program.

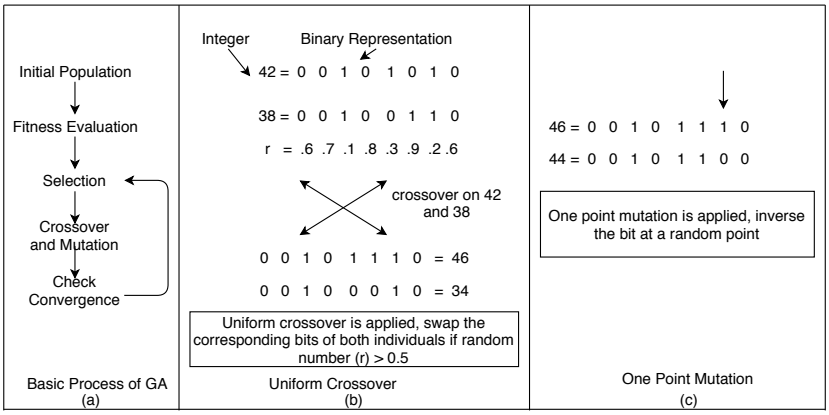

Figure 3. Basics of GA (**a**) basic process, (**b**) uniform crossover, and (**c**) mutation.

2.2. Description of Proposed Approach

The flow chart in Figure 4 illustrates the functionality of our proposed approach *tdgen_gamt* that begins with reading the source code of the original program and outputs: list of methods in the original program, the number of input variables for method under test for random initialization of the population. Here, the population refers to a collection of test cases that are forwarded for fitness evaluation over the artificial mutants (the faulty version of a subject program), and the fault matrix (Figure 5) also gets updated in each iteration. The approach then proceeds with the selection of parent test cases for reproduction, which in turn applies intensification and diversification, if not converged. We perform intensification (crossover) when there is a chance of improvement in test case locally, otherwise, perform diversification in the form of mutation that intends to diversify the solution globally. At the end of each iteration, if *tdgen_gamt* converges, it stops functioning and provides the non-redundant test suite with mutation coverage information.

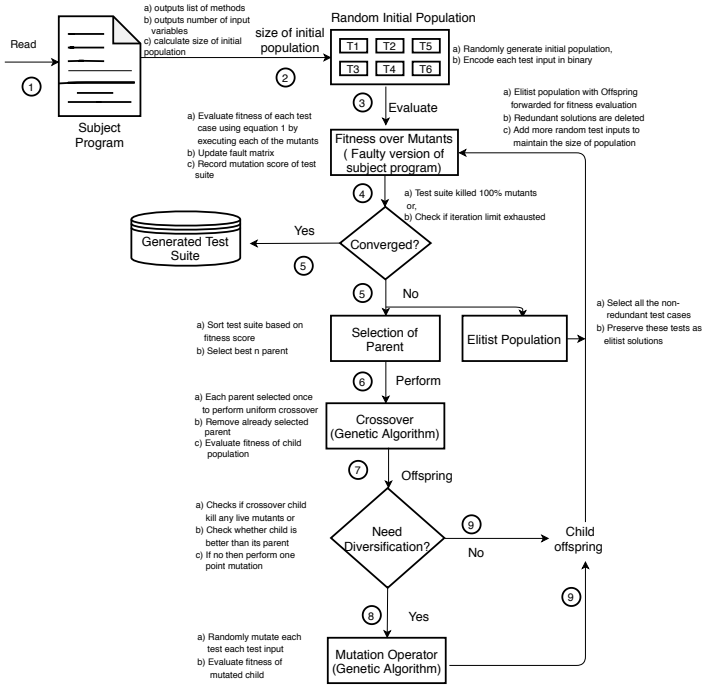

Figure 4. Procedural flow of the proposed approach *tdgen_gamt*.

Test Case ID	Fitness	Complexity	Mutants				
			M1	M2	M3	.....	Mn
T1	45.343	30	1	1	0		0
T2	18.911	90	0	0	0		1
T3	18.933	30	0	0	0		1
T4	30.790	11	0	1	1		0

Figure 5. Structure of fault matrix used in *tdgen_gamt*.

As shown in Algorithm 1, our approach *tdgen_gamt* creates a random solution of test inputs, i.e., pop, which is initially empty. In this paper, each test input can have its value in the range $[-10, 110]$. Here, we use a binary-coded Genetic Algorithm and perform crossover and mutation on the binary string. Therefore, the integer test inputs are converted in binary using ($8\times$ number of inputs) bits for reproduction (8 bits are sufficient to represent each test input in the range $[-10, 110]$). There are variants for chromosomes encoding in GA e.g., gray, binary and, real, and each has its own advantages and disadvantages [68–70]. Binary encoding is beneficial to include a sudden change in the population of solutions, which is desirable in the current study to diversify the population for increasing the chances of detecting live mutants. We evaluate the quality of the test suite by executing it against the mutants (each test case is executed over each mutant in the set) (Section 2.2.1), and the fitness of each test case is recorded in the fault matrix (Figure 5). For example, we have n number of mutants ($M_1 - M_n$) and 4 test cases uniquely identified by test case IDs ($T_1 - T_4$). Each test case has its own fitness, complexity, and mutant detection information in the form of 0 and 1 which in turn express live and killed mutant, respectively.

Algorithm 1: The Proposed Approach *tdgen_gamt*

Input : Subject program under test S, Initial population size p_size, Iteration limit itr_limit, Selection criteria $sel\_parent\%$, Generated non-equivalent mutants M
Output: non-redundant test data set, T

1 $T \leftarrow \emptyset$
2 $pop \leftarrow \emptyset$
/* Value of num_inputs is read from S for method under test and pop is randomly initialized within range [−10, 110] using InitializePopulation(p_size). InitializePopulation(p_size) generates the random population using Math.random() */
3 $pop \leftarrow$ InitializePopulation(p_size)
4 $MS \leftarrow 0.0$
/* measure the mutation score MS of current Population pop using function FitnessEvaluation(pop, M), update fault matrix simultaneously */
5 $MS \leftarrow$ FitnessEvaluation(pop, M)
6 $iteration \leftarrow 0$
7 **while** $iteration < itr\_limit$ *or* $MS < 100$ **do**
  /* size of the population is maintained throughout the process by random generation of some new test cases. InitializePopulation() is used for random generation of new solutions. */
8 $pop \leftarrow pop \cup$ InitializePopulation($p\_size - pop.size()$)
9 $MS \leftarrow$ FitnessEvaluation(pop, M)
10 $parent \leftarrow$ select bestfit $sel\_parent\%$ testcases from pop for evolution
  /* each solution in $parent$ is selected once to perform crossover, for n parent, only n new solutions are generated using Crossover($parent$) and only non-redundant solutions are kept in $Offspring$ */
11 $offspring \leftarrow$ Crossover($parent$)
12 **if** *diversification required?* **then**
    /* perform mutation using Mutation($parent$) in case when no crossover offspring could kill any live mutants. After mutation, only non-redundant solutions are kept in $Offspring$ */
13    $Offspring \leftarrow$ Mutation($parent$)
  /* elitism is applied to preserve the non-redundant previous population pop and $Offspring$ are merged in pop */
14 $pop \leftarrow pop \cup Offspring$
  /* evaluate fitness for new population, remove redundant tests and update fault matrix */
15 $MS \leftarrow$ FitnessEvaluation(pop, M)
16 $iteration \leftarrow iteration + 1$
17 $T \leftarrow pop$
18 **return** T

2.2.1. Fitness Evaluation

In this study, we aim to generate test cases with maximum effectiveness (measured in terms of mutation score, which is the fault-finding capability of a test case) along with minimum test case complexity (TCC). Therefore, the inverse of TCC appears as part of fitness evaluation (Equation (1)). Furthermore, there is no correlation between test case effectiveness and TCC since effectiveness depends only on the value of the test case that is used for program execution and mutant detection. Algorithm 2 explains the calculation of fitness function and mutation score.

$$\text{Fitness Function} = TCE \ + \ (1 \ / \ TCC \ ) \tag{1}$$

Algorithm 2: FitnessEvaluation (T, M)

Input : Set of test cases for fitness evaluation T, non-equivalent mutants M
Output: Mutation score MS

```
/* MS is the test suite mutation score, MS[t] is the mutation score of a test case t
   which is test case effectiveness and expressed as TCE in Equation (1)            */
MS ← 0.0
n ← T.size()
x ← M.size()
allkilledMutants ← ∅
/* Method is the method under test which is identified initially after reading the
   source program S                                                                  */
foreach t ∈ (T1....Tn) do
    /* for elitist test cases, MS[t] and Fitness[t] are already evaluated in previous
       iteration, no need to re-evaluate it                                         */
    if Fitness[t] is null then
        /* killedMutants[t] is a set of killed mutants by test case t               */
        killedMutants[t] ← ∅
        /* TCC (Test Case Complexity) is measured by invoking the method under test
           using Java reflection which makes use of an in-built library invoke()    */
        TCC[t] ← Method.invoke(S, t)
        foreach m ∈ (M1....Mx) do
            if Method.invoke(m, t) ≠ Method.invoke(S, t) then
                killedMutants[t] ← killedMutants[t] ∪ m;
        MS[t] ← killedMutants[t].size() × 100/M.size()
        Fitness[t] ← MS[t] + 1/TCC[t]
    else
        /* for elitist test cases, killedMutants[t] is already evaluated in previous
           iteration                                                                */
    /* allkilledMutants is a set of killed mutants by all test cases in T          */
    allkilledMutants ← allkilledMutants ∪ killedMutants[t];
/* Mutation score, MS for whole test suite T is calculated                         */
MS ← allkilledMutants.size() × 100/M.size()
return MS
```

Here, TCE (Henceforth, TCE refers to Test Case Effectiveness and measured using mutation score (MS)) is measured using the fault-finding capability of a test case, i.e., mutation score that has been frequently used in the literature. TCC (TCC refers to Test Case Complexity in terms of time-steps) is the test case complexity in terms of time-steps measured in microseconds using an in-built library of Java (java.lang.reflect.Method). Here, TCC is not source code complexity or cyclomatic complexity. The latter is used for path coverage while we are generating the tests for mutant coverage. We assume

that a complex test case might take more time for execution than the less complex one. Two test cases may detect the same faults and have the same mutation score but definitely, differ in complexity (e.g., a test case with value 100 will run "for-loop" 100 times and take more steps for execution than another test case with value 1 or less than 100. In this case, a test case with lower execution time-steps is selected first to be kept in the fault matrix if both detect the same faults). The designed fitness function intends to select better tests with minimum cost. At any time that fitness is evaluated, redundant tests are also removed, and consequently, fault matrix gets updated.

A redundant test case detects the faults previously identified by another test. Such test cases do not contribute to testing and only increase the cost [71]. Let us take an example to understand the concept. Assume there are two test cases T_1 and T_2. T_1 identifies faults M_1 and M_2. If test case T_2 only detects fault M_2. Then, we say that execution of T_2 is not required because both faults can be killed by executing only T_1; therefore, test case T_2 is redundant and can be deleted from the test suite without losing the effectiveness of the test suite. Removal of such tests leads to an efficient test suite. The pseudocode in Algorithm 3 illustrates how redundant tests are identified and removed.

Algorithm 3: RemoveRedundantTests (T)

Input : Set of test cases with its fitness information T
Output: Set of non-redundant test cases T

```
/* T is sorted in ascending order of fitness score using Collections.sort() function */
1  T ← Collections.sort(T);
2  foreach t ∈ (T1....Tn) do
3      AllKilledMutants ← ∅
4      set flag ← False
5      foreach p ∈ (T1....Tn) do
6          if !t.getkilledMutants().containsAll(p.getkilledMutants()) &
            !p.getkilledMutants().containsAll(t.getkilledMutants()) then
7              AllKilledMutants ← t.getkilledMutants()
8          else if p.getkilledMutants().containsAll(t.getkilledMutants()) & p.getkilledMutants().size() >
            t.getkilledMutants().size() then
9              AllKilledMutants ← p.getkilledMutants()
10         if AllKilledMutants.size()>0 & (AllKilledMutants.containsAll(t.getkilledMutants()) ||
            t.getkilledMutants().containsAll(AllKilledMutants)) then
11             set flag ← True
12             break;
13     if flag then
14         remove t from T;
15 return T
```

2.2.2. Diversification vs. Intensification for Reproduction

While generating solutions, search-based algorithms (evolutionary algorithms) perform two operations, i.e., intensification and diversification [72–74]. In intensification, it searches the neighborhood search space and exploits the solution by selecting the best of these local solutions. Diversification, however, explores the search space globally and tries to diverse the solution. In this study, during every successive iteration, the current population of tests evolve based on the fact that it might be improved in the local optimum (intensification) or needs diversification globally. In GA, intensification favors the current population and perform crossover to find the better offspring in terms of fitness [72]. Two chromosomes exchange their properties at a random position and create two new offspring. In this study, we perform uniform crossover (Algorithm 1) on the parent population with 0.5 random probability (this type of crossover is recommended for the chromosomes with moderate

or no linkage among its genes [67], which suits to this study). We also ensure that each pair of test case participate in this phenomenon only once. Therefore, n parent test cases generate n new offspring and thus reduces the time and space complexity. Each offspring is then evaluated for fitness using Equation (1). We then check for the offspring, able to kill some live mutants or better than its parent population. These crossover test cases are merged with the previous population and the process is repeated till convergence. However, if crossover test cases fail to kill some live mutants or is not better than its parent, then, diversification in the form of one-point mutation (Algorithm 1) is preferred to increase the probability of detection of live faults as well as reduces the risk of identifying an already killed fault. Mutation is applied to all crossover test cases. Here, a single bit is flipped from 0 to 1 or vice versa at a random position between 0 to the length of the gene in a chromosome. The intention behind this strategy of intensification and diversification is only to improve the effectiveness of the test suite iteratively. We present an example (Figure 6) for ease in understanding the idea (a detailed example is given in Section 2.3).

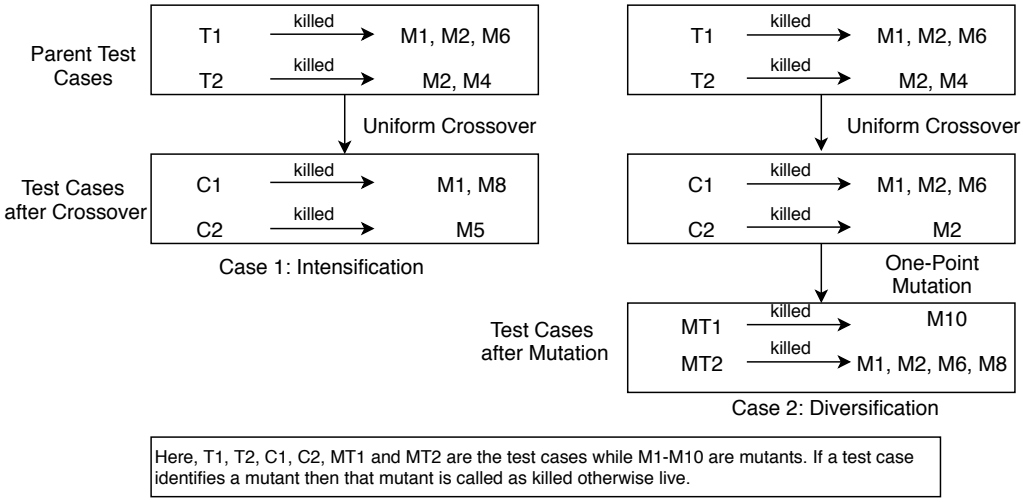

Figure 6. Illustration of intensification and diversification.

Let we have two test cases T_1, T_2 with their killed mutants (M_1, M_2, M_6) and (M_2, M_4) respectively. Consider case 1, parent test cases T_1, T_2 are improved via intensification (crossover); however, offspring from crossover (C_1, C_2) is not more effective than parent test case, but C_1, C_2 kill live mutant M_8 and M_5 respectively. It makes C_1 and C_2 valuable in the entire population. Meanwhile in case 2, C_1 and C_2 do not enhance the effectiveness of the complete test suite, therefore C_1, C_2 are diversified using mutation. This may produce effective test cases.

2.2.3. Population Replacement Strategy and Elitism

In general GA, all the individuals of a current population are removed and new individuals for the new population are derived using reproduction over the current population. By doing so, it may lose the best individuals due to its stochastic nature. Therefore, some best solutions are retained as elitist solutions and guarantees that the quality of the solutions will be improved iteratively [67,75,76]. We use the benefits of elitism to sustain all non-redundant individuals of the previous population which can be 10%, 20% or 50% of the entire population depending on fault-finding behavior of the test cases during execution (Algorithm 1). Usually, GA works on the principle of human reproduction. In this, the older and less fit solutions get dead in each iteration and some of the fitted solutions are kept as the elitist solution. With time, these solutions lose their fitness and are replaced with the new ones. However, in testing, a test case will have the same fitness throughout the process. Test case fitness is evaluated on all the mutants and during the process, no new mutant is added. Thus, if we get a test case that is good in finding the fault, we can preserve it and can cut the cost of re-generating a similar test case which is already created in the previous iteration. Fitness evaluation for such test cases is

also not required in successive iteration, we can use the preserved fitness. That is why we save all the non-redundant and valuable test cases as the elitist solution in each iteration regardless of their fitness (best or worst). A test case in an adequate test suite is considered to be effective and relevant if it succeeds in killing a resistant or hard mutant irrespective of its fitness [77]. This elitism strategy may minimize the cost of finding the optimum solutions in a fewer number of iterations.

2.2.4. Convergence

In general, search-based algorithms stop functioning at convergence based on some criteria such as reaching a time limit, iteration, or coverage. The proposed approach *tdgen_gamt* converges when all the non-equivalent mutants are killed by the test suite, i.e., achieving 100% mutation score. However, it may be possible that some mutants are not killed even after a significant number of iterations; such mutants are too hard to be identified and might be killed only by specific tests, e.g., in the case of equilateral triangle (all inputs must be equal). To avoid such a situation, another criterion is also defined, i.e., iteration limit. The approach stops functioning when any of the criteria satisfy (Algorithm 1).

2.3. A Detailed Example

To show how the approach works for test generation, we take an example for generating tests for a single input problem (Table 2). Let the population size is 8 and the number of non-equivalent mutants is 10 ($M_1 - M_{10}$). Initially, eight test cases ($T_1 - T_8$) are randomly created and executed against all the mutants ($M_1 - M_{10}$). For each test case ($T_1 - T_8$), fitness, and mutation score are evaluated. In iteration 1, test T_1 detects 5 mutants out of 10, therefore its mutation score is evaluated as 50. We also checked the status of each test case as redundant (R) or non-redundant (N). After fitness evaluation, best tests (T_1, T_7) are selected to perform crossover (intensification) and generate two new offspring i.e., C_1, C_2. After their fitness evaluation, it is found that test case C_1 is redundant but C_2 is non-redundant. In this case, we say that intensification is worthy and adds one new test case in the population. At the end of the iteration, non-redundant crossover test cases are merged with the previous non-redundant solution and total five test cases are obtained (T_1, T_5, T_6, T_7, C_2). We then check for convergence, mutation score of the complete test suite is 90% that is <100. We then again re-iterate the complete process. At the beginning of each iteration, the size of the population is maintained, and 3 more random test cases are added in iteration 2. We then perform crossover on T_1 and T_7 which lead to C_3, C_4. Here, it is found that both crossover test cases are redundant and could not kill any live mutant. Here, we say that intensification could not produce valuable test cases. Therefore, we try to diverse the crossover population i.e., C_3, C_4 using mutation with the possibility to obtain the desired test cases. This leads to MT_1, MT_2 and only MT_1 is found to be non-redundant. The leftover live mutant M_{10} is killed by this new offspring MT_1. Now all the non-redundant new offspring and previous population are kept together and it is found that all the mutants are now successfully detected by these test cases (T_1, T_5, T_7, C_2, MT_1). At convergence, our approach *tdgen_gamt* stops and returns the non-redundant test suite.

Table 2. A detailed example of how *tdgen_gamt* works.

Initial Population Evaluation						
Initial Population	***#M***	***#KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
T_1 (24)	10	5	M_1, M_2, M_3, M_7, M_9	50	50.125	N
T_2 (40)	10	3	M_1, M_2, M_9	30	30.23	R
T_3 (59)	10	2	M_2, M_9	20	20.145	R
T_4 (15)	10	2	M_9, M_6	20	20.52	R
T_5 (81)	10	1	M_4	10	10.23	N
T_6 (103)	10	1	M_5	10	10.45	N
T_7 (33)	10	4	M_6, M_7, M_2, M_1	40	40.245	N
T_8 (112)	10	1	M_5	10	10.100	R
Mutation Coverage for Initial Population: 80%, Live Mutants: M_8, M_{10}						

Table 2. *Cont.*

Begin Iteration 1: Perform Crossover (Intensification)						
Selected Cases	**Crossover Offspring**	**#*KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
T_1 (24)	C_1 (25)	5	M_1, M_2, M_3, M_7, M_9	50	50.234	R
T_7 (33)	C_2 (32)	1	M_8	10	10.5	N
At the end of Iteration 1: Non-Redundant Test suite (Crossover and Elitist Test Cases)						
Initial Population	**#*M***	**#*KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
T_1 (24)	10	5	M_1, M_2, M_3, M_7, M_9	50	50.125	N
T_5 (81)	10	1	M_4	10	10.23	N
T_6 (103)	10	1	M_5	10	10.45	N
T_7 (33)	10	4	M_6, M_7, M_2, M_1	40	40.245	N
C_2 (32)	10	1	M_8	10	10.5	N
Mutation Coverage for current Population: 90%, Live Mutants: M_{10}						
Begin Iteration 2: Add 3 more test cases to maintain the size of population						
Initial Population	**#*M***	**#*KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
T_1 (24)	10	5	M_1, M_2, M_3, M_7, M_9	50	50.125	N
T_5 (81)	10	1	M_4	10	10.23	N
T_6 (103)	10	1	M_5	10	10.45	N
T_7 (33)	10	4	M_6, M_7, M_2, M_1	40	40.245	N
C_2 (32)	10	1	M_8	10	10.5	N
T_9 *(56)*	*10*	*2*	M_2, M_9	*20*	*20.45*	*R*
T_{10} *(78)*	*10*	*1*	M_8	*10*	*10.20*	*R*
T_{11} *(87)*	*10*	*1*	M_4	*10*	*10.30*	*R*
Perform Crossover (Intensification)						
Selected Cases	**Crossover Offspring**	**#*KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
T_1 (24)	C_3 (40)	3	M_1, M_2, M_9	30	30.23	R
T_7 (33)	C_4 (17)	2	M_6, M_9	20	20.55	R
Perform Mutation (Diversification)						
Crossover Cases	**Mutation Offspring**	**#*KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
C_3 (40)	MT_1 (44)	2	M_5, M_{10}	20	20.43	N
C_4 (17)	MT_2 (21)	1	M_8	10	10.8	R
At the end of Iteration 2: Non-Redundant Test suite (Mutation and Elitist Test Cases)						
Initial Population	**#*M***	**#*KM***	**Mutants Killed**	**Mutation Score**	**Fitness**	**Status**
T_1 (24)	10	5	M_1, M_2, M_3, M_7, M_9	50	50.125	N
T_5 (81)	10	1	M_4	10	10.23	N
T_7 (33)	10	4	M_6, M_7, M_2, M_1	40	40.245	N
C_2 (32)	10	1	M_8	10	10.5	N
MT_1 (44)	10	2	M_5, M_{10}	20	20.43	N
Mutation Coverage for current Population: 100%, No Live Mutants						

Here, #*M*: Number of Non-equivalent Mutants, #*KM*: Number of Killed Mutants, R: Redundant, N: Non-redundant.

3. Experimental Setup

This section explains the experimental settings of different scenarios presented in this study. First, it presents the subject programs (Section 3.1), along with how mutants are generated (Section 3.2). Then, we discuss the evaluation metrics used to measure the efficacy of our proposed approach.

3.1. Subject Programs under Test

We conduct the empirical experiment on 14 Java programs used widely in mutation testing and test data generation [10,18,35,36]. The number of inputs varies between 1–6 in the selected programs. S2 and S5 have the minimum number of inputs, i.e., one. Furthermore, S8 and S9 programs have the largest set of inputs, i.e., five, six, respectively. The LOC of the selected programs ranges between 19–153 and their specifications are listed in Table 3. In this study, test cases are only generated for the considered method which is the main calling method in the corresponding subject program.

Table 3. Subject programs with their methods under consideration.

S	Programs	#Methods	Considered Method	#Inputs	Description
S1	Bessj [78]	3	bessj	2	artificial numeric case study
S2	EvenOdd [4]	1	checkEvenOdd	1	checks a number if even or odd
S3	GCD* [79]	1	gcd	2	finds the greatest common divisor
S4	Power [80]	1	power	2	calculates the power of a number
S5	Primes [81]	2	nextPrime	1	returns the next prime number
S6	Quadratic [65]	1	get_roots	3	quadratic equation solver
S7	Remainder [4]	1	getRemainder	2	returns remainder of two numbers
S8	SmallestLargest [82]	3	getMinMax	5	finds smallest and largest number
S9	Stats [80]	1	computeStats	6	returns statistics of given numbers
S10	StudentDivison [65]	1	cal_division	3	returns the division of a student
S11	TrashAndTakeOut [80]	2	trash	1	performs some calculations
S12	Trityp [80]	1	Triang	3	returns the type of triangle
S13	Tritype2 [83]	1	Triang	3	returns the type of triangle
S14	Triangle [65]	1	find_tri_class	3	returns the type of triangle

Note: #Methods—Total number of methods, Considered Methods- methods considered in this study; however, follows Caller–Callee relationship, #Inputs- number of inputs in considered method, GCD*—Greatest Common Divisor.

3.2. Mutants Used

To check the adequacy of the test data, artificial faults are created using *MuJava* [84] instead of real faults. The procedure of mutant generation is illustrated in Figure 7. MuJava creates method-wise mutants (methods in the subject program) and test cases are generated for the driver method (main calling method) in this study. Inspired by the reduced cost of selective mutation, only few mutation operators (SDL-Statement Deletion, ODL—Operator Deletion, CDL—Constant Deletion, and VDL—Variable Deletion) are incorporated for mutant generation (Figure 1) because all operators are not cost-effective and produce a higher number of equivalent mutants and redundant mutants [20]. According to Untch [66], these mutants are powerful and generates fewer equivalent mutants and require highly effective tests for fault detection [85,86]. It is also noticed that ODL mutants are a super-set of VDL and CDL mutants. Therefore, only SDL and ODL mutants are kept for further use. All the mutants, including SDL and ODL are again analyzed for equivalence detection using 1000 random test cases, and live mutants are further analyzed manually for equivalent mutant detection.

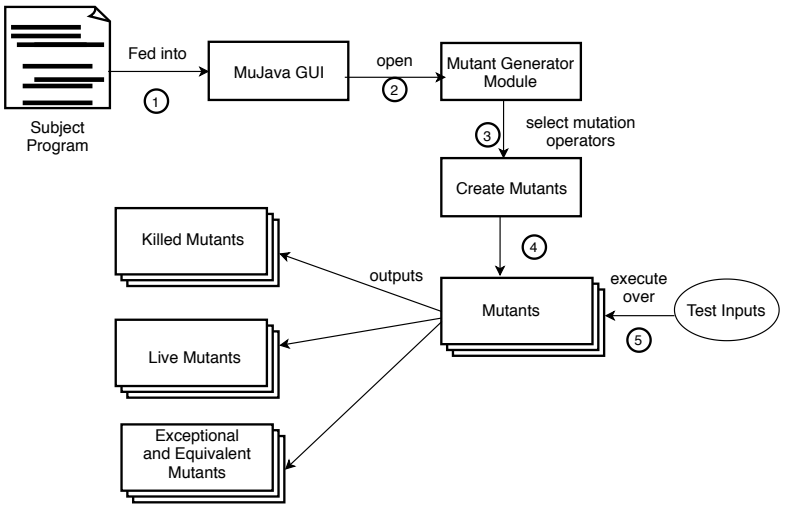

Figure 7. Step by step procedure for creating the mutants.

In this study, we created two types of artificial faults for method under test of subject programs. The first set contains the mutants (SDL, ODL) which are used for test case creation while another set (traditional mutants) is used for evaluating the results and for comparison with the state-of-the-art techniques (Figure 1). Traditional mutants [14] are generated and found to be 80% more than the set 1

mutants. The executable mutants for the considered method under test of each subject program are listed in Table 4. Total mutants generated by each of the SDL, ODL and traditional operators are 283, 385, and 3452 respectively; out of these, 5%, 2%, and 7% mutants are found to be equivalent for each operator respectively (Table 5). Some of the mutants are not executable, throw an exception, and stick in infinite loops. Such mutants are deleted before test generation. Non-executable mutants are 3%, 1%, 6% for SDL, ODL and traditional mutants respectively (Table 5).

Table 4. Description of executable mutants for method under test.

S	LOC	ODL%	SDL%	Traditional%
S1	153	100	93	85
S2	19	100	100	91
S3	20	93	80	88
S4	43	100	83	85
S5	54	58	76	55
S6	42	100	100	86
S7	19	100	86	78
S8	62	100	90	90
S9	30	98	85	82
S10	36	86	100	93
S11	37	100	100	77
S12	53	98	97	89
S13	73	100	100	95
S14	44	100	96	95
Total % for S1–S14		97%	92%	87%

Table 5. Description of equivalent and exceptional mutants for method under test.

S	Equivalent Mutants%			Exceptional Mutants%		
	ODL	SDL	Traditional	ODL	SDL	Traditional
S1	0	7	11	0	0	4
S2	0	0	4	0	0	4
S3	0	0	3	7	20	9
S4	0	0	5	0	17	10
S5	8	17	16	33	7	29
S6	0	0	10	0	0	4
S7	0	0	9	0	14	13
S8	0	10	9	0	0	1
S9	2	8	5	0	8	12
S10	14	0	5	0	0	2
S11	0	0	14	0	0	9
S12	3	3	10	0	0	1
S13	0	0	4	0	0	2
S14	0	0	4	0	4	1
Total % for S1–S14	2%	5%	7%	1%	3%	6%

3.3. Evaluation Metrics

To compare the effectiveness of our approach with Evosuite and Initial Random tests, different evaluation metrics are considered and listed in GQM template (Table 6) [27,87]. Answers to the questions are discussed in Section 4.

Table 6. GQM oriented research goals.

Parameters	Description
Goal	To investigate how GA with higher mutation coverage perform to automatically generate the low-cost test suite
Question	Q1 Is our approach more effective and efficient than Evosuite. How much our approach can improve the Initial Random tests at finding the faults? Q2 Is the approach stable in killing above 90% mutants? Q3 How different parameters of the approach impacts the performance?
Metric	Mutation Score, Number of Hard Mutants, Test Suite Size, Test Case Efficiency
Object of Study	*tdgen_gamt*, Evosuite and Initial Random tests
Purpose	Measure the effectiveness and efficiency of *tdgen_gamt* and evaluate the improvement in performance over Evosuite and Initial Random tests. To evaluate how much efficient tests are generated.
Focus	Generating the low-cost test data, covering the maximum number of faults
Perspective	Researcher point of view
Context	First Order SDL, ODL and Traditional Method Level Mutants

3.4. Experiments

We apply *tdgen_gamt* to generate and select only non-redundant test inputs and repeat this 50 times to alleviate the consequences of random variation. To quantify the efficacy of our approach, various statistics measures are recorded, including mutation score, test suite size, and test case efficiency. Experiments are carried out on a 32-bit system with core i7 processor and 4GB RAM. To automate the approach *tdgen_gamt*, Eclipse IDE for Java developer Mars version is used with JDK8.

4. Results and Discussion

This section discusses the results of each research goal listed in Table 6. The performance of our approach is also evaluated and discussed in the succeeding Section 4.1. To evaluate the efficacy of the proposed approach, it is compared with Evosuite and Initial Random tests in terms of various aspects i.e., test suite effectiveness, test suite size, and test case efficiency. For a fair comparison, Evosuite test cases are executed only for the method under test.

4.1. Performance of the Proposed Approach tdgen_gamt

Our proposed approach creates the optimized test suite using elitist GA (Algorithm 1). We collected all the results related to achieved mutation coverage, test generation time (in seconds), number of iterations required, final test suite size and the total number of fitness evaluations performed till convergence and detailed in Table 7. For each of the problem, our approach takes $[0.1, 10.2]$ s on average for creating a test suite by successfully detecting [88–100%] mutants. Fitness evaluations are also recorded to show that only [24–574] test cases are evaluated till convergence. On average, only 10 test cases out of 238 tests are found to be non-redundant and valuable. This shows that our approach generates low-cost test suite by repetitive deletion of obsolete tests for the method under test of each subject program. *tdgen_gamt* takes advantage of intensification and diversification for finding the solution in fewer number of iterations as well as time. This seems to be useful in the area of search-based optimization by balancing between the control parameters.

Table 7. Performance evaluation of *tdgen_gamt*.

S	Coverage %	TGT* (Seconds)	Iterations	TSS*	Fitness Evaluation
S1	100.00	2.1	6	7	92
S2	100.00	0.1	3	4	24
S3	99.91	0.8	10	6	168
S4	99.90	0.7	9	6	146
S5	98.62	0.7	22	8	105
S6	98.00	5.5	28	9	633
S7	99.60	0.7	10	6	166
S8	100.00	1.2	3	16	107
S9	100.00	7.9	2	13	117
S10	100.00	0.5	2	11	60
S11	99.06	0.4	13	5	99
S12	88.11	10.2	30	16	532
S13	96.85	5.8	30	13	574
S14	94.24	9.3	30	15	513
Average	98.16%	3.3	14	10	238

* TGT: Test Generation Time, TSS: Test Suite Size.

4.2. Effectiveness Comparison with Evosuite and Initial Random Tests

This section evaluates whether the proposed approach with intensification and diversification can guide the search process to obtain the desired outcome. Evosuite is a state-based evolutionary test suite generation tool implemented by Fraser and Arcuri [10], where tests are created using weak mutation coverage criteria. There is no provision to find out the information about the number of mutants generated and mutation score achieved by the resultant test suite. On the other hand, our approach *tdgen_gamt* reports above-listed measures along with test suite size and fitness evaluations. Both approaches also differ in fitness evaluation and mutation operator selection. Evosuite calculates the fitness using the distance to calling function, distance to mutation and mutation impact, while in *tdgen_gamt*, it is computed using test case effectiveness and its complexity in terms of time-steps. Evosuite internally generated mutants for eight mutation operators [10] (Table 8) while we create mutants only for delete operators (SDL, ODL).

Table 8. List of mutation operators implemented in Evosuite.

S. No	Mutation Operator
1.	Delete Field
2.	Delete Call
3.	Insert Unary Operator
4.	Replace Arithmetic Operator
5.	Replace Bitwise Operator
6.	Replace Comparison Operator
7.	Replace Constant
8.	Replace Variable

The results by taking only the average mutation score into consideration show that Evosuite tests (With greater test size than our proposed mechanism) outperform the other two by 9% (our approach) and 17% (Initial Random Tests) referring all traditional mutants. Evosuite needs on average 12 test cases to identify 96.35% traditional mutants while our approach requires only 10 test cases to kill 87.83% traditional mutants. As stated in [88], a technique is recognized more effective when it generates tests, killing more faults using an equal size test suite. Considering the size of the test suite along with mutation score, we also analyze the effectiveness of each approach and named it as test case efficiency

(*TCEF*) (Equation (2)). The discussion related to *TCEF* is given in Section 4.3. Overall, out of 2993 executable traditional mutants, Evosuite could detect 2884 mutants, *tdgen_gamt* could kill 2629 mutants while Initial Random tests killed only 2383 mutants. For some of the programs (S8, S9, S10), Initial tests could detect more traditional mutants than *tdgen_gamt*. For such cases, removing redundant tests found to be harmful. As a test case is redundant for a mutant set but may be non-redundant for some other mutant set. Therefore, Initial tests perform better for traditional mutants for S8–S10. On the other hand, *tdgen_gamt* could achieve high fault exposure when considering SDL, ODL faults.

We run each of the approaches 50 times and record average and median mutation score achieved against traditional as well as SDL, ODL mutants (Table 9). Standard deviation is also measured to show the variability among data from the mean. For each of the program, the standard deviation is found to be minimum in case of *tdgen_gamt*, which in turn suggests that our approach performs similarly and produces stable results when executed for any number of times. Meanwhile, for Evosuite, test suite deviates from the mean in the range [0, 9]. For our approach, average and median are found to be equal for most of the programs that further demonstrate that it produces the test suite in a symmetric fashion.

Table 9. Effectiveness of test suites.

S	Coverage Over SDL-ODL Mutants (%)			Coverage Over Traditional Mutants (%)		
	tdgen_gamt	**Evosuite**	**Initial**	**tdgen_gamt**	**Evosuite**	**Initial**
S1	100.00 ± 0.000 (100.00)	96.33 ± 2.530 (96.47)	96.69 ± 0.019 (95.29)	95.99 ± 0.005 (96.00)	98.53 ± 0.086 (98.56)	95.05 ± 0.011 (95.41)
S2	100.00 ± 0.000 (100.00)	100.00 ± 0.000 (100.00)	90.00 ± 0.094 (91.66)	90.65 ± 0.007 (90.69)	95.35 ± 0.000 (95.35)	86.47 ± 0.045 (86.04)
S3	99.91 ± 0.006 (100.00)	85.91 ± 6.818 (86.36)	88.09 ± 0.047 (90.90)	92.92 ± 0.018 (93.17)	99.88 ± 0.485 (100.00)	91.08 ± 0.026 (91.36)
S4	99.90 ± 0.007 (100.00)	94.40 ± 3.955 (95.00)	89.60 ± 0.043 (90.00)	96.53 ± 0.007 (97.00)	100.00 ± 0.000 (100.00)	90.98 ± 0.025 (91.18)
S5	98.62 ± 0.020 (100.00)	94.14 ± 7.042 (96.55)	62.14 ± 0.162 (62.00)	88.72 ± 0.023 (88.88)	97.46 ± 0.383 (97.53)	66.64 ± 0.095 (66.00)
S6	98.00 ± 0.009 (97.56)	85.85 ± 4.927 (87.80)	83.76 ± 0.047 (82.90)	83.34 ± 0.057 (80.23)	98.10 ± 1.664 (97.67)	73.33 ± 0.055 (72.10)
S7	99.60 ± 0.016 (100.00)	88.67 ± 8.459 (93.33)	86.00 ± 0.090 (86.66)	98.26 ± 0.009 (98.55)	99.83 ± 0.900 (100.00)	93.68 ± 0.038 (94.20)
S8	100.00 ± 0.000 (100.00)	61.09 ± 8.987 (60.87)	95.87 ± 0.025 (95.65)	92.42 ± 0.007 (92.70)	97.98 ± 3.092 (99.23)	93.07 ± 0.016 (93.48)
S9	100.00 ± 0.000 (100.00)	88.69 ± 2.532 (88.57)	98.59 ± 0.010 (99.00)	95.96 ± 0.006 (95.91)	99.23 ± 0.905 (99.55)	97.00 ± 0.006 (97.05)
S10	100.00 ± 0.000 (100.00)	91.62 ± 5.275 (91.89)	94.59 ± 0.043 (94.59)	69.14 ± 0.046 (67.89)	90.98 ± 2.962 (90.20)	77.95 ± 0.064 (76.90)
S11	99.06 ± 0.022 (100.00)	100.00 ± 0.000 (100.00)	73.76 ± 0.132 (70.58)	86.99 ± 0.039 (87.95)	91.30 ± 0.000 (91.30)	68.03 ± 0.092 (63.77)
S12	88.11 ± 0.049 (89.47)	62.82 ± 7.832 (64.47)	50.79 ± 0.080 (47.36)	79.55 ± 0.052 (80.70)	90.93 ± 9.436 (94.62)	45.67 ± 0.083 (44.88)
S13	96.85 ± 0.018 (97.87)	89.53 ± 6.938 (91.49)	83.15 ± 0.044 (82.97)	90.55 ± 0.022 (91.60)	98.94 ± 0.853 (99.30)	69.33 ± 0.106 (67.48)
S14	94.24 ± 0.010 (93.90)	65.10 ± 2.956 (65.24)	85.76 ± 0.056 (86.58)	68.62 ± 0.091 (65.07)	90.38 ± 4.858 (94.59)	66.22 ± 0.081 (66.36)

Note: Each first row displays average ± standard deviation while second row illustrates median value for 50 runs.

During experimentation, we calculate the mutant detection rate of each mutant and analyze which approach is successful in detecting stubborn mutants [89]. Results (Table 10) reveal that out of 2993 executable traditional mutants, approximately 2–3% artificial faults could never be killed by any of the approaches that further indicates our approach conclusively perform good at recognizing the traditional mutants. Meanwhile, in the case of SDL, ODL faults, our approach successfully detected approximately all the mutants out of 634 executable mutants.

Table 10. Subject program wise stubborn mutants for method under test.

S	SDL-ODL Mutants (%)			Traditional Mutants (%)		
	tdgen_gamt	**Evosuite**	**Initial**	**tdgen_gamt**	**Evosuite**	**Initial**
S1	0.0	0.0	0.0	2.6	1.4	1.4
S2	0.0	0.0	0.0	7.0	4.7	4.7
S3	0.0	0.0	0.0	4.3	0.0	0.0
S4	0.0	0.0	0.0	1.0	0.0	0.0
S5	0.0	0.0	0.0	4.0	2.5	2.5
S6	0.0	0.0	2.4	0.6	0.0	1.7
S7	0.0	0.0	0.0	1.4	0.0	0.0
S8	0.0	0.0	0.0	3.8	0.8	0.8
S9	0.0	1.0	0.0	2.3	0.5	0.0
S10	0.0	0.0	0.0	2.0	2.0	0.0
S11	0.0	0.0	0.0	1.4	8.7	1.4
S12	1.3	1.3	9.2	3.6	3.1	5.9
S13	0.0	0.0	2.1	0.7	0.7	2.4
S14	1.2	6.1	3.7	2.6	5.4	4.6
Total (%)	0.3%	1.1%	1.9%	2.5%	1.9%	2.1%

4.3. Efficiency and Cost Comparison with Evosuite and Initial Random Test

We evaluate the efficiency and cost by recording the statistics for test case efficiency (Equation (2)) ($TCEF$), and test suite size (TSS) (Table 11).

$$\text{Test Case Efficiency}(TCEF) = TSE \ / \ TSS \tag{2}$$

Table 11. Efficiency and cost measures.

S	Test Case Efficiency			Test Suite Size		
	tdgen_gamt	**Evosuite**	**Initial**	**tdgen_gamt**	**Evosuite**	**Initial**
S1	13.7 ± 0.24 13.7	8.9 ± 0.97 9.0	4.8 ± 0.05 4.8	7.0 ± 0.14 7.0	11.1 ± 1.12 11.0	20.0 ± 0.00 20.0
S2	22.7 ± 0.18 22.7	19.1 ± 0.00 19.1	8.6 ± 0.45 8.6	4.0 ± 0.00 4.0	5.0 ± 0.00 5.0	10.0 ± 0.00 10.0
S3	15.5 ± 0.47 15.6	13.1 ± 1.16 12.5	4.6 ± 0.13 4.6	6.0 ± 0.14 6.0	7.7 ± 0.71 8.0	20.0 ± 0.00 20.0
S4	16.2 ± 0.36 16.2	12.7 ± 0.83 12.5	4.5 ± 0.12 4.6	6.0 ± 0.14 6.0	7.9 ± 0.50 8.0	20.0 ± 0.00 20.0
S5	11.6 ± 0.72 11.3	9.7 ± 0.04 9.8	6.7 ± 0.94 6.6	7.7 ± 0.61 8.0	10.0 ± 0.00 10.0	10.0 ± 0.00 10.0
S6	9.1 ± 0.24 8.9	9.6 ± 0.36 9.8	2.4 ± 0.18 2.4	9.2 ± 0.39 9.0	10.2 ± 0.38 10.0	30.0 ± 0.00 30.0
S7	16.6 ± 0.73 16.4	14.4 ± 0.64 14.3	4.7 ± 0.19 4.7	5.9 ± 0.24 6.0	7.0 ± 0.28 7.0	20.0 ± 0.00 20.0
S8	5.9 ± 0.22 5.8	5.7 ± 0.34 5.7	1.9 ± 0.03 1.9	15.8 ± 0.63 16.0	17.2 ± 0.89 17.0	50.0 ± 0.00 50.0
S9	7.4 ± 0.04 7.4	6.1 ± 0.26 6.2	1.6 ± 0.01 1.6	13.0 ± 0.00 13.0	16.4 ± 0.72 16.0	60.0 ± 0.00 60.0

Table 11. *Cont.*

S	Test Case Efficiency			Test Suite Size		
	tdgen_gamt	**Evosuite**	**Initial**	**tdgen_gamt**	**Evosuite**	**Initial**
S10	6.3 ± 0.41 6.2	7.0 ± 0.42 6.9	2.6 ± 0.21 2.6	11.0 ± 0.00 11.0	13.1 ± 0.64 13.0	30.0 ± 0.00 30.0
S11	18.0 ± 0.96 17.7	18.3 ± 0.00 18.3	6.8 ± 0.91 6.4	4.8 ± 0.37 5.0	5.0 ± 0.00 5.0	10.0 ± 0.00 10.0
S12	5.1 ± 0.22 5.1	5.2 ± 0.65 5.3	1.5 ± 0.27 1.50	15.5 ± 1.01 16.0	17.5 ± 1.37 17.0	30.0 ± 0.00 30.0
S13	7.1 ± 0.11 7.1	5.2 ± 0.75 5.0	2.3 ± 0.35 2.2	12.7 ± 0.50 13.0	19.3 ± 2.33 20.0	30.0 ± 0.00 30.0
S14	4.7 ± 0.62 4.4	5.6 ± 0.45 5.6	2.2 ± 0.27 2.2	14.7 ± 0.46 15.0	16.3 ± 0.93 17.0	30.0 ± 0.00 30.0

Note: Each first row displays average ± standard deviation while second row illustrates median value for 50 runs.

Here, TSE (Henceforth, TSE refers to test suite effectiveness) is the effectiveness of complete test suite i.e., mutation score of the test suite. TSS (TSS refers to Test Suite Size) is the size of the test suite i.e., the number of test cases in the resultant test suite.

Figure 8 demonstrates the improvement in efficiency and reduction in test suite size for our proposed approach over Evosuite tests. We also analyze how much initial tests are genetically improved using *tdgen_gamt*. From the results (Table 11, Figure 8), it is found that *tdgen_gamt* generates 13%, 205% more efficient and 18%, 60% reduced test suite than Evosuite and Initial Random tests respectively. Removing non-redundant test cases greatly minimizes the resultant test suite size, which in turn increases $TCEF$ (Equation (2)). However, efficiency is not improved when compared with Evosuite and varies in the range $[-15, -2]$ for some of the subject programs. It might be the reason that Evosuite generates the test cases using some of the traditional mutants while our approach uses only SDL, ODL mutants for test creation. Therefore, we can say that the test suite from both approaches will definitely perform better for the mutants they use for fitness evaluation.

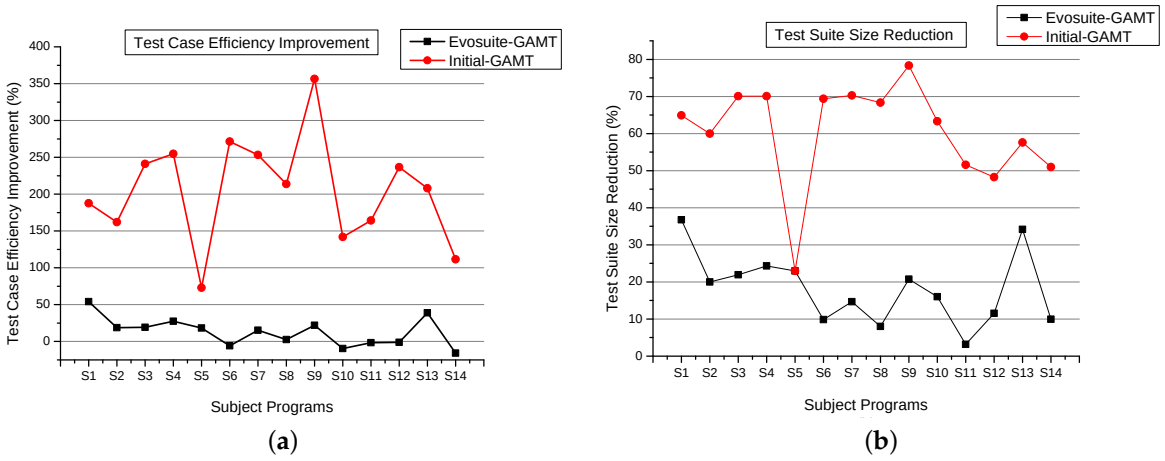

Figure 8. Efficiency and cost comparison (**a**) compared Evosuite to *tdgen_gamt*, (**b**) compared Initial random tests to *tdgen_gamt*.

4.4. Stability Analysis

Reproducibility and conclusion stability are two significant characteristics of any new proposed approach. To validate the stability of our approach, we repeat 50 experiments and collect the average results (Table 12). When considering full mutation coverage, our tests perform best for five subject programs; however, is found to be worst for S6, S12, S13, and S14 only. It might be a reason that these

programs require some specific test cases, e.g., to identify an equilateral triangle, all inputs must be equal. On average, it could not achieve the full coverage for [36, 100]% times for six out of 14 programs. For each program, test data always revealed more than 90% mutants except for subject program S12. For each of the 50 runs, the standard deviation is too minimum and very close to 0 (Table 9), therefore, it can be stated that it is stable. According to the results, it can be declared that our algorithm can generate the test suite with higher mutation coverage.

Table 12. Number of times failed to achieve 100%, above 95%, above 90% mutant coverage (results are recorded in percentage of failure times).

S	$= 100\%\ MS$	$\geq 95\% MS$	$\geq 90\%\ MS$
S1	0	0	0
S2	0	0	0
S3	2	0	0
S4	2	0	0
S5	36	4	0
S6	82	0	0
S7	6	6	0
S8	0	0	0
S9	0	0	0
S10	0	0	0
S11	16	16	0
S12	100	100	64
S13	98	20	0
S14	100	92	0

4.5. Selection of Control Parameters

The performance of GA primarily depends on its parameters, i.e., population size, number of iterations, reproduction operator probability and method, and convergence criteria. We investigate the impact of these parameters on different efficiency measures. The population size symbolizes the number of individuals present in each population. It is expressed using an array of bits, i.e., binary-coded values of each input. For example, in the case of Power program, each individual is encoded as an array of 8 bits, each byte representing the value of inputs. Our approach is replicated for different values of control parameters (Table 13) and repeated 50 times due to the stochastic nature of GA. Impact on several measures, i.e., test generation time and coverage are recorded and illustrated in Figure 9. For each of the 14 subject programs (method under test), we execute *tdgen_gamt* 50 times for 24 different parameter values ($24 \times 14 \times 50$ =16,800).

Table 13. Control parameters of *tdgen_gamt*.

Parameters	Value
Initial Population	5×Inputs, 10×Inputs, 15×Inputs, 20×Inputs
Fitness Function	$TCE + 1/TCC$
Parent Selection	Fittest 25%, 50%
Crossover	Uniform Crossover
Mutation	One-Point Mutation
Convergence Criteria	100% Mutation Score or Iteration Limit
Iteration Limit	30, 50, 100

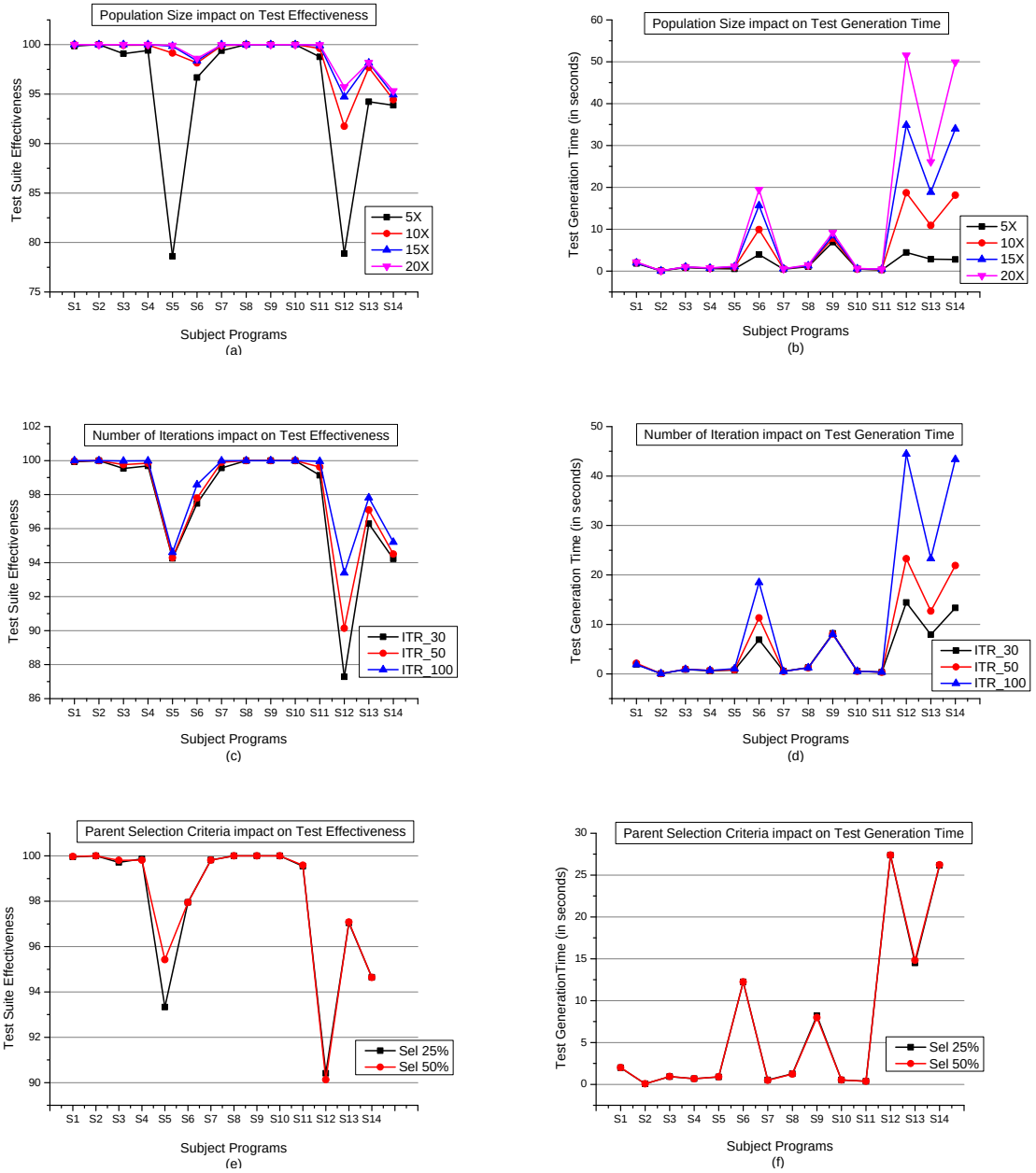

Figure 9. Impact of different parameters on mutation score and test generation time. (**a**,**b**) population size (**c**,**d**) number of iterations and, (**e**,**f**) parent selection criteria.

For each of the subject programs, the size of the population depends on the number of input variables. Performance is evaluated for four different sizes of the population, and we noticed that increasing the population size has a remarkable impact on test generation time. Handling a too-large population is more time-consuming. Our approach *tdgen_gamt* keeps mutation coverage and test generation time in the range of $[79, 100]$ and $[0.06, 4]$ (seconds) respectively, for the minimal population size; while for higher population, there is little improvement in coverage but it drastically increases the cost (Figure 9). Therefore, we can state that with $10\times$Inputs population, *tdgen_gamt* perform better when considering both measures, i.e., time and coverage.

Considering three different iteration limits, i.e., 30, 50, and 100, we find a drastic improvement in test effectiveness (mutation score) for only subject program S12 (Figure 9). It indicates that within 30 iterations, our approach successfully killed most of the mutants. However, there is a continuous

increment in test generation time for S6, S12, S13, and S14 programs. Considering both the parameters, the proposed approach is recommended to run for 30 generations.

To select how many parent test cases participate in reproduction, we experiment with the best 25% and 50% of test cases. Results (Figure 9) reveal that there is no improvement in mutation score except for S5. Therefore, we state that the fittest 25 % test cases are sufficient for parent selection.

4.6. Limitation of the Proposed Approach

The choice of mutation operators can significantly impact the test suite size and its effectiveness. A test case may be redundant for a set of mutants while it may not be so for other sets of mutants. Removing redundant tests could miss out some valuable test cases which otherwise may be good at detecting other types of faults. To establish a preponderance of the proposed approach, test cases were executed against traditional mutants also. The size and input type of the subject programs is a limitation of our study. At present, this study considers fixed integer inputs; however, it would be relevant to extend it for other data types, including dynamic arrays and strings. Moreover, further experimentation with varying and large program size can also be examined.

5. Conclusions

Test data generation is a time-consuming and critical process, which can be optimized using different search-based algorithms satisfying specific coverage measures. In the literature, mutation coverage is considered to be more powerful than other measures. However, taking mutation coverage as a stopping criterion may result in a large test suite. In this paper, a GA with the objective of low-cost mutation coverage (*tdgen_gamt*) is implemented for generating the test data. To generate the highly qualified test for fault detection, a fitness function is also proposed that maximizes the effectiveness as well as minimizes the complexity of each test case. Each test case in the solution set is different and non-redundant from others in killing the faults. To preserve the valuable test cases in each iteration, the concepts of 'elitism' along with 'intensification' and 'diversification' are employed; it speeds up the process of convergence.

A good number of experiments are performed on widely used 14 Java programs to tune the control parameters and to mitigate the effects of random generation. We compared the results with a popular automatic tool in academia Evosuite and Initial Random tests. These three techniques do not perform equally in identifying the mutants with low-cost test data. Major findings of this experimental work are listed below.

- Empirically, the obtained test suite could detect on average 87.83% (*tdgen_gamt*), 96.35% (Evosuite), and 79.6% (Initial Random tests) executable traditional faults irrespective of test suite size. This anomaly of preponderance can be explained on the account of measuring the effectiveness of the approach by considering the size of the test suite (test case efficiency). The proposed approach detects the maximum number of mutants with fewer and less complex test cases.
- Additionally, we also analyze the detection rate of each fault type from each of the approaches. The results report that *tdgen_gamt* could perform equally at finding the stubborn mutants. Meanwhile, only 0.3%, 1.1%, and 1.9% SDL-ODL mutants are identified as stubborn by *tdgen_gamt*, Evosuite, and Initial Random tests, respectively. This indicates *tdgen_gamt* successfully killed approximately all the mutants and may easily detect stubborn mutants.
- Also the removal of redundant tests raises the efficiency of the approach. In particular, based on the conducted study, our approach *tdgen_gamt* generates 13%, 205% more efficient and 18%, 60% reduced test suite than Evosuite and Initial Random tests respectively. A set of test cases that is redundant for one set of mutants may not be redundant for another set of mutants.
- During reproduction, crossover operation is performed only once on each parent test case. This choice of reproduction operator also lowers the time complexity of *tdgen_gamt*.
- The use of elitism helps in fast convergence.

- The suggested fitness function appropriately guides the search process by finding the highly effective and less complex test cases in terms of finding the faults.
- Our approach successfully qualifies the stability test and fails only 5% (on average) in identifying more than 90% mutants.
- Use of low-cost mutation operators (produces 80% fewer mutants than traditional mutants) makes it easily adaptable by others.

Author Contributions: All authors contributed equally to this work.

Funding: This research received no external funding.

Acknowledgments: The authors would like to acknowledge the Ministry of Electronics and Information Technology, Govt. of India for supporting this research under Visvesvaraya Ph. D. Scheme for Electronics and IT.

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

References

1. Dave, M.; Agrawal, R. Search based techniques and mutation analysis in automatic test case generation: A survey. In Proceedings of the 2015 IEEE International Advance Computing Conference (IACC), Bangalore, India, 12–13 June 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 795–799.
2. McMinn, P. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.* **2004**, *14*, 105–156. [CrossRef]
3. McMinn, P. Search-based software testing: Past, present and future. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11), Berlin, Germany, 21–25 March 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 153–163.
4. Sahin, S.; Akay, B. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. *Appl. Soft Comput.* **2016**, *49*, 1202–1214. [CrossRef]
5. Ali, S.; Briand, L.C.; Hemmati, H.; Panesar-Walawege, R.K. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.* **2010**, *36*, 742–762. [CrossRef]
6. Holland, J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*; MIT Press: Cambridge, MA, USA, 1992.
7. Yang, X.-S. *Nature-Inspired Optimization Algorithms*; Elsevier: Amsterdam, Poland, 2014.
8. Sivanandan, S.N.; Deepa, S.N. *Intorduction to Genetic Algorithms*; Springer: Berlin, Germany, 2008.
9. Fraser, G.; Zeller, A. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Softw. Eng.* **2012**, *38*, 278–292. [CrossRef]
10. Fraser, G.; Arcuri, A. Achieving scalable mutation-based generation of whole test suites. *Empir. Soft. Eng. Springer* **2015**, *20*, 783–812. [CrossRef]
11. Andrews, J.H.; Briand, L.C.; Labiche, Y. Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, MO, USA, 15–21 May 2005; ACM: Copenhagen, Denmark, 2005; pp. 402–411.
12. Just, R.; Jalali, D.; Inozemtseva, L.; Ernst, M.D.; Holmes, R.; Fraser, G. Are Mutants a Valid Substitute for Real Faults in Software Testing. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), Hong Kong, China, 16–22 November 2014; ACM: Copenhagen, Denmark, 2014; pp. 654–665.
13. Rad, M.; Akbari, F.; Bakht, A. Implementation of common genetic and bacteriological algorithms in optimizing testing data in mutation testing. In Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering (CiSE), Wuhan, China, 10–12 December 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 1–6.
14. Ma, Y.S.; Offutt, J. Description of method-level mutation operators for java. *Electronics and Telecommunications Research Institute, Korea*. 2005. Available online: https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf (accessed on 8 September 2019).
15. DeMillo, R.A.; Lipton, R.J.; Sayward, F.G. Hints on test data selection: Help for the practicing programmer. *Computer* **1978**, *11*, 34–41. [CrossRef]

16. Hamlet, R.G. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* **1977**, *3*, 279–290. [CrossRef]
17. Offutt, A.J.; Untch, R.H. Mutation 2000: Uniting the Orthogonal. In *Mutation Testing for the New Century, MUTATION 2001 Workshop*; Springer: Berlin, Germany, 2001; pp. 34–44.
18. Jia, Y.; Harman, M. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **2011**, *37*, 649–678. [CrossRef]
19. Grun, B.J.M.; Schuler, D.; Zeller, A. The Impact of Equivalent Mutants. In Proceedings of the 2009 International Conference on Software Testing, Verification, and Validation Workshops, Denver, CO, USA, 1–4 April 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 192–199.
20. Just, R.; Schweiggert, F. Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. *Softw. Test. Verif. Reliab.* **2015**, *25*, 490–507. [CrossRef]
21. Budd, T.A.; Angluin, D. Two Notions of Correctness and Their Relation to Testing. *Acta Inf. Springer* **1982**, *18*, 31–45. [CrossRef]
22. Offutt, A.J.; Pan, J. Detecting equivalent mutants and the feasible path problem. In Proceedings of the Eleventh Annual Conference on Computer Assurance, Systems Integrity, Software Safety. Process Security(COMPASS'96), Stockholm, Sweden, 17–21 June 1996; IEEE: Piscataway, NJ, USA, 1996; pp. 224–236.
23. Kintis, M.; Papadakis, M.; Jia, Y.; Malevris, M.; Traon, Y.L.; Harman, M. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Trans. Softw. Eng.* **2018**, *44*, 308–332. [CrossRef]
24. Untch, R. Mutation-based Software Testing Using Program Schemata. In Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92), Raleigh, NC, USA, 8–10 April 1992; ACM: Copenhagen, Denmark, 1992; pp. 285–291.
25. Offutt, A.J.; Lee, A.; Rothermel, G.; Untch, R.H.; Zapf, C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* **1996**, *5*, 99–118. [CrossRef]
26. Jia, Y.; Harman, M. Constructing Subtle Faults Using Higher Order Mutation Testing. In Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08), Beijing, China, 28–29 September 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 249–258.
27. Mateo, P.R.; Usaola, M.P. Reducing Mutation Costs through uncovered Mutants. *Softw. Test. Verif. Reliab.* **2015**, *25*, 464–489. [CrossRef]
28. Ma, Y.S.; Kim, S. Mutation testing cost reduction by clustering overlapped mutants. *J. Syst. Softw.* **2016**, *115*, 18–30. [CrossRef]
29. Just, R.; Kurtz, B.; Ammann, P. Inferring Mutant Utility from Program Context. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17), Toronto, ON, Canada, 17–21 July 2017; ACM: Copenhagen, Denmark, 2017; pp. 284–294.
30. Gopinath, R.; Ahmed, I.; Alipour, M.A.; Jensen, C.; Groce, A. Mutation Reduction Strategies Considered Harmful. *IEEE Trans. Reliab.* **2017**, *66*, 854–874. [CrossRef]
31. Jimenez, M.; Checkam, T.T.; Cordy, M.; Papadakis, M.; Kintis, M.; Traon, Y.L.; Harman,M. Are mutants really natural?: A study on how naturalness helps mutant selection. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'18), Oulu, Finland, 11–12 October 2018; ACM: Copenhagen, Denmark, 2018; pp. 1–10.
32. Papadakis, M.; Checkam, T.T.; Traon, Y.L. Mutant Quality Indicators. In Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Västerås, Sweden, 9–13 April 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 32–39.
33. Jimenez, M.; Checkam, T.T.; Cordy, M.; Papadakis, M.; Kintis, M.; Traon, Y.L.; Harman, M. Predicting the fault revelation utility of mutants. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE'18), Gothenburg, Sweden, 27 May–3 June 2018; pp. 408–409.
34. Pizzoleto, A.V.; Ferrari, F.C.; Offutt, J.; Fernandes, L. A systematic literature review of techniques and metrics to reduce the cost of mutation testing *J. Syst. Softw.* **2019**, *157*, 110388. [CrossRef]
35. Silva, R.A.; de Souza, S.; Do, R.S.; de Souza, P.S.L. A systematic review on search based mutation testing. *Inf. Softw. Technol.* **2017**, *81*, 19–35. [CrossRef]
36. Jatana, N.; Suri, B.; Rani, S. Systematic Literature Review on Search Based Mutation Testing. *e-Inf. Softw. Eng. J.* **2017**, *11*, 59–76.
37. Rodrigues, D.S.; Delamaro, M.E.; Correa, C.G.; Nunes, F.L.S. Using Genetic Algorithms in Test Data Generation: A Critical Systematic Mapping. *ACM Comput. Surv.* **2018**, *51*, 41:1–41:23. [CrossRef]

38. Zhu, Q.; Panichella, A.; Zaidman, A. A Systematic Literature Review on how mutation testing supports quality assurance processes. *Softw. Test. Verif. Reliab.* **2018**, *28*, e1675. [CrossRef]
39. Souza, F.C.; Papadakis, M.; Durelli, V.H.S.; Delamaro, M.E. Test Data Generation Techniques for mutation testing: A systematic mapping. In Proceedings of the 11th Workshop on Experimental Software Engineering Latin Americal Workshop (ESELAW), Pucón, Chile, 23–25 April 2014. Available online: http://pages.cs.aueb.gr/~mpapad/papers/eselaw2014.pdf (accessed on 8 September 2019)
40. Baudry, B.; Hanh, V.L.; Jezequel, J.; Traon, Y.L. Building trust into OO components using a genetic analogy. In Proceedings of the 11th International Symposium on Software Reliability Engineering, San Jose, CA, USA, 8–11 October 2000; IEEE: Piscataway, NJ, USA, 2000; pp. 4–14.
41. Baudry, B.; Hanh, V.L.; Traon, Y.L. Testing-for-trust: The genetic selection model applied to component qualification. In Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages, St. Malo, France, 5–8 June 2000; IEEE: Piscataway, NJ, USA, 2000; pp. 108–119.
42. Louzada, J.; Camilo-Junior, C.; Vincenzi, A.; Rodrigues, C. An elitist evolutionary algorithm for automatically generating test data. In Proceedings of the 2012 IEEE Congress on Evolutionary Computation (CEC), Brisbane, Australia, 10–15 June 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 1–8.
43. Subramanian, S.; Natarajan, N. A tool for generation and minimization of test suite by mutant gene algorithm. *J. Comput. Sci.* **2011**, *7*, 1581–1589. [CrossRef]
44. Haga, H.; Suehiro, A. Automatic Test Case Generation based on Genetic Algorithm and Mutation Analysis. In Proceedings of the IEEE International Conference on Control System, Computing and Engineering, Penang, Malaysia, 23–25 November 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 119–123.
45. Rani, S.; Suri, B. Implementing Time-Bounded Automatic Test Data Generation Approach Based on Search Based Mutation Testing. In *Progress in Advanced Computing and Intelligent Engineering*; Springer: Berlin, Germany, 2018; pp. 113–122.
46. Molinero, C.; Nunez, M.; Andres, C. Combining Genetic Algorithms and Mutation Testing to Generate Test Sequences. In Proceedings of the International Work-Conference on Artificial Neural Networks, Salamanca, Spain, 9–12 June 2009; Springer: Berlin, Germany, 2009; pp. 343–350.
47. Nilsson, R.; Offutt, J.; Mellin, J. Test Case Generation for Mutation-based Testing of Timeliness. *Electron. Notes Theor. Comput. Sci.* **2004**, *164*, 97–114. [CrossRef]
48. Bottaci, L. A genetic algorithm fitness function for mutation testing. In Proceedings of the International Workshop on Software Engineering, Metaheuristic Innovation Algorithms, Workshop of 23rd International Conference on Software Engineerng, Toronto, ON, Canada, 12–19 May 2001; pp. 3–7.
49. Masud, M.; Nayak, A.; Zaman, M.; Bansal, N. Strategy for mutation testing using genetic algorithms. In Proceedings of the IEEE CCECE CCGEI, Sankatoon, SK, Canada, 1–4 May 2005; IEEE: Piscataway, NJ, USA, 2005; pp. 1049–1052.
50. Mishra, K.K.; Tiwari, S.; Kumar, A.; Misra, A.K. An approach for mutation testing using elitist genetic algorithm. In Proceedings of the 3rd IEEE International Conference on Computer Science Information Technology, Chengdu, China, 9–11 July 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 426–429.
51. 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]
52. Almasi, M.M.; Hemmati, H.; Fraser, G.; Eler, M.; Arcuri, A.; Benefelds, J. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Buenos Aires, Argentina, 20–28 May 2017; pp. 263–272.
53. Shamshiri, S.; Just, R.; Rojas, J.M.; Fraser, G.; McMinn, P.; Arcuri, A. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Washington, DC, USA, 9–13 November 2015; pp. 201–211.
54. Shamshiri, S.; Rojas, J.M.; Gazzola, L.; Fraser, G. Random or Evolutionary Search for Object-Oriented Test Suite Generation? *Softw. Test. Verif. Reliab.* **2017**, *28*, 1–29. [CrossRef]
55. Gay, G. Detecting Real Faults in the Gson Library Through Search-Based Unit Test Generation. In Proceedings of the International Symposium on Search Based Software Engineering (SSBSE), Montpellier, France, 8–9 September 2018; Springer: Berlin, Germany, 2018; pp. 385–391.

56. Bashir, M.B.; Nadeem, A. A fitness function for evolutionary mutation testing of object-oriented programs. In Proceedings of the 9th International Conference on Emerging Technology, Islamabad, Pakistan, 9–10 December 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 1–6.
57. Bashir, M.B.; Nadeem, A. Improved Genetic Algorithm to Reduce Mutation Testing Cost. *IEEE Access* **2017**, *5*, 3657–3674. [CrossRef]
58. Bashir, M.B.; Nadeem, A. An Experimental Tool for Search-Based Mutation Testing. In Proceedings of the International Conference on Frontiers of Information Technology, Islamabad, Pakistan, 17–19 December 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 30–34.
59. Bashir, M.B.; Nadeem, A. An Evolutionary Mutation Testing System for Java Programs: eMuJava. In *Intelligent Computing, Proceedings of the Intelligent Computing—Proceedings of Computing Conference (CompCom'19), London, UK, 16–17 July 2019*; Springer: Berlin, Germany, 2019; Volume 998, pp. 847–865.
60. Delgado-Pérez, P.; Medina-Bulo, I. Search-based mutant selection for efficient test suite improvement: Evaluation and results. *Inf. Softw. Technol.* **2018**, *104*, 130–143. [CrossRef]
61. Delgado-Pérez, P.; Rose, L.M.; Medina-Bulo, I. Coverage-based quality metric of mutation opertaors for test suite improvement. *Softw. Qual. J.* **2019**, *27*, 823–859. [CrossRef]
62. Delgado-Pérez, P.; Medina-Bulo, I.; Segura, S.; Garcia-Dominguez, A.; Jose, J. GiGAn: Evolutionary mutation testing for C++ object-oriented systems. In Proceedings of the Symposium on Applied Computing, Marrakech, Morocco, 3–7 April 2017; ACM: Copenhagen, Denmark, 2017; pp. 1387–1392.
63. Ghiduk, A.S.; El-Zoghdy, S.F. CHOMK: Concurrent Higher-Order Mutants Killing Using Genetic Algorithm. *Arab. J. Sci. Eng.* **2018**, *43*, 7907–7922. [CrossRef]
64. Rani, S.; Suri, B. An approach for Test Data Generation based on Genetic Algorithm and Delete Mutation Operators. In Proceedings of the Second International Conference on Advances in Computing and Communication Engineering, Rohtak, Dehradun, India, 1–2 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 714–718.
65. Singh, Y. *Software Testing*; Cambridge University Press: Cambridge, UK, 2012.
66. Untch, R.H. On reduced neighborhood mutation analysis using a single mutagenic operator. In Proceedings of the 47th Annual Southeast Regional Conference, New York, NY, USA, 19–21 May 2009; ACM: Copenhagen, Denmark, 2009; pp. 71:1–71:4.
67. Luke, S. *Essentials of Metaheuristics, Lulu*. 2009. Available online: https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf (accessed on 8 September 2019).
68. Chakraborty, U.K.; Janikow, C.Z. An Analysis of Gray versus Binary Encoding in Genetic Search. *Inf. Sci.* **2003**, *156*, 253–269. [CrossRef]
69. Gaffney, G.; Green, D.A.; Pearce, C.E.M. Binary versus real coding for genetic algorithm: A false dichotomy? *ANZIAM J.* **2009**, *51*, C347–C359. [CrossRef]
70. Varshney, S.; Mehrotra, M. Search-Based Test Data Generator for Data-Flow Dependencies Using Dominance Concepts, Branch Distance and Elitism. *Arab. J. Sci. Eng.* **2016**, *41*, 853–881. [CrossRef]
71. Fraser, G.; Wotawa, F. Redundancy Based Test-Suite Reduction. In Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering, Braga, Portugal, 24 March–1 April 2007; Springer: Berlin, Germany, 2007; pp. 291–305.
72. Scheibenpflug, A.; Wagner, S. An Analysis of the Intensification and Diversification Behavior of Different Operators for Genetic Algorithms. In Proceedings of the International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, 10–15 February 2013; Springer: Berlin, Germany, 2013; pp. 364–371.
73. Wei, W.; Li, C.M.; Zhang, H. A Switching Criterion for Intensification and Diversification in Local Search for SAT. *J. Satisf. Boolean Model. Comput.* **2008**, *4*, 219–237.
74. Yang, X.-S.; Deb, S.; Fong, S. Metaheuristic Algorithms: Optimal Balance of Intensification and Diversification *Appl. Math. Inf. Sci.* **2014**, *8*, 977–983. [CrossRef]
75. Baluja, S.; Caruana, R. Removing the Genetics from the Standard Genetic Algorithm. In Proceedings of the Twelfth International Conference on International Conference on Machine Learning (ICML'95 P), Tahoe City, CA, USA, 9–12 July 1995; Morgan Kaufmann Publishers: Burlington, MA, USA, 1995; pp. 38–46.
76. Guan, B.; Zhang, C.; Ning, J. Genetic algorithm with a crossover elitist preservation mechanism for protein–ligand docking. *AMB Express* **2017**, *7*, 174. [CrossRef] [PubMed]

77. Estero-Botaro, A.; Palomo-Lozano, F.; Medina-Bulo, I.; Domínguez-Jiménez, J.J.; García-Domínguez, A. Quality metrics for mutation testing with applications to WS-BPEL compositions *Softw. Test. Verif. Reliab.* **2015**, *25*, 536–571. [CrossRef]
78. EvoSuite—Automated Generation of JUnit Test Suites for Java Classes. Available online: https://github.com/EvoSuite/evosuite/ (accessed on 8 May 2019).
79. Liang, Y.D. *Introduction to Java Programming*; Pearson Education Inc.: Upper Saddle River, NJ, USA, 2011.
80. Ammann, P.; Offutt, J. *Introduction to Software Testing*; Cambridge University Press: Cambridge, UK, 2012.
81. Apache Commons Math. Available online: https://github.com/apache/commons-math (accessed on 8 May 2019).
82. Smallest Largest. Available online: https://github.com/VMehta99/SmallestLargest/blob/master/SmallestLargest.java (accessed on 8 August 2018).
83. Software-Artifact Infrastructure Repository. Available online: http://sir.unl.edu/portal/index.php (accessed on 8 May 2019).
84. Ma, Y.S.; Offutt, J.; Kwon, Y.R. Mujava: An automated class mutation system. *Softw. Test. Verif. Reliab.* **2005**, *15*, 97–133. [CrossRef]
85. Deng, L.; Offutt, J.; Li, N. Empirical evaluation of the statement deletion mutation operator. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–20 March 2013; IEEE Computer Society: Washington, DC, USA, 2013; pp. 84–93.
86. Delamaro, M.E.; Offutt, J.; Ammann, P. Designing deletion mutation operators. In Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, Cleveland, OH, USA, 31 March–4 April 2014; IEEE Computer Society: Washington, DC, USA, 2014; pp. 11–20.
87. Basili, V.R.; Shull, F.; Lanubile, F. Building knowledge through families of experiments. *IEEE Trans Softw. Eng.* **1999**, *25*, 456–473. [CrossRef]
88. Patrick, M.; Jia, Y. KD-ART: Should we intensify or diversify tests to kill mutants? *Inf. Softw. Technol.* **2017**, *81*, 36–51. [CrossRef]
89. Gonzalez-Hernandez, L.; Offutt, J.; Potena, P. Using Mutant Stubbornness to Create Minimal and Prioritized Test Sets. In Proceedings of the International Conference on Software Quality, Reliability, and Security, Lisbon, Portugal, 16–20 July 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 446–457.