symmetry

MDPI

Article

A Novel Real Coded Genetic Algorithm for Software Mutation Testing

Deepti Bala Mishra [1], Biswaranjan Acharya [2,*], Dharashree Rath [1], Vassilis C. Gerogiannis [3,*] and Andreas Kanavos [4,*]

[1] GITA Autonomous College, Bhubaneswar 752054, India
[2] Department of Computer Engineering-AI, Marwadi University, Rajkot 360001, India
[3] Department of Digital Systems, University of Thessaly, Geopolis Campus, 45100 Larissa, Greece
[4] Department of Digital Media and Communication, Ionian University, 28100 Kefalonia, Greece
* Correspondence: biswaranjan.acharya@marwadieducation.edu.in (B.A.); vgerogian@uth.gr (V.C.G.); akanavos@ionio.gr (A.K.)

Abstract: Information Technology has rapidly developed in recent years and software systems can play a critical role in the symmetry of the technology. Regarding the field of software testing, white-box unit-level testing constitutes the backbone of all other testing techniques, as testing can be entirely implemented by considering the source code of each System Under Test (SUT). In unit-level white-box testing, mutants can be used; these mutants are artificially generated faults seeded in each SUT that behave similarly to the realistic ones. Executing test cases against mutants results in the adequacy (mutation) score of each test case. Efficient Genetic Algorithm (GA)-based methods have been proposed to address different problems in white-box unit testing and, in particular, issues of mutation testing techniques. In this research paper, a new approach, which integrates the path coverage-based testing method with the novel idea of tracing a Fault Detection Matrix (FDM) to achieve maximum mutation coverage, is proposed. The proposed real coded GA for mutation testing is designed to achieve the highest Mutation Score, and it is thus named RGA-MS. The approach is implemented in two phases: path coverage-based test data are initially generated and stored in an optimized test suite. In the next phase, the test suite is executed to kill the mutants present in the SUT. The proposed method aims to achieve the minimum test dataset, having at the same time the highest Mutation Score by removing duplicate test data covering the same mutants. The proposed approach is implemented on the same SUTs as these have been used for path testing. We proved that the RGA-MS approach can cover maximum mutants with a minimum number of test cases. Furthermore, the proposed method can generate a maximum path coverage-based test suite with minimum test data generation compared to other algorithms. In addition, all mutants in the SUT can be covered by less number of test data with no duplicates. Ultimately, the generated optimal test suite is trained to achieve the highest Mutation Score. GA is used to find the maximum mutation coverage as well as to delete the redundant test cases.

Keywords: Fault Detection Matrix; Mutation Score; path coverage-based testing; real coded Genetic Algorithm; test case generation; test data optimization

Citation: Mishra, D.B.; Acharya, B.; Rath, D.; Gerogiannis, V.C.; Kanavos, A. A Novel Real Coded Genetic Algorithm for Software Mutation Testing. *Symmetry* **2022**, *14*, 1525. https://doi.org/10.3390/sym14081525

Academic Editor: Zhixun Su

Received: 14 June 2022
Accepted: 22 July 2022
Published: 26 July 2022

1. Introduction

Software is used to control the entire hardware system. Due to this, it has become part of the most complex components and its use has grown more and more. Indeed, its growth has caused new algorithms and development methodologies to be implemented, and thus, software is being applied to a large number of different areas. What is more, software can also support symmetry as its use has been widespread across entire systems and independent content. Symmetry can be identified as an extraordinary characteristic that has been widely deployed in diverse research fields of computer engineering.

Software testing constitutes the process of executing a software product or a portion of it in a controlled environment with a given set of input (test cases) [1]. Moreover, it involves the execution and examination of a program or a software system in order to identify errors or faults. The main goal of software testing is to determine the errors in a complete software product (or in a component of it) to ensure a high probability that the corresponding software is correct [2].

Search-Based Software Engineering (SBSE) is an emerging field of research and practice in Software Engineering, where the term "search" refers to the metaheuristic search-based optimization techniques used. SBSE aims to reformulate problems in the area of software engineering, particularly in software testing problems, into search-based optimization problems. A comprehensive survey of SBSE is proposed in Ref. [3], where related research fields are organized into categories drawn from the corresponding ACM subject categories within the Software Engineering body of knowledge.

In software testing, mutants form simulated and artificially generated faults that behave in a similar way to the realistic ones [4,5]. In addition, mutants can be used for test data generation in a software testing activity [6]. These are created by systematic injection of faults using some predefined mutation operators [7]. Mutation testing is thus a form of white-box testing initially suggested in Ref. [8] and later explored by different researchers [9–11]. Execution of a test case (test inputs) against mutants results in the adequacy score of that test case, where this result is also called the Mutation Score (MS).

In recent years, a number of search-based algorithms, metaheuristic methods, Evolutionary Algorithms (EA) and optimization techniques have already been developed to automatically generate test data in the area of software testing. Different EAs like Genetic Algorithms (GA), PSO, ACO, Monarch Butterfly Optimization Algorithm (MBO), Slime Mold Algorithm (SMA), Moth Search Algorithm (MS), RUNge–Kutta method (RUN), Colony Predation Algorithm (CPA), Weighted Mean of Vectors (INFO), and Harris Hawks Optimization (HHO) have been proposed in unit level testing to generate optimized test data for path testing and these path coverage-based testing methods can only generate test data to traverse target paths [12]. However, these techniques may fail to detect all software faults because information about fault detection is not incorporated into the process of generating test data [13]. So, existing path coverage-based techniques may not always guarantee that the test data can effectively detect all faults in target paths. Hence, it is vital to implement the best of each test case through Mutation Analysis [14].

Monarch Butterfly Optimization Algorithm (MBO) employs the migration behaviors of butterflies where all monarch butterflies are located at two different lands. These lands are fixed and unchanged during the whole optimization process. A parameter p is used for calculation at the beginning stage of the search process [15]. The Slime Mold Algorithm basically refers to Physarum Polycephalum [16]. It is an eukaryote that inhabits cool and humid places. Also, the main nutritional stage is considered the Plasmodium; in this stage, the organic matter in slime mold seeks food, surrounds and secretes enzymes with the aim of digesting it. In the optimization process, the front end extends into a fan-shape, followed by an interconnected venous network that allows cytoplasm to flow inside. It can be used for multiple food sources at the same time to form a venous network connecting them. This metaheuristic algorithm is mainly used to solve many graph theory problems and also for generating different networks.

In addition, moths are like butterflies that belong to the order Lepidoptera. The main features of moths are Phototaxis, signifying movement of an organism towards or away from a source of light and Levy flights. Taking all the features of moths, a new kind of metaheuristic algorithm has been developed, called Moths Search Algorithm (MSA) [17]. The phototaxis and levy flights of the moths can be used to build up a general-purpose optimization method. Also, the Colony Predation Algorithm (CPA) is based on the co-existence of animals and mimics the supportive behavior of social animals as well as the predation strategy of hunting animals [18]. The different operations applied in CPA are communications and collaboration, disperse food, encircle foods, supporting the closest

individual and searching for foods. Through these five processes, the global optimum solution can be achieved.

The RUNge–Kutta method (RUN) is an iterative process used to solve the numerical Cauchy problems for a system of ordinary differential equations. To solve the differential equations, both the arithmetic and the geometric mean are applied [19]. The most important strategy of Harris Hawks Optimization (HHO) is catching its prey in order to hunt in groups and to collaborate with the hawks, as opposed to other predators [20]. In this process, a number of hawks attack from different ways their chosen prey in collaboration with simultaneous delusion and approached in a controlled manner. The attack is desired to be completed in a few seconds and it continues until the hunting is successful or, on the other hand, the prey manages to completely escape. Finally, the hunting process is completed as the prey, which has low energy and has lost its defensive abilities, is easily hunted by the leader. HHO is a population-based optimization technique and it can be applied to any optimization problem with appropriate limitations and constraints. The optimum solution can be found in the prey itself.

Although there are various studies related to generating test data for path coverage, few researchers have proposed techniques to detect faults via path coverage-based test data. The purpose of this particular research is to generate test data for path coverage as well as to detect faults lying in the System Under Test (SUT). In the relevant literature, some of these methods not only generate test data with the aim of covering a single or multiple paths, but also attempt to find the mutation score of every test case for a specific SUT [21].

In this paper, we propose an SBSE algorithm that can effectively identify the maximum mutation score (coverage) by taking into account all the corresponding test cases, which are generated through path coverage-based testing. Initially, path coverage-based test cases are generated and then the proposed algorithm aims to find the mutation score for every test case by employing different mutation operators. In particular, a real coded Genetic Algorithm (GA) is proposed, which automatically generates test data to cover multiple paths at a time and further exercises the test data to achieve the maximum mutation score. The proposed algorithm is abbreviated as RGA-MS (Real coded Genetic Algorithm for maximizing the Mutation Score).

The rest part of the paper is organized in different sections as follows: In Section 2 some basic concepts on mutation testing are described along with some relevant studies. In Section 3, the steps of the proposed RGA-MS algorithm are outlined. Experimental setup along with the analysis of the experiment results are discussed in Section 4. The same Section also presents a comparative study of the proposed approach with other existing methods focusing on achieving the highest mutation score, and the threats to validity regarding the proposed technique. Finally, our research's conclusion and future scopes are outlined in Section 5.

2. Background

In this section, an overview of mutation testing and GA fields of research is presented.

2.1. Mutation Testing

Mutation testing is a fault-based white-box software testing method. Its role is to evaluate and to improve the quality of test data by killing the active mutants present in the SUT [22]. During a mutation testing process, the SUT is initially tested with use of the test cases designed by the coverage-based testing. After the coverage-based testing is completed, mutation testing is performed. Using this technique, certain mutation operators are inserted into the code to mutate the program [23]. The mutated program is then tested against corresponding test cases previously designed to kill the mutants. If any test case detects a mutant, then this mutant has to be killed; otherwise, it is still considered as being alive [24]. Therefore, the main goal of mutation testing is to select the most efficient test data (i.e., those with efficient error detection capabilities) and to distinguish the initial program from the seeded mutants [25].

Even though test sets can be selected as well as generated by a tester, these manual practices are often complex and time-consuming, so the need for automated test case generation arises. Although test data generation is known to be a complex process, several researchers have proposed techniques for the automation of this task. In this field, a systematic mapping study, whose purpose is to identify test data generation approaches based on mutation testing, can be found in Ref. [26].

Furthermore, Search-based Mutation Testing uses meta-heuristic optimization techniques for mutation testing, like Genetic Algorithm, Hill Climbing, and other evolutionary approaches. The survey paper in Ref. [27] discusses the challenges and opportunities faced in applying search-based techniques for mutation testing. It is proven that the major challenge consists of the computational cost as well as the storage for a large set of mutants.

Mutation testing may overcome some limitations of other testing approaches, but it is often considered costly. Evolutionary algorithms can be used to reduce the cost of data generation in different testing methodologies. As a result, the authors in Ref. [28] introduced a novel strategy with the elitist genetic algorithm for generating efficient test input data in the context of mutation testing. Search-Based Software Testing (SBST) has also been applied to the generation of weakly adequate mutation-based test data [29,30]. Specifically, SBST was firstly used with the aim of killing mutants in Ref. [31].

Mutation testing is executed through a variety of different steps, which are summarized in Refs. [22,23,32]:

- Mutants construction regarding the program under test;
- Implementation of the test cases with the use of the mutation system to verify the program output;
- If the output is incorrect, then the test case detects a fault and the program must be alternated in order to restart the mutation testing process;
- If the output is found to be correct, then the test case is executed against each live mutant;
- After each test case has been executed against each live mutant, the remaining mutants are either equivalent or killed. An equivalent mutant always produces the same output as the original program, so there is no test case to kill it. However, for live mutants, new test cases must be created and the process is repeated until all mutants are killed;
- The efficiency of the mutation testing can be assessed by utilizing a higher mutation score, which is defined by the following Equation (1) [32].

$$MS = \frac{\text{Total Mutants Killed}}{\text{Total Mutants Present In The Program}} \times 100 \quad (1)$$

2.2. Real Coded GA

A Genetic Algorithm is initialized with a number of initial solutions, entitled initial population. Successively, it forms the new population or new off-springs from the old population based on the fitness value of the population towards the problem. The more the fitness value, the more they will be suitable to reproduce [33]. The process of a new off-spring generation starts with the selection of parents based on a specific selection process. After the parent selection process, the new off-spring is generated using two different operators, i.e., crossover and mutation [34].

The most critical task in software mutation testing is to generate test data, which could kill maximum mutants present in the SUT. In this case, the optimization problem is the achievement of the highest mutation score with a lesser number of test data. From literature, many evolutionary algorithms have already been used to solve this particular problem. So, considering mutation coverage as the test adequacy criteria, the GA-based method is proposed in this paper to identify the optimized test cases, which give maximum mutation score. GA is used in terms of automation process and objective function is formulated to find maximum mutation score for the SUT.

2.3. Related Work

In literature, different evolutionary algorithms have often been utilized to address the issues identified in mutation testing. A number of corresponding research approaches, from the domain of test case generation as well as optimization for mutation testing using different GA-based techniques, are presented in this section.

A hybrid GA (HGA)-based method for test case optimization, which combines the features of GA along with local search techniques, is proposed in Ref. [35]. This particular approach is employed in terms of test case generation and optimization regarding path coverage and mutation analysis. Different mutation operators for calculating the mutation score have been presented. From the experimental evaluation, the authors proved that the recommended HGA-based method generates both locally and globally optimal solutions with rapid convergence. Another work introduces a GA-based technique for mutation analysis through automatic test case generation [25]. The three following mutation operators, e.g., Condition Overall Replacement (COR), Condition Statement Replacement (CSR) and Branch Value Increment (BVI), have been used to kill the mutants. This particular GA-based approach reduces the number of test cases using the Fault Detection Matrix (FDM).

Authors in a similar work developed a GA-based method to solve their formulated model and then applied this technique to several real-world programs [36]. The experimental results confirm that the proposed approach can generate test data for traversing the target path and detecting faults in a SUT. The authors in Ref. [12] presented a novel method to generate test data for covering multiple paths at a time. The proposed method can not only simultaneously cover multiple paths but can also detect faults present in the SUT. A weighted GA-based model is constructed to solve their multi-objective optimization problem. This technique is compared with different methods, like the random method and the methods proposed in Refs. [36,37]. The reported results confirm that the proposed technique can efficiently generate test data for traversing target paths and detect maximum faults in the SUT.

A novel method for the automatic generation of test data has been pertained in Ref. [38]. The authors have used GA for mutant analysis to achieve the maximum mutation score, where the proposed GA-based approach achieves optimized test data without any redundancy. The reported results in mutation analysis can be benefited from the Delete operator. The authors in Ref. [39] introduced another GA-based technique for the automatic generation of test cases. The produced test suite is in addition implemented for mutation analysis, where the mutation score is computed according to the total number of mutants killed from a particular test suite. Specifically, only one program for computing the power value to perform the experiments, has been employed. The derived experimental results have demonstrated that the utilized method can achieve 100% mutation score, also achieving better test cases. In another similar work [40], a GA-based method for optimizing software testing efficiency by taking into account path coverage and boundary coverage-based test data, has been introduced. A model that reveals faults and kills a mutant with use of genetic algorithms, is introduced in Ref. [41]. The source program along with the mutant are instrumented so that each unit's input as well as output's behavior can be tracked, where a checker module is employed in order to compare and track each unit's output.

Another hybrid GA (HGA)-based method for the automatic production of test cases for mutation testing has been introduced in Ref. [42]. Authors utilized SUT data flow information for the computation of mutation scores. Moreover, the source code of a software program that generates prime numbers within a specific range for their experiments, has been utilized along with the development of a tool in *C#* to generate Control Flow Graph (CFG) for a particular program developed in *C*. A comparison of this approach against both random and GA-based methods has been performed and the results depicted that this particular HGA-based method generates a maximum mutation score. Moreover, the work in Ref. [43] suggests dummy variables introduction into some already defined constraints regarding the specification; the proposed method utilizes the constraints in input as well as

output variables with the aim of guiding the generation of the test data. This is achieved on the contrary to the ordinary specification-based testing (SBT) methods that concern only constraints over input variables. This approach employs the conjunction of three techniques, e.g., GA, mutation testing, and SBT.

A novel two-way crossover and adaptable mutation method for generating better offspring was presented in Ref. [44]. Their introduced genetic algorithm is compared with both normal and random GA methods and proved to identify the optimal test cases in a smaller number of attempts with higher mutation scores, a fact that leads to reduced cost of computations. In Ref. [24], a novel approach for mutation testing called Particle Swarm Optimization along with Mutation Testing (PSO-MT) to generate test data for killing active mutants, was introduced. More to this point, more extensive programs in terms of benchmarks from the Software-artifact Infrastructure Repository (SIR) for their experiments, have been considered. The experimental results have been compared with GA-based results in test data generation and it has been found that PSO-MT achieved better than GA-based mutation testing in terms of computational efficiency. Specifically, five different benchmark programs for the experiments, e.g., triangle, quadratic equation, TCAS, Schedule 1 and 2, have been employed. The reported results have shown that the proposed method can achieve a maximum value of mutation score equals to 79.32%.

Dubbed Sentinel [45–48] is a novel multi-objective EA-based approach for automatic generation of test data with optimal cost reduction strategies for every program. This approach was evaluated by conducting a thoroughly empirical study from 40 releases of 10 open-source real-world software systems with the execution of 4800 experiments with the use of quality indicators and statistical significance tests. The reported results have depicted that strategies computed by the Sentinel approach outperform the baseline strategies in 95% of the cases when considering larger sizes. Furthermore, the proposed method can automatically generate mutation strategies with reduced mutation testing cost and simultaneously without affecting its testing effectiveness in terms of mutation score.

An improved fitness evaluation along with the incorporation of elitism as well as a performance comparison with the existing techniques is performed in Ref. [49]. A GA variant, by effectively blending the advantages of mutation testing for non-redundant test suite generation, was implemented. This variant is followed by a novel fitness function that considers test case complexity in terms of time steps with high fault exposure. GAMuT is another method for generating a test suite for the testing of Finite State Machines (FSMs) by using a Genetic Algorithm and mutation testing [50]. A standard mutant dataset tool, called MutantBench, was employed in Ref. [51] where an Abstract Syntax Tree (AST) was used in conjunction with a tree-based convolutional neural network (TBCNN) for the classification of mutants with the aim of reducing the cost of mutation testing in software testing of android applications.

3. Proposed Algorithm for Mutation Testing (RGA-MS)

The proposed algorithm in this paper is designed to identify and extract a representative test suite, which achieves maximum path coverage. In the following, this test suite is exercised for mutation testing, i.e., to find the maximum mutation score. The algorithm is entitled Real Coded Genetic Algorithm for highest Mutation Score (RGA-MS) and is outlined in Algorithm 1.

Initially, the RGA-MS generates the test data for maximum path coverage by using the fitness, which is designed by considering the total number of paths covered by a chromosome. The fitness for path coverage is defined in following Equation (2) as it is calculated by the ratio of paths covered by a chromosome given the total number of paths.

$$f(x) = \frac{\text{Paths Covered By A Chromosome}}{\text{Total Number Of Paths}} \times 100 \quad (2)$$

In the following, the two best populations are selected and the maximum path coverage is calculated. If the maximum path coverage is achieved, then the optimized test

suite can be printed, otherwise the mutation must be applied and the new chromosomes should be replaced with their duplicates. The generated test data are exercised to find the mutation score.

Different mutation operators are inserted in the source code of different SUTs, which are listed in Table 1.

Algorithm 1 Real Coded Genetic Algorithm for highest Mutation Score (RGA-MS)

input Set of basic paths for a specific SUT (Table 1)
    GA parameters
output An optimized test data with maximum Mutation Score (MS)
Step 1: Create *Initial Population*()
Step 2: Initialize the Population as Test Suites (where Test Suite = number of Chromosomes)
Step 3: Calculate Fitness using Equation (2) /* Fitness Value */
Step 4: Select two best Populations
Step 5: do
        Average Crossover for selected Chromosomes /* Maximum Path Coverage */
while (*Feasible Solution Reached*)
Step 6: if (*Maximum Path Coverage Achieved***)**
        Goto **Step 8**
    else
        Apply Mutation()
Step 7: do
        Replace the new Chromosome with the duplicate one
while (*Critical Path is not found*)
Step 8: if (*Maximum Path Coverage Achieved***) then**
        Print the optimized Test Suite with 100% Path Coverage
    else
        Goto **Step 1**
Step 9: *Trace FDM*()
Step 10: Remove redundant test data covering same mutants
Step 11: *Calculate Mutant Score MS*() using Equation (1)
Step 12: Print optimized test data with highest MS

Table 1. SUT's details.

Program Name	Description	LOC	Paths
P#1: TCP	Check the type of the Triangle	28	25
P#2: Max three	Find Maximum of three numbers	25	4
P#3: Evaluate X^Y	Determine the value of X^Y	27	3
P#4: Evaluate X/Y	Determine the value of X/Y	38	5
P#5: GCD	Returns the greatest common divisor	18	5
P#6: QES	Find the roots of Quadratic Equation	42	6
P#7: Calculate Days	Find Days between two Dates	153	18
P#8: Binary Search Tree	An ordered or sorted binary tree	119	16
P#9: Calculator	Find the results of arithmetic operations	60	9
P#10: Accept Saving	Banking process to deposit in saving account	24	8

The proposed algorithm aims to identify the minimum test data that covers all mutants by eliminating the redundant test data. The redundant test data is traced using a Fault Detection Matrix (FDM) [25]. A fault-detection matrix contains sufficient information for finding minimal-length, fault-diagnosis test sets. The necessary condition is that any sub-matrix of this matrix should not contain equal rows. The flow of the proposed algorithm is shown in Figure 1.

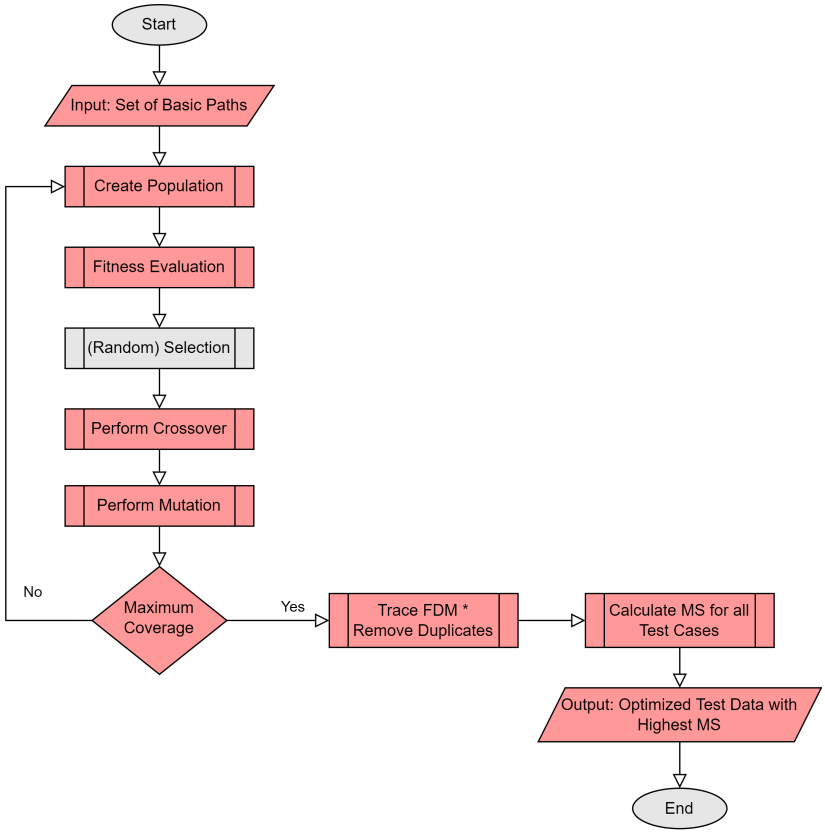

Figure 1. Flow of the proposed Algorithm RGA-MS for mutation analysis.

4. Results

4.1. Experimental Setup

The proposed algorithm is first implemented on a particular case study, entitled as the Greatest Common Divisor (GCD), which produces the greatest common divisor of any two input numbers. The source code for GCD with two integers as input, namely m and n, is illustrated in Listing 1. Furthermore, the proposed algorithm is implemented on different SUTs, as listed in Table 1. Moreover, the Control Flow Graph of the GCD Function is illustrated in Figure 2.

Listing 1: Source code of the GCD program

```
int GCD(int m, int n) {
    int r;
    if(n>m){
        r = m;
        m = n;
        n = r;
    }
    r = m % n;
    while(r!=0){
        m = n;
        n = r;
        r = m % n;
    }
    return n;
}
```

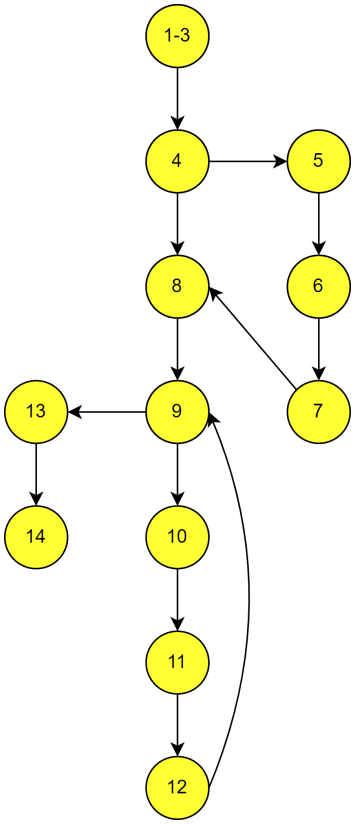

Figure 2. Control Flow Graph (CFG) of the GCD Function.

The various mutation operators [38,44] used for mutation analysis are listed in Table 2. Three different columns are listed, namely the type, the description, as well as the corresponding symbols. For example, for type equals to NOR, the description is "No Replacement Operator" and these operators present as they are.

Table 2. Different operators used for mutation analysis.

Type	Description	Symbols
AOR	Arithmetic Operator Replacement	+, −, *, /, %
LOR	Logical Operator Replacement	==, !=
ROR	Relational Operator Replacement	>, <, >=, <=
NOR	No Replacement Operator	Operator present as it is
UOI	Unary Operator Insertion	++, − −
CDL	Constant Deletion	Delete a constant (if present)
ABS	Absolute Value Insertion	1 is replaced with 0 and vice versa

Regarding the Control Flow Graph of the GCD function, which is presented in Figure 2, we notice that one extra path is added in order to perform mutation analysis, as in the GCD program, the number of linearly independent paths is small, i.e., three. So, the different paths for GCD are listed below:

- PATH 1: 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14;
- PATH 2: 1, 3, 4, 8, 9, 10, 11, 12, 13, 14;
- PATH 3: 1, 3, 4, 8, 9, 13, 14;
- PATH 4: 1, 3, 4, 5, 6, 7, 8, 9, 13, 14.

Regarding mutation analysis, some faults are inserted in the aforementioned source code. The proposed method aims to achieve maximum mutation score through the path coverage-based test data. The following Table 3 introduces the details of mutants inserted to the GCD program. Concretely, for specific lines of the program, some operators can be replaced by others and the corresponding mutation will be utilized.

The fitness value proposed before is also considered for computing the maximum path coverage, as defined in Equation (2). Finally, the optimized path coverage-based test suite is further exercised for mutation analysis. So, the mutation score of each test data is calculated using, as previously, Equation (1).

Furthermore, regarding the genetic algorithm operation, Table 4 presents the parameters used for mutation analysis. These parameters have specific values, e.g., the population size equals 1000, and Gaussian is considered as the mutation type.

Table 3. Operators used in GCD for mutation analysis.

Line Number	Operator Present	Replace with Operator	Mutant Name
4	>	!=	M1
6	=	/ =	M2
8	%	/	M3
9	!=	==	M4
12	%	/	M5

Table 4. Parameters used for mutation analysis.

SI. Number	Parameters	Values
1	Population size	1000
2	Input range	1 to 100
3	Encoding type	Real Encoding
4	Crossover Type and Rate	Average Crossover (AX), 0.8
5	Mutation Type and Rate	Insertion, Gaussian 0.02, 0.07
6	Generation	5

4.2. Result Analysis

The various steps of the proposed algorithm and the results obtained in GA operations are depicted in Tables 5–8. Specifically, Table 5 presents the initial population for five

particular test suites, where the corresponding test data, the target path, and the fitness value are also introduced. The fitness value ranges from 0.25 to 0.75.

Table 5. Initial population.

Test Suite Number	Test Data	Target Path	Fitness
1	(12,4), (8,27), (45,8), (9,44)	3, 1, 3, 1	0.5
2	(14,9), (23,8), (33,45), (14,5)	2, 2, 2, 3	0.5
3	(49,9), (7,33), (28,5), (39,8)	2, 1, 2, 2	0.5
4	(7,12), (6,18), (16,4), (9,42)	1, 4, 3, 1	0.75
5	(32,6), (44,16), (20,7), (17,12)	2, 2, 2, 2	0.25

In addition, Table 6 introduces the two best populations, which are test suites with number 1 and 4 having fitness values equal to 0.5 and 0.75, respectively.

Table 6. Best population.

Test Suite Number	Test Data	Target Path	Fitness
1	(12,4), (8,27), (45,8), (9,44)	3, 1, 3, 1	0.5
4	(7,12), (6,18), (16,4), (9,42)	1, 4, 3, 1	0.75

After the crossover process, a new trait has been generated, with a fitness value equal to 0.75, whereas the test data and the target path have different values than the 5 initial test data as presented in Table 7.

Table 7. New trait after crossover.

New Trait	Test Data	Target Path	Fitness
1	(9,8), (7,22), (30,6), (19,43)	2, 1, 3, 1	0.75

The optimized test suite that covers maximum path, after the mutation, is presented in Table 8. As before, the test data and the target path have different values than the 5 initial test data whereas the fitness value achieves percentage equal to 100%.

Table 8. Optimized test suite after mutation.

Test Suite	Test Data	Target Path	Fitness
(T1, T2, T3, T4)	(21,7), (14,9), (7,22), (6,18)	1, 2, 3, 4	100%

The FDM is traced out from the optimized test suite, which indicates the mutant coverage by individual test cases as shown in Table 9. This table presents all the potential combinations between the five mutants and the four test suites.

Table 9. FDM for GCD.

Test Suite / Mutant	T1	T2	T3	T4
M1	1	0	1	0
M2	0	0	1	1
M3	0	1	1	0
M4	1	1	1	1
M5	0	1	1	0

Regarding the detection matrix, a value equal to 1 represents the case where a mutant is covered by the corresponding test case (and otherwise, for a value equal to 0). Next, the

mutation score is calculated using Equation (1), and the mutation score for every test case is presented in Table 10. Finally, the redundant test cases, which cover the same mutants, are eliminated in order to find the optimum test data. We observe that the third test case achieves a mutation score equal to 100%, while the other three test cases have values equal to 40% and 60%.

Table 10. Mutation score of test cases.

Test Case ID	Mutation Score (%)
T1	40
T2	60
T3	100
T4	40

The proposed algorithm for mutation analysis has been experimented on different SUTs and executed in Dev C++ IDE-5.11 several times to obtain more accurate results. As a result, the complexity of the proposed GA-based approach for mutation analysis is equal to $O(gen \times pop)$, where gen is the maximum number of generations and pop is the number of populations.

The GA process has been executed several times with the same population scale, and the optimized test data are fetched. As aforementioned, Table 10 showed that T3 is the only test case that achieves the highest mutation score or full mutant coverage. The following Figure 3 illustrates the mutation score of each test case for the SUT GCD.

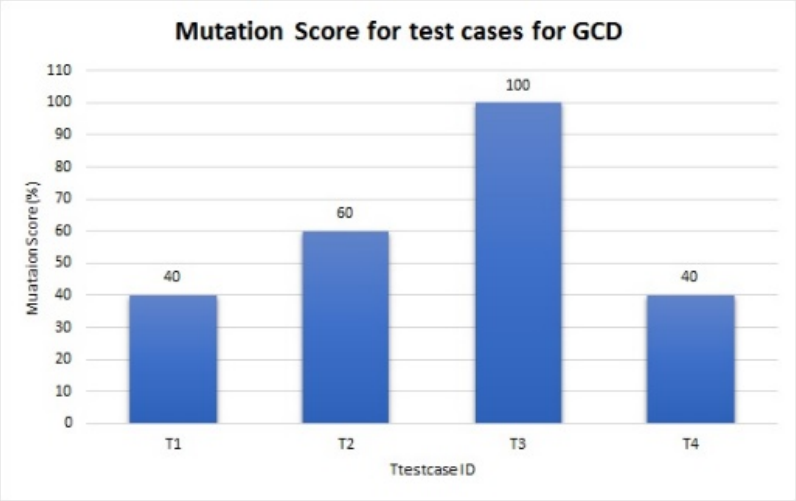

Figure 3. Mutation score of test cases for GCD.

The same process has been applied to other benchmark programs listed in Table 11. This table records the test case id for each program name and the corresponding mutation score. From recorded mutation scores, it is observed that duplicate test cases have the same mutation score for all the System Under Tests (SUTs). So, to remove the duplicates, an elitism operation has to be implemented in the GA method.

Table 11. Highest mutation score of test cases for different SUTs.

Program Name	Test Case ID	Mutation Score (%)
P#1: TCP	T4, T5	85.71
P#2: Max three	T3	100
P#3: Evaluate X^Y	T2	100
P#4: Evaluate X/Y	T2, T5	100
P#5: GCD	T3	100
P#6: QES	T3, T4	100
P#7: Calculate Days	T4, T5, T11, T15, T18	71.42
P#8: Binary Search Tree	T3, T5, T8, T10, T11, T15, T16	85.35
P#9: Calculator	T3, T6, T8	95
P#10: Accept Saving	T4, T7, T8	100

After identifying the test cases with the highest mutation score, it can be clearly seen that duplicate test cases achieve the same mutation score. So, through the elitism process, the optimized test case can be found. The optimized test cases with the highest mutation score are presented in the following Table 12.

Table 12. Optimized test cases with the highest mutation score for different SUTs.

Program Name	Test Case ID	Mutation Score (%)
P#1: TCP	T4	85.71
P#2: Max three	T3	100
P#3: Evaluate X^Y	T2	100
P#4: Evaluate X/Y	T2	100
P#5: GCD	T3	100
P#6: QES	T3	100
P#7: Calculate Days	T4	71.42
P#8: Binary Search Tree	T3	85.35
P#9: Calculator	T3	95
P#10: Accept Saving	T4	100
	Average MS	93.748

The following Figure 4 depicts the optimized test cases with the highest mutation score for different SUTs.

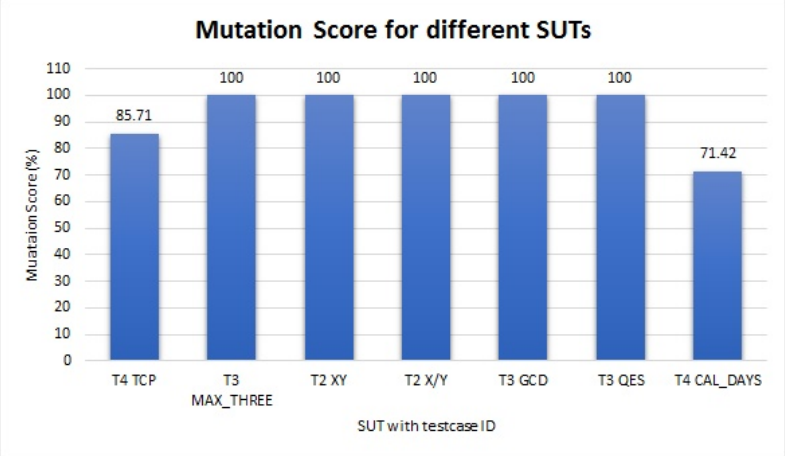

Figure 4. Optimized test cases with the highest mutation score.

4.3. Comparison with Related Work

Mutation analysis has been identified as a powerful testing method for revealing faults in SUTs. In literature, as aforementioned, several similar researches exist for the mutation

analysis via path coverage testing strategies. Specifically, the authors in Refs. [24,35,36] have developed different techniques to cover maximum mutation coverage using the path aware approach. Specifically, authors have utilized different mutation operators to kill the active mutants present in the SUT. More to the point, the proposed approach also takes the path coverage test data as input for achieving the maximum mutation score.

However, we can only compare our work with the research proposed in Ref. [35] on the same domain in terms of mutation operators used to add mutants. Table 13 presents a comparative study of this previous related work with the proposed approach. Authors in Ref. [35] have taken different academic and industrial-based programs with various operators, and their proposed HGA method achieves a mutation score equal to 70%. On the other hand, our proposed GA-based approach achieves a mutation score equal to 93.74%, whereas the proposed RGA-MS method is implemented in several academic programs.

Table 13. Comparison with related work.

Algorithm	Programs	Operators Taken	Mutation Score (%)
HGA [35]	Academic Industrial based Programs	AOR, LOR, SOR, LOD, ASR	70
RGA-MS	Academic Programs	AOR, LOR, ROR, NOR, UOI, CDL, ABS	93.74

4.4. Threats to Validity

The evaluation of mutation scores in terms of test cases is implemented with the use of the path coverage-based test data. Therefore, it does not necessarily mean that the results can be extrapolated to the other coverage-based test data.

The proposed method is tested on some benchmark programs written in C/C++ language, and the target paths are found from CFG, which is independent of any programming language. The programs used for the experiments are small-scale as the flow graphs are hand coded. So, the approach requires tools that support the task of automatic flow graphs and path generation.

5. Conclusions and Future Work

The automatic test data generation regarding white box testing is considered vital in accomplishing both qualitative and reliable software. Although mutation testing is computationally costly, thus it can improve the test data quality in terms of higher mutation scores. In addition, it can enhance the reliability of the software by killing the active mutants present in the source code of a SUT.

In this paper, a Search-Based Software Engineering algorithm, which can effectively identify the maximum mutation score by considering all the corresponding test cases generated through path coverage-based testing, is proposed. Initially, path coverage-based test cases are produced. Using different mutation operators, the proposed algorithm aims to find the mutation score for every test case. Then, a real-coded Genetic Algorithm is utilized, automatically generating test data to cover multiple paths simultaneously and further exercising the test data to achieve maximum mutation score. This proposed Real coded Genetic Algorithm for maximizing the Mutation Score is called RGA-MS.

The proposed RGA-MS-based method can generate test data for both path testing and mutation testing. Real coded GA is applied to generate an optimized test suite that achieves maximum path coverage, and further, the suite is exercised to kill the mutants present in the SUT. The proposed method can achieve optimized test data with the highest mutation score.

Regarding future work, we think more tools could be developed towards having a standardized mutant dataset, as this will increase the research interest in other areas of mutation testing. Furthermore, more mutation operators like SDL, VDL, ODL, EHF, OBAF, NPDF, etc., should be used to implement the proposed technique further, and

the framework could be further developed into an android application to ease access and increase the use of the model. Lastly, we could implement an additional number of experiments to ensure that our proposed method can be utilized in different kinds of programs.

Author Contributions: Conceptualization: D.B.M., B.A. and D.R.; methodology: D.B.M., B.A. and D.R.; software: D.B.M., B.A. and D.R.; writing—original draft preparation: D.B.M., B.A., D.R., V.C.G. and A.K.; writing—review and editing: D.B.M., B.A., D.R., V.C.G. and A.K.; supervision: B.A., V.C.G. and A.K.; project administration: B.A. and V.C.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Ahmed, A.A.; Shaheen, M.; Kosba, E. Software Testing Suite Prioritization Using Multi-Criteria Fitness Function. In Proceedings of the 22nd International Conference on Computer Theory and Applications (ICCTA), Alexandria, Egypt, 13–15 October 2012; pp. 160–166.
2. Huang, M.; Zhang, C.; Liang, X. Software Test Cases Generation based on Improved Particle Swarm Optimization. In Proceedings of the 2nd International Conference on Information Technology and Electronic Commerce, Dalian, China, 20–21 December 2014; pp. 52–55.
3. Harman, M.; Mansouri, S.A.; Zhang, Y. *Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications*; Technical Report TR-09-03; Department of Computer Science, King's College London: London, UK, 2009.
4. 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), St. Louis, MO, USA, 15–21 May 2005; pp. 402–411.
5. 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), Hong Kong, China, 16–21 November 2014; pp. 654–665.
6. Fraser, G.; Zeller, A. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Trans. Softw. Eng.* **2012**, *38*, 278–292. [CrossRef]
7. Ma, Y.; Offutt, J.; Kwon, Y.R. MuJava: A Mutation System for Java. In Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China, 20–28 May 2006; pp. 827–830.
8. 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]
9. Hamlet, R.G. Testing Programs with the Aid of a Compiler. *IEEE Trans. Softw. Eng.* **1977**, *3*, 279–290. [CrossRef]
10. Jia, Y.; Harman, M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* **2011**, *37*, 649–678. [CrossRef]
11. Offutt, A.J.; Untch, R.H. Mutation 2000: Uniting the orthogonal. In *Mutation Testing for the New Century*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 34–44.
12. Zhang, Y.; Gong, D. Generating Test Data for Both Paths Coverage and Faults Detection Using Genetic Algorithms: Multi-Path Case. *Front. Comput. Sci.* **2014**, *8*, 726–740. [CrossRef]
13. Mishra, D.B.; Acharya, A.A.; Acharya, S. White box testing using genetic algorithm—An extensive study. In *A Journey Towards Bio-inspired Techniques in Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 167–187.
14. Mishra, D.B.; Mishra, R.; Acharya, A.A.; Das, K.N. Test Data Generation for Mutation Testing Using Genetic Algorithm. In Proceedings of the Soft Computing for Problem Solving (SocProS), Bhubaneswar, India, 23–24 December 2017; Volume 817, pp. 857–867.
15. Wang, G.; Deb, S.; Cui, Z. Monarch Butterfly Optimization. *Neural Comput. Appl.* **2019**, *31*, 1995–2014. [CrossRef]
16. Li, S.; Chen, H.; Wang, M.; Heidari, A.A.; Mirjalili, S. Slime mould algorithm: A new method for stochastic optimization. *Future Gener. Comput. Syst.* **2020**, *111*, 300–323. [CrossRef]
17. Wang, G. Moth search algorithm: A bio-inspired metaheuristic algorithm for global optimization problems. *Memetic Comput.* **2018**, *10*, 151–164. [CrossRef]
18. Tu, J.; Chen, H.; Wang, M.; Gandomi, A.H. The Colony Predation Algorithm. *J. Bionic Eng.* **2021**, *18*, 674–710. [CrossRef]
19. Honeycutt, R.L. Stochastic Runge-Kutta algorithms. I. White noise. *Phys. Rev. A* **1992**, *45*, 600. [CrossRef]
20. Heidari, A.A.; Mirjalili, S.; Faris, H.; Aljarah, I.; Mafarja, M.M.; Chen, H. Harris hawks optimization: Algorithm and applications. *Future Gener. Comput. Syst.* **2019**, *97*, 849–872. [CrossRef]

21. Mishra, D.B.; Mishra, R.; Das, K.N.; Acharya, A.A. A Systematic Review of Software Testing Using Evolutionary Techniques. In Proceedings of the 6th International Conference on Soft Computing for Problem Solving (SocProS), Patiala, India, 23–24 December 2016; Volume 546, pp. 174–184.
22. Silva, R.A.; do Rocio Senger de Souza, S.; de Souza, P.S.L. A Systematic Review on Search based Mutation Testing. *Inf. Softw. Technol.* **2017**, *81*, 19–35. [CrossRef]
23. Dave, M.; Agrawal, R. Search based Techniques and Mutation Analysis in Automatic Test Case Generation: A Survey. In Proceedings of the IEEE International Advance Computing Conference (IACC), Banglore, India, 12–13 June 2015; pp. 795–799.
24. 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]
25. 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 (ICCSCE), Penang, Malaysia, 27–28 August 2012; pp. 119–123.
26. Souza, F.C.M.; Papadakis, M.; Durelli, V.H.S.; Delamaro, M.E. Test Data Generation Techniques for Mutation Testing: A Systematic Mapping. In Proceedings of the XVII Iberoamerican Conference on Software Engineering (CIbSE), Pucon, Chile, 23–25 April 2014; pp. 419–432.
27. Jatana, N.; Rani, S.; Suri, B. State of Art in the Field of Search-based Mutation Testing. In Proceedings of the 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), Noida, India, 2–4 September 2015; pp. 1–6.
28. Mishra, K.K.; Tiwari, S.; Kumar, A.; Misra, A.K. An Approach for Mutation Testing using Elitist Genetic Algorithm. In Proceedings of the 3rd International Conference on Computer Science and Information Technology, Chengdu, China, 9–11 July 2010; Volume 5, pp. 426–429.
29. 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]
30. Harman, M.; Jones, B.F. Search-based Software Engineering. *Inf. Softw. Technol.* **2001**, *43*, 833–839. [CrossRef]
31. Bottaci, L. A Genetic Algorithm Fitness Function for Mutation Testing. In Proceedings of the 8th Workshop on Software Engineering using Metaheuristic INovative Algorithms (SEMINAL), Toronto, ON, Canada, 12–19 May 2001.
32. Mathur, A.P. *Foundations of Software Testing*; Pearson Education India: Noida, India, 2013.
33. Soni, N.; Kumar, T. Study of Various Mutation Operators in Genetic Algorithms. *Int. J. Comput. Sci. Inf. Technol.* **2014**, *5*, 4519–4521.
34. Jena, T.; Mohanty, J.R. Disaster Recovery Services in Intercloud Using Genetic Algorithm Load Balancer. *Int. J. Electr. Comput. Eng.* **2016**, *6*, 1828–1838.
35. Mala, D.J.; Mohan, V. Quality Improvement and Optimization of Test Cases: A Hybrid Genetic Algorithm based Approach. *ACM Softw. Eng. Notes* **2010**, *35*, 1–14. [CrossRef]
36. Gong, D.; Zhang, Y. Generating Test Data for Both Paths Coverage and Faults Detection Using Genetic Algorithms. *Front. Comput. Sci.* **2013**, *7*, 822–837. [CrossRef]
37. Ahmed, M.A.; Hermadi, I. GA-based Multiple Paths Test Data Generator. *Comput. Oper. Res.* **2008**, *35*, 3107–3124. [CrossRef]
38. Rani, S.; Suri, B. An Approach for Test Data Generation Based on Genetic Algorithm and Delete Mutation Operators. In Proceedings of the 2nd International Conference on Advances in Computing and Communication Engineering (ICACCE), Rohtak, India, 1–2 May 2015; pp. 714–718.
39. Khan, R.; Amjad, M. Automatic Test Case Generation for Unit Software Testing Using Genetic Algorithm and Mutation Analysis. In Proceedings of the IEEE UP Section Conference on Electrical Computer and Electronics (UPCON), Allahabad, India, 4–6 December 2015; pp. 1–5.
40. Khan, R.; Amjad, M. Optimize the Software Testing Efficiency using Genetic Algorithm and Mutation Analysis. In Proceedings of the 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 16–18 March 2016; pp. 1174–1176.
41. Masud, M.; Nayak, A.; Zaman, M.; Bansal, N. Strategy for Mutation Testing using Genetic Algorithms. In Proceedings of the Canadian Conference on Electrical and Computer Engineering, Saskatoon, SK, Canada, 1–4 May 2005; pp. 1049–1052.
42. Khan, R.; Amjad, M.; Srivastava, A.K. Generation of Automatic Test Cases with Mutation Analysis and Hybrid Genetic Algorithm. In Proceedings of the 3rd IEEE International Conference on Computational Intelligence and Communication Technology (IEEE-CICT), Ghaziabad, India, 9–10 February 2017; pp. 1–4.
43. Wang, R.; Sato, Y.; Liu, S. Mutated Specification-Based Test Data Generation with a Genetic Algorithm. *Mathematics* **2021**, *9*, 331. [CrossRef]
44. Bashir, M.B.; Nadeem, A. Improved Genetic Algorithm to Reduce Mutation Testing Cost. *IEEE Access* **2017**, *5*, 3657–3674. [CrossRef]
45. Doss, S.; Paranthaman, J.; Gopalakrishnan, S.; Duraisamy, A.; Pal, S.; Duraisamy, B.; Van, C.L.; Le, D.N. Memetic Optimization with Cryptographic Encryption for Secure Medical Data Transmission in IoT-Based Distributed Systems. *Comput. Mater. Contin.* **2021**, *66*, 1577–1594. [CrossRef]
46. Guizzo, G.; Sarro, F.; Krinke, J.; Vergilio, S.R. Sentinel: A Hyper-Heuristic for the Generation of Mutant Reduction Strategies. *IEEE Trans. Softw. Eng.* **2022**, *48*, 803–818. [CrossRef]

47. Le, D.N.; Nguyen, G.N.; Garg, H.; Huynh, Q.T.; Bao, T.N.; Tuan, N.N. Optimizing Bidders Selection of Multi-Round Procurement Problem in Software Project Management Using Parallel Max-Min Ant System Algorithm. *Comput. Mater. Contin.* **2021**, *66*, 993–1010. [CrossRef]
48. Trinh, B.N.; Huynh, Q.; Nguyen, X.T.; Nguyen, G.N.; Le, D. A Novel Particle Swarm Optimization Approach to Support Decision-Making in the Multi-Round of an Auction by Game Theory. *Int. J. Comput. Intell. Syst.* **2020**, *13*, 1447–1463.
49. Rani, S.; Suri, B.; Goyal, R. On the Effectiveness of Using Elitist Genetic Algorithm in Mutation Testing. *Symmetry* **2019**, *11*, 1145. [CrossRef]
50. Molinero, C.; Núñez, M.; Andrés, C. Combining genetic algorithms and mutation testing to generate test sequences. In Proceedings of the 10th International Work-Conference on Artificial Neural Networks (IWANN): Part I: Bio-Inspired Systems: Computational and Ambient Intelligence, Salamanca, Spain, 10–12 June 2009; Volume 5517, pp. 343–350.
51. Kusharki, M.B.; Misra, S.; Muhammad-Bello, B.L.; Salihu, I.A.; Suri, B. Automatic Classification of Equivalent Mutants in Mutation Testing of Android Applications. *Symmetry* **2022**, *14*, 820. [CrossRef]