Computing and Informatics, Vol. 43, 2024, 709–734, doi: 10.31577/cai_2024_3_709

LEVERAGING GENETIC ALGORITHMS FOR EFFICIENT SEARCH-BASED HIGHER ORDER MUTATION TESTING

Serhat Uzunbayir, Kaan Kurtel

Department of Software Engineering
Izmir University of Economics
Sakarya Cd. No. 156
35330, Balçova, İzmir, Türkiye
e-mail: {uzunbayir.serhat, kaan.kurtel}@ieu.edu.tr

Abstract. Higher order mutation testing is a type of white-box testing in which the source code is changed repeatedly using two or more mutation operators to generate mutated programs. The objective of this procedure is to improve the design and execution phases of testing by allowing testers to automatically evaluate their test cases. However, generating higher order mutants is challenging due to the large number of mutants needed and the complexity of the mutation search space. To address this challenge, the problem is modeled as a search problem. The purpose of this study is to propose a genetic algorithm-based search technique for mutation testing. The expected outcome is a reduction in the number of equivalent high order mutants produced, leading to a minimum number of mutant sets that produce an adequate mutation score. The experiments were carried out and the results were compared with a random search algorithm and four different versions of the proposed genetic algorithm which use different selection methods: roulette wheel, tournament, rank, and truncation selection. The results indicate that the number of equivalent mutants and the execution cost can be reduced using the proposed genetic algorithm with respect to the selection method.

Keywords: Search-based mutation testing, higher order mutation testing, equivalent mutants, genetic algorithms, selection methods

1 INTRODUCTION

Software testing is a vital step in the software development process to create reliable products. To ensure success, the testing team must create an effective test suite that thoroughly examines every aspect of the program to prevent defects. However, the increased number of functional requirements and program specifications can make testing a time-consuming part of the software development process. Unit testing, a white-box testing approach, involves evaluating individual components or software units to confirm that each component operates as intended.

An efficient method to evaluate product quality early on is by utilizing mutation testing for unit tests, as proposed by DeMillo et al. [1]. This is a white-box testing approach that focuses on identifying faults in the test cases by altering the original program through a set of predefined rules on its syntax structure. This alteration process, referred to as mutation, involves making small modifications to the code that mimic potential mistakes made by programmers. This method is useful in evaluating the program and highlighting the shortcomings of the test cases in the test suite, and is therefore sometimes referred to as mutation analysis rather than testing.

Mutation analysis has been researched for many years in software engineering studies [2] and applied to industry applications [3, 4, 5]. It is used to tackle different software testing problems such as measuring test suite quality [6, 7], evaluating and minimizing the number of test cases in a test suite [8, 9, 10], locating faults in a program [11, 12], evaluating and comparing test coverage criteria [13], automating mutations of web applications [14], and software programs for popular programming languages including Java [15] and C# [16].

Mutants generated by the mutation testing process can be grouped into two categories: first order mutants and higher order mutants. First order mutants are generated through applying mutation operators once during the mutant generation process of mutation analysis, and higher order mutants through applying mutation operators multiple times. This type of testing is called higher order mutation testing [17]. The aim of higher order mutation is to demonstrate more complex, more realistic programming faults, and therefore improving mutation analysis and reducing the number of equivalent mutants, i.e., mutants that output the same result as the original program and cannot be killed in the mutation analysis process, relative to first order mutation. A study on higher order mutation claims that mutants are unlikely to be equivalent mutants [18], and the overall number of mutants can be reduced [19]. Regardless of the extensive studies on higher order mutation testing, finding useful mutants is still a problem, since the search space is large by nature. Therefore, there is an opportunity to optimize the search space in the context of mutation testing using heuristic approaches. The use of artificial intelligence technologies to eliminate the problems of higher order mutation and increase the test quality will be helpful in producing faster and more workable test suites.

Search-based mutation testing involves the use of meta-heuristic optimization techniques. Meta-heuristics represent a category of algorithms specifically designed

to tackle complex optimization challenges, which are typically resistant to conventional optimization methods [20]. Such problems are characterized by their large-scale, non-linear nature, and they either lack a precise solution or such a solution is impractical to determine within a reasonable time. There are many traditional methods such as ant colony optimization, genetic algorithms, simulated annealing, or particle swarm optimization [21]. Some newer approaches are giant trevally optimizer [22], elephant clan optimization [23], and reptile search algorithm [24]. These methods offer sufficiently effective solutions within a manageable period of time. Applying meta-heuristics to search-based mutation testing can be achieved by defining fitness functions [25], generating test data [26], and incorporating mutation operators [27]. Meta-heuristics are capable of solving complex problems with the aim of finding an optimal solution from a large search space [28]. However, one of the main limitations of higher order mutation is the size of its search space. To overcome this problem, recent research suggests that utilizing meta-heuristic approaches in mutation testing can effectively reduce the number of mutants generated [29].

This study mainly aims to analyze the problems of higher order mutation testing by proposing a meta-heuristic approach using genetic algorithms to solve them. The main contributions of this study can be summarized as follows:

- A meta-heuristic approach employing genetic algorithms to address the challenges of higher order mutation testing is proposed. The proposed method includes four different variants of the genetic algorithm, each incorporating a different selection method (roulette wheel, tournament, rank, and truncation selection). This variety allows for a comprehensive evaluation of different strategies in the context of higher order mutation generation.
- A comparative analysis between four variants and a random search algorithm is provided to validate and verify the effectiveness of the solution.
- The algorithm has been implemented in the C# programming language and has been tested in seven different test programs, each accompanied by a specially designed test suite. Given that each programming language has its unique mutation testing tool, this study specifically experimented with a C# mutation testing tool named VisualMutator.
- The findings from the study show that the genetic algorithm-based approach with respect to the selection method outperforms the random search algorithm in key performance metrics, such as reducing the occurrence of equivalent mutants and achieving this in a shorter execution time, thereby enhancing computational efficiency.

The remainder of the paper is organized as follows. Section 2 presents related work and background research on the topic. Section 3 discusses the proposed method and presents the details of the solution. In Section 4, experiments details, research questions, and datasets are presented. Section 5 presents and discusses the results. In Section 6, the conclusions and future work are discussed.

2 RELATED WORK

This section presents the concepts and terms of mutation testing outlined in the study. Section 2.1 delves into the concept of first order mutants and provides examples of equivalent mutant terms. Section 2.2 focuses on the higher order mutation testing. Section 2.3 investigates search-based mutation testing and examines significant studies related to it. Section 2.4 presents a review of the literature on this topic.

2.1 First Order and Equivalent Mutant

The fundamental method in mutation testing is to create a replica of the original program, apply mutation operators to introduce artificial errors and call it a mutant, and then use a test suite to try to detect these errors in the faulty version of the program. An example of this can be seen in Table 1, where the original program is replicated and a mutant is created by changing the operator "greater than or equal to (>=)" to the "less than (<)" operator.

When a mutant is created by applying a mutation operator only once to an original program, it is called a first order mutant. Table 1 shows a first order mutant.

Algorithm 1 Original Program	**Algorithm 2** First Order Mutant
1: *Read a number*	1: *Read a number*
2: **if** *number* >= 30 **then**	2: **if** *number* < 30 **then**
3: *Print "GREEN"*	3: *Print "GREEN"*
4: **else**	4: **else**
5: *Print "BLUE"*	5: *Print "BLUE"*
6: **end if**	6: **end if**

Table 1. Original program and its first order mutant

The equivalent mutant outputs the same result as the original program. This situation leads to the problem of mutant detection during mutation analysis. Fundamentally, the test suite cannot detect an equivalent mutant due to the injected fault, and it cannot be killed automatically; a manual intervention is needed to detect and kill it. Studies have shown that these mutants are not very rare; in fact, they are actually generated quite often. Sometimes more than 50 % of the overall mutants can be equivalent at the end of the test [30]. Table 2 shows an example of an equivalent mutant.

The idea of ordering mutation testing is performed by applying more than one fault artificially to an original program. To represent the order of a mutant, the number of inserted faults is used. For example; if two faults are inserted to create a mutant, it becomes a second order mutant; if three faults are inserted, it becomes a third order mutant.

Algorithm 3 Original Program
1: $a = 2$
2: **if** $b == 2$ **then**
3: *Print "b"*
4: $b = a + b$
5: **end if**

Algorithm 4 Equivalent Mutant
1: $a = 2$
2: **if** $b == 2$ **then**
3: *Print "b"*
4: $b = a * b$
5: **end if**

Table 2. Original program and an equivalent mutant

2.2 Higher Order Mutation Testing

Second order mutation testing was first proposed in 1992 by Offutt [31] while studying the coupling effect of mutation analysis. He claimed that complex faults can be detected using second ordering and showed that fewer second order mutants survived after experiments. This phenomenon was then investigated further by Polo et al. [32], Madeyski [33], and Mateo et al. [34], who showed that the number or equivalent mutants can be reduced by combining first order mutants in order to create second order mutants, the number of mutants can be reduced without changing the test suite quality. However, one drawback from these studies suggested that some second order mutants become more difficult to kill during mutation analysis.

Higher order mutation testing was proposed in 2009 by Jia and Harman [17]. Instead of injecting a single fault into a mutant, two or more faults are injected. Table 3 shows an example of a higher order mutant with two faults, i.e. a second order mutant.

Algorithm 5 Original Program
1: *Read A and B*
2: *multiplication* $= 1$
3: **if** $A > B$ **then**
4: *multiplication* $= A * B$
5: *result* $=$ *multiplication*$/2$
6: **end if**

Algorithm 6 Higher Order Mutant
1: *Read A and B*
2: *multiplication* $= 1$
3: **if** $A > B$ **then**
4: *multiplication* $= A + B$
5: *result* $=$ *multiplication* $* 2$
6: **end if**

Table 3. Original program and its higher order mutant

In practice, a considerable number of higher order mutants are useless. Since a test that kills a first order mutant which is generated by one of the mutation operators of a higher order mutant, can also kill that higher order mutant due to coupling effect hypotheses of mutation testing. However, Jia and Harman [17] underline that there are some types of higher order mutant that are useful; there exist several classes of higher order mutants that are potentially valuable, because

they show strange characteristics. These mutants are named as strongly subsuming higher order mutants, in which the combining mutants can make higher order mutants more difficult to kill. Therefore, the authors used the three search-based algorithms to generate subsuming higher order mutants. They experimented with ten programs and showed that 67 % of the higher order mutants generated were indeed more difficult to kill than the first order mutants used to generate them.

2.3 Search-Based Mutation Testing

Exhaustive testing is impossible due to the infinite number of test data. Therefore, it cannot be used to test an entire system in practice. In mutation testing, the number of mutants is so large and it is costly to find required amount of mutants that can increase mutation score among those having no effects at all. In the literature, search-based approaches have been applied to several optimization problems [35]. Software test design problems can be formulated as an optimization problem and have been solved using meta-heuristic techniques [36]. To this end, the generation of test cases using search-based testing techniques to find a good solution within the search space can be automated [37].

Search-based mutation testing involves the formulation of test data generation and mutant optimization as optimization problems. Then, meta-heuristic algorithms are applied to solve these problems with full or partial automation. Meta-heuristics are high-level algorithmic frameworks aimed at generating or finding solutions with strategies for heuristic optimization algorithms [38]. Promising solutions can be produced using meta-heuristic algorithms such as genetic algorithms, ant colony optimization, particle swarm optimization, simulated annealing, artificial bee colony, and harmony search combined with mutation testing. Figure 1 shows the categorization of the algorithms and some examples [39].

Bottaci [40] proposed a fitness function and applied it to an optimization algorithm to kill mutants. He suggested that his function was based on reachability, sufficiency, and necessity conditions. Since then, these three conditions have become the foundation of search-based mutation testing.

In spite of having advantages, each meta-heuristics has limitations. Some can be listed as follows:

- Genetic algorithms have premature convergence, slow speed, large iteration times, and low efficiency in test case generation.
- Ant colony optimization is difficult to analyze its theoretical concepts, and there is an uncertainty of time convergence.
- Particle swarm optimization suffers from a low convergence rate and a high chance of being stuck in local optima.
- Simulated annealing has many parameters to be tuned and trade-offs to be considered between result quality and run time of the algorithm.

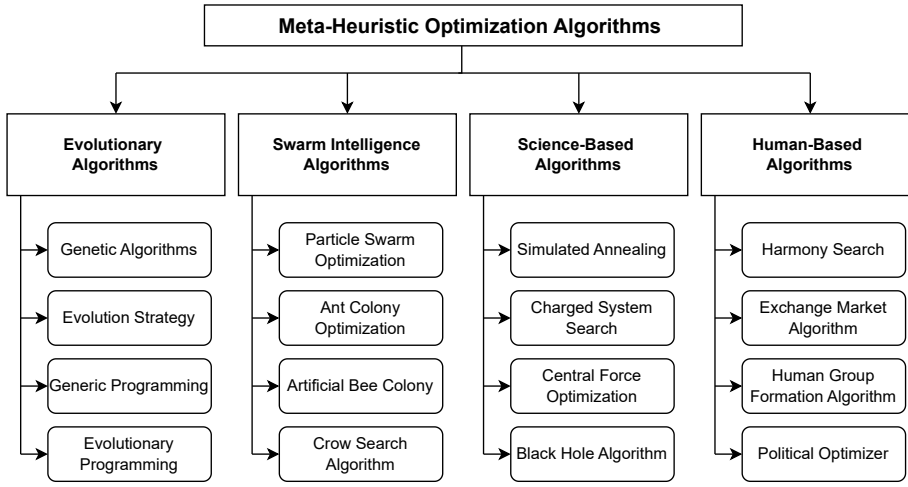

Figure 1. Meta-heuristic optimization algorithms categorization with examples

- Artificial bee colony requires a good balance between exploration and exploitation and is not very predictable due to its stochastic nature.
- Harmony search is sensitive to its parameters and can stuck on local optima especially in complex programs.

There are many studies that tackle the disadvantages of meta-heuristics proposing additional methods to avoid local optimum, better parameter selection mechanisms, or increasing accuracy additions. Therefore, the area of meta-heuristics is always open for further research.

2.4 Literature Review

This section presents a review of related work on higher order mutation, automated test data generation, and search-based mutation testing.

The concept and application of higher order mutation testing are extensively explored in the literature. Jia and Harman [17] introduced higher order mutation and types of higher order mutants, particularly focusing on subsuming mutants which are harder to kill than their constituent mutants. They demonstrated the existence of such mutants in C programs using search-based optimization techniques. This was the first study to propose higher order mutation, paving the way for further experiments especially for test data generation processes. Similarly, Kapoor [41] emphasized the difficulty of killing higher order mutants compared to first order mutants, highlighting the reduced testing cost when a subsuming higher order mutant is present. Although the paper offers theoretical arguments and highlights potential benefits, it does not provide comprehensive empirical validation through

large-scale experiments. Polo et al. [32] proposed a method to reduce mutation costs for second order mutants. They merged first order mutants generated containing one fault to generate second order mutants, halving the number of mutants are reducing and the costs. While the technique showed positive results in both experiments, further research involving industrial software is necessary for its validation. Madeyski et al. [42] conducted a systematic review on the equivalent mutant problem, categorizing solutions into detect, suggest, and avoid approaches. Furthermore, they proposed some second order mutation methods based on the work of Polo et al. Wong et al. [43] and Oh et al. [44] contributed by suggesting methods for identifying strongly subsuming higher order mutants. While Wong et al. proposed a three-step approach including variational search and prioritized search, Oh et al. utilized Causal Program Dependence Analysis for this purpose. The first study faced limitations due to the specific choices of programs, mutation operators, and test cases. Technical constraints necessitated the exclusion of some tests and mutations, which may have slightly impacted the results. The second study, on the other hand, required the implementation of different heuristic designs and a larger dataset for experimentation. Langdon et al. [45] investigated the relationship between mutant syntax and semantics, particularly in mutants difficult to kill. Their approach considered large and small syntax changes and experimented to determine whether the effects are similar. They showed that there are some higher order mutants that are similar to the original program semantically. The research was centered on the C programming language, utilizing traditional mutation operators. It did not encompass object-oriented mutation operators typical of languages such as Java or C#.

Alshraideh and Bottaci [46], and later Alshraideh et al. [47], made significant contributions to automated test data generation. They employed genetic algorithms for generating test data, particularly focusing on branch coverage and string operations in the programs. They also proposed a multiple-population genetic algorithm, which outperformed the single-population version in terms of execution time and search quality. Alshraideh et al.'s method uses acyclic predicate paths in order to reach all branches with various island populations. As a result, the multiple-population algorithm performs better than the single-population version based on execution time and search quality. While superior to single-population algorithms in several performance metrics, its main limitation is inefficiency for simpler programs, where it might target irrelevant code sections. This inefficiency in small programs is balanced by its effectiveness in more complex scenarios, where single-method searches typically struggle. Mala et al. [48] presented a hybrid genetic algorithm with the aim of improving the quality of test cases. The study showed that the mutation score is increased and the total number of test cases can be reduced, also reducing the execution time. The results were compared with a generic genetic algorithm and the implementation of a bacteriologic algorithm. Therefore, the proposed hybrid genetic algorithm produced near-optimal solutions. The study would benefit from additional experimentation with methods such as hill climbing or swarm intelligence to demonstrate the effectiveness of the proposed methods more comprehensively.

The use of search-based techniques in higher order mutation testing has been a focal point of several studies. Omar and Ghosh [49] worked on AspectJ programs and proposed four approaches to automatically generate higher order mutants. They arranged faults based on their parts occurrence, such as base classes or aspect interactions. The four proposed approaches are for aspect-oriented programming faults from a previous work [50]. The study is promising, but there is also a need for more extensive research on different HOM construction methods, particularly with larger programs and extensive test sets. Omar et al. [51] continued with this approach in their next study by introducing three new search techniques to find higher order mutants; guided local search, restricted random search, and restricted enumeration search. Omar et al. [52] later introduced HOMAJ, a tool for higher order mutation in Java and AspectJ, demonstrating its efficacy in creating and executing mutants. The research also suggests an exploration into extending existing techniques to identify equivalent HOMs, which could reduce the manual labor involved in distinguishing non-equivalent subtle HOMs. Derezinska and Halas [53] used Python to propose four algorithms for generating higher order mutants; Between-Operators, FirstTo-Last, Random, and Each-Choice, to generate higher order mutants. As a result, they were able to generate 50 % fewer higher order mutants than the number of first order mutants. The research on Python programs may require for more comprehensive evaluation of equivalent mutants and further exploration of additional mutation operators. A more recent study by Aratesh et al. [54] employed an artificial bee colony optimization algorithm to identify the most fault-prone paths, suggesting this method could reduce the number of overall mutants and the associated costs in popular Java mutation testing tools. Nishta et al. [55] conducted a systematic review of the search-based mutation testing literature, highlighting trends up to 2017. They focused on the application of these techniques primarily in mutant generation and test case optimization. Silva et al. [28] also reviewed search-based mutation testing, particularly involving meta-heuristic approaches. However, both reviews require updates to include newer, more comprehensive methodologies in the field, such as machine learning or other artificial intelligence methods.

3 PROPOSED METHOD

In this section, the justification for selecting the investigated methods, the details of the implementation of the proposed genetic algorithm, and the random search algorithm are discussed.

3.1 Justification for Selected Methods

Higher order mutation testing is potentially useful and provides improved opportunities compared to first order mutation testing. It can simulate more realistic faults that a programmer can make with complex changes and improved test optimization techniques [17].

The proposed method aims to attack the most important problem in higher order mutation testing; the reduction of the number of mutants at mutant generation level.

Higher order mutants are generated by using search-based testing techniques, namely meta-heuristics, by aiming to reduce costs. There are three reasons for selecting this technique:

- There is a large set of mutants. Some of these mutants are useful, but not all. A selection process for good mutants would reduce costs.
- A meta-heuristic search is guided by a fitness function. Instead of applying an exhaustive technique to visit all the solutions, a specific search process can be processed in a clever way, and good solutions can be found.
- There are several related studies in the literature that present promising results for search-based mutation testing.

3.2 A Genetic Algorithm for Higher Order Mutant Generation

Our approach generates higher order mutants by using a genetic algorithm, aiming to find the fittest mutants in the search space. We define chromosome representation and the fitness function and discuss the algorithm details for *higher order mutant generation* in this section.

3.2.1 Chromosome Representation

The algorithm uses a population that includes individuals to represent candidate solutions. These solutions contain chromosomes that act on features. Therefore, a higher order mutant represents a chromosome. It is an array of strings, and each element of it is a one-line from the source code of a higher order mutant. Every single line of a higher order mutant is included in chromosomes. Mutated statements come from their related first order mutants. There can be two or more faults contained in chromosomes to represent higher order mutants. Figure 2 shows a chromosome representation as an array of strings.

3.2.2 Fitness Function

A fitness function is used to evaluate a candidate solution compared to the optimal solution to the problem. In other words, it defines whether a solution is good, bad, or close to both sides. It is one of the essential requirements of a genetic algorithm in order to select a good solution from the search space.

The algorithm starts the selection phase after the first initialization. To perform a selection, the fitness of the mutants is calculated. The result of the fitness can be a value from 0 to 1. If the value is closer to 1, it means that the mutant can be easily killed. Otherwise, the mutant is more difficult to kill and requires more than one test case to kill it. Taking into account TC_n as the n^{th} test case of the test suite

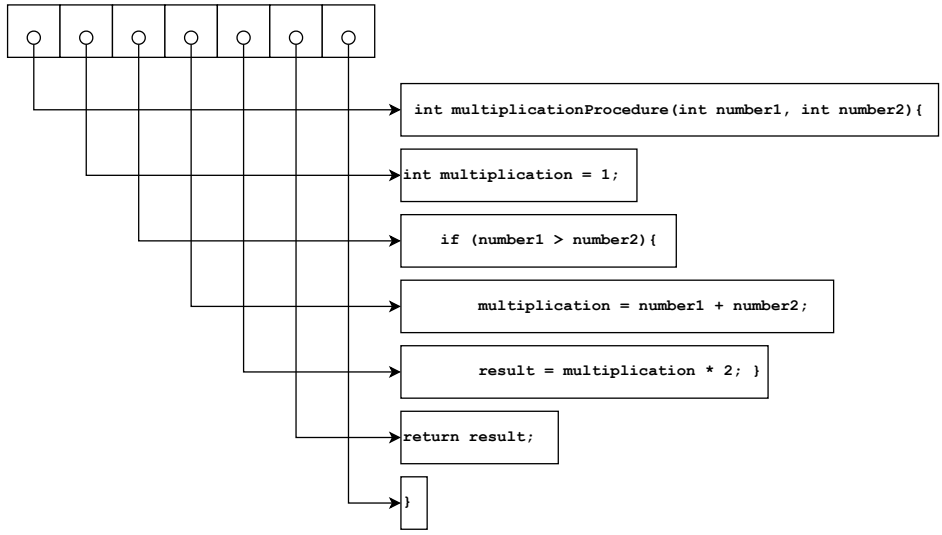

Figure 2. A chromosome representation

that kills mutant c, the total number of test cases is TC_{total} and the fitness value $f(c)$ in a population is calculated as follows:

$$f(c) = \frac{\sum_{i=1}^{n} TC_n}{TC_{total}}.$$

Basically, the total number of test cases that kill a mutant c is divided by the total number of test cases that produce the fitness value of the algorithm.

3.2.3 Initialization

The first population is generated by merging some first order mutants randomly. Two or more first order mutants can be merged, depending on the user's preferences. For example, to create a third order mutant, three first order mutants should be merged.

3.2.4 Selection

In the selection phase, higher order mutants are selected to generate offspring in the later stages. This process serves as the linchpin for identifying candidate solutions for application to subsequent generations. In this study, four prevalent selection mechanisms were selected for experimentation: roulette wheel, tournament, rank, and truncation selection.

Roulette Wheel Selection: The algorithm uses the roulette wheel selection procedure to replace the statements in the first version. In this method, the aim is

to select the best parents among a set of parents to create an offspring. In a regular roulette wheel, the slot sizes are the same; therefore, all slots have the same probability of being selected. In genetic algorithms, the aim is to have a weighted version of a roulette wheel. The individual is more likely to be selected if the fitness is greater than the others. For each chromosome c and the corresponding fitness value $f(c)$ in a population P of size n, $P = \{c_1, c_2, c_3, \ldots, c_n\}$. The probability $p(c_k)$ is calculated using the following formula:

$$p(c_k) = \frac{f(c_k)}{\sum_{i=1}^{n} f(c_l)},$$

where $l = 1, 2, 3, \ldots, n$.

In other words, the sum of fitness is calculated for each individual in the population and the relative fitness value for each is evaluated using the above formula. Roulette is partitioned according to the proportions calculated in the previous step. The roulette wheel is spun n times, where n is the size of the population. Two individuals are selected from the partition when the roulette stops.

Tournament Selection: Tournament selection is used in the second version of the algorithm. It involves selecting a random subset of individuals from the population and conducting a "tournament" among these. The individual with the highest fitness score in this subset is then selected as a parent for the next generation. This process is usually repeated to select multiple parents to create offspring through crossover and mutation operations. In the mutation testing paradigm, each "individual" in the population represents a test suite, and fitness is a measure of the test suite's ability to detect mutants.

Rank Selection: Rank selection is used as the third version of the algorithm. It is a strategy in which individuals are sorted according to their fitness levels and then, within the sorted list, selected according to their rank. Unlike methods such as roulette wheel selection, where the absolute fitness values dictate selection probability, in rank selection, the focus is on the relative standing of an individual within the population. The advantage is to mitigate the risk of premature convergence by focusing on relative, rather than absolute, fitness. This is important in mutation testing where the objective function can be rugged.

Truncation Selection: The last version of the algorithm uses the truncation selection method. It is a selection strategy that truncates the population by discarding the least fit individuals and replicating the fittest ones for the next generation. This method is highly elitist and is best suited for scenarios where rapid convergence to a high-quality solution is desirable. The formula to calculate how many individuals are selected can be given as:

$$\text{Number of Selected Individuals} = \lceil T \times N \rceil,$$

where T is the truncation threshold, a number between 0 and 1. For example, $T = 0.2$ would mean that the highest 20 % of the individuals is selected based

on their mutation score. And N represents the total number of individuals in the population.

3.2.5 Mating

In the mating step, crossover and mutation are applied to selected higher order mutants based on the following criteria:

Crossover: Two selected higher order mutants are selected and combined together to create offspring. Crossover points can be one or more based on user selection. In this study, double point crossover is used. It should not generate a new offspring as a first order mutant, but if it does, another mutation is applied randomly to generate a second order mutant.

Mutation: Mutation is performed on a higher order mutant by randomly adding a first order mutant or removing it. If a first order mutant is to be added, it is selected randomly among all first order mutants. On the other hand, if a first order mutant is to be removed from a higher order mutant, it is selected from first order mutants that generate related higher order mutants. The user can determine the number of first order mutants to be added or removed from higher order mutants. It should not convert a higher order mutant to a first order mutant.

3.2.6 Stopping Condition

Deciding when to stop a genetic algorithm is an important consideration in its implementation. The choice of stopping criteria can influence both the performance and the computational cost of the algorithm. One of the stopping criteria can be a fixed number of generations, which is the simplest approach; the algorithm stops once this number is reached. Another criterion is convergence. The algorithm stops if the population has converged, which means that there is very little variance in the fitness values of the individuals. Convergence might suggest that the algorithm has found a local or global optimum, although this is not necessarily the case. The threshold fitness value can also be a stopping criterion. The algorithm can be set to stop once an individual reaches or exceeds a certain fitness value. This is particularly useful if there is a known solution or a minimum acceptable solution quality.

In this study, the algorithm stops by a predefined parameter by the user, which is the maximum number of different higher order mutants. After the algorithm stops, it returns a list of higher order mutants found. The proposed genetic algorithm is given in Algorithm 7 below. The flow chart of the algorithm can be seen in Figure 3.

3.3 A Random Search Algorithm for Higher Order Mutant Generation

Random search is a non-heuristic-based approach used to explore the search space of possible solutions. Genetic algorithms evolve a population of solutions over several

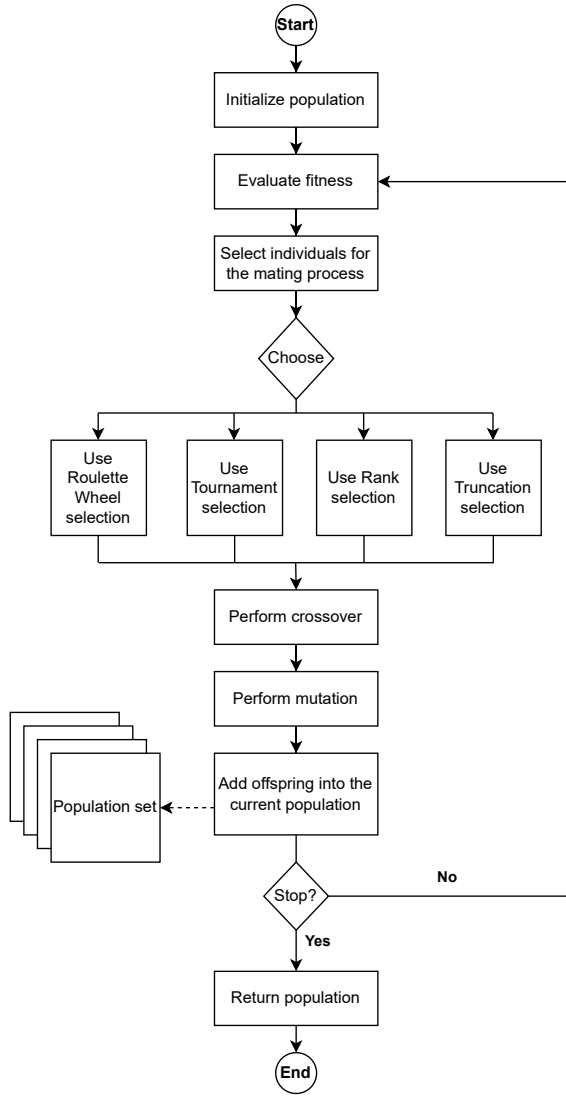

Figure 3. Flow chart of the proposed algorithm

Algorithm 7 Proposed Genetic Algorithm

Require: *firstOrderMutants*, *size*, *selectionMethod*, *cPoint*, *mRatio*

```
1: i = 0
2: higherOrderMutants ← ∅
3: population[i] ← initializePopulation(firstOrderMutants, size)
4: population[i].execute()
5: population[i].calculateFitness()
6: while stoppingCondition ≠ true do
7:    parent ← population[i].select(selectionMethod)
8:    offspring ← parents.crossover(cPoint)
9:    offspring ← parents.mutation(mRatio)
10:   population.add(offspring)
11: end while
12: i ← i + 1
13: population[i].execute()
14: population[i].calculateFitness()
15: higherOrderMutants.add(population[i])
16: return higherOrderMutants
```

generations by employing techniques like selection, crossover, and mutation, while a random search algorithm uniformly samples the search space and directly evaluates the randomly generated solutions. The parameters of selection are random, and there is no intelligence procedure for sampling solutions from the search space.

In this study, the random search approach randomly selects candidate higher order mutants among all higher order mutants individually. The algorithm requires an arbitrarily selected set of first order mutants to generate higher order mutants, and their fitness is evaluated. According to the results, they are selected and stored in a higher order mutants list.

Random search can serve as a baseline to evaluate the performance of more sophisticated algorithms such as the method proposed in this study. Showing that our solution outperforms a simple random search provides empirical evidence that the proposed algorithm is adding value, rather than simply navigating the search space randomly. The pseudocode of the algorithm is given in Algorithm 8.

4 EXPERIMENT DETAILS

In this section, we present the details of the experiments performed in order to evaluate our research goals. Section 4.1 describes the setup of the test environment used in the study. Section 4.2 introduces the research questions of this study. Section 4.3 introduces the subject programs that were used with the experiments. Section 4.4 describes the settings of the genetic algorithm parameters before performing the experiments.

Algorithm 8 Proposed Random Search Algorithm

Require: *firstOrderMutants*

$i = 0$
higherOrderMutants $\leftarrow \emptyset$
while *stoppingCondition* $\neq$ *true* **do**
  randomHigherOrderMutant $\leftarrow$ *makeHigher*(*firstOrderMutants*)
  randomHigherOrderMutant.execute()
  fitness $\leftarrow$ *randomHigherOrderMutant.calculateFitness*()
  if *fitness* $==$ *OK* **then**
    higherOrderMutants $\leftarrow$ *randomHigherOrderMutant*
  end if
  $i \leftarrow i + 1$
  population.add(*offspring*)
end while
return *higherOrderMutants*

4.1 Test Environment Setup

We created our test environment as shown in Figure 4 to create first order mutants based on the following setup and programs:

- The experiments were performed on a desktop computer running Windows 11 operating system with an Intel i7 9700k 2.8 GHz processor.
- In this experiment, we used VisualMutator which is a mutation testing tool for C# that generates, compiles, and runs mutants. We used this tool to generate first order mutants. These first order mutants are taken as inputs to assess test cases.
- To generate and organize test cases in our test suite, we used IntelliTest and SentryOne tools, which are automated test case generators for unit testing in NUnit format. The aim of using both tools is to guarantee full coverage for the subject programs that can kill all generated first order mutants. If, for whatever reason, the mutants are not killed, more test cases are added manually.

After first order mutants for all subject programs are created, the random search algorithm and four variants of the genetic algorithm are executed to create higher order mutants to evaluate the results. Figure 5 is a diagram showing this process.

4.2 Research Questions

In this study, the focus is on addressing the following key research questions (RQs):

RQ1: *Which selection method generates the least number of equivalent higher order mutants?* There are different selection strategies that can be applied during the selection phase of a genetic algorithm. The aim of this question is to find the

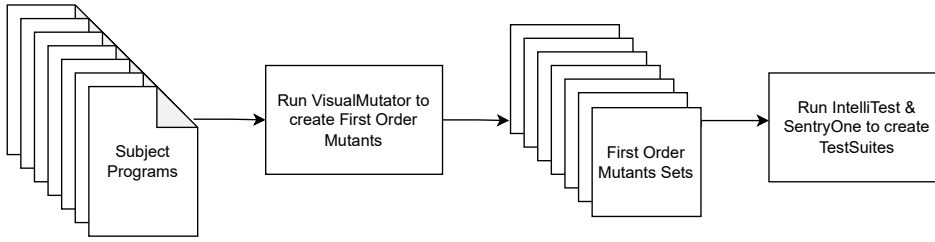

Figure 4. Test environment to create first order mutants

best selection method that can produce a smaller amount of equivalent mutants when generating higher order mutants.

RQ2: *Which selection method has the highest and lowest execution cost?* Considering different selection strategies, the objective of this RQ is to evaluate the fastest and the slowest selection method compared to the others implemented in this study on average.

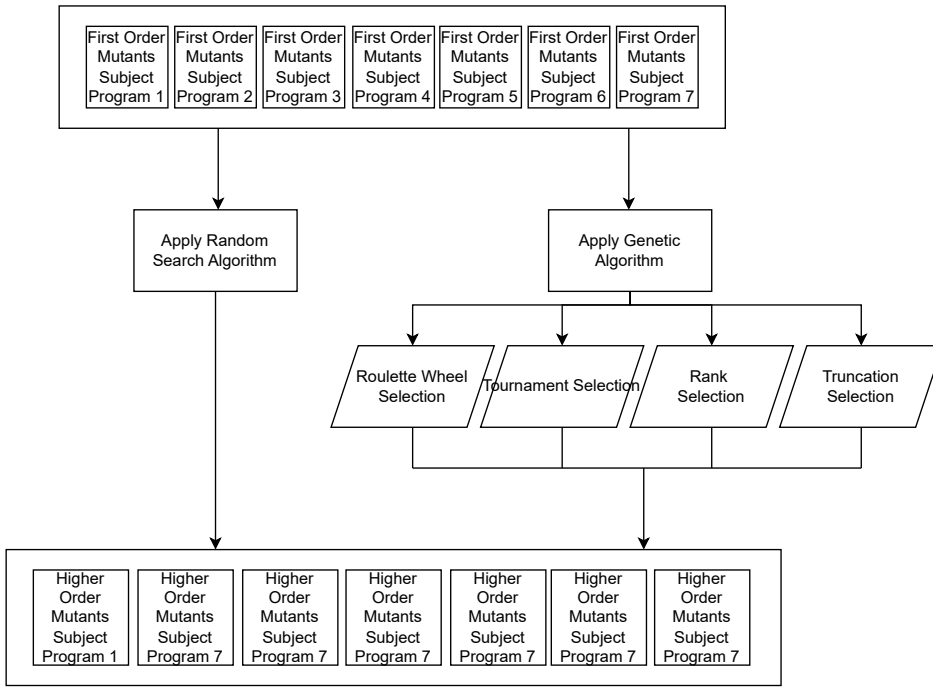

Figure 5. Experiment details to create higher order mutants

RQ3: *What is the percentage of higher order mutants generated for each mutation order?* One of the main drawbacks of higher mutation testing is the amount of mutants generated. The motivation behind this RQ is to test whether the order of mutants is able to lower the number of mutants generated and still produce better results.

4.3 Subject Programs

Subject programs are implemented using C# and are given in Table 4 below including their size, the number of first order mutants, and the number of test cases.

Subject programs are described as follows:

- TriangleType: The program determines whether a given triangle is equilateral, isosceles, or scalene.
- PrintPrimes: The program finds and outputs all prime numbers from a given input.
- CalculateDays: The program calculates the number of days between given inputs and prints the result.
- HashTable: The program creates a hash table and fills it according to a given test file, and then finds values based on their specified keys.
- CocktailSort: The program is an alteration of the bubble-sort algorithm. The idea is to traverse over an input array starting from the beginning and the end as opposed to starting point only.
- MatchPattern: The program finds the occurrences of a given pattern in an expression.
- MoonPhases: The program calculates the moon's phase at the moment and roughly how many days remain in the cycle.

Project Name	Lines of Code	# of First Order Mutants	# of Test Cases
TriangleType	40	102	77
PrintPrimes	49	99	65
CalculateDays	51	85	89
HashTable	95	172	56
CocktailSort	75	204	83
MatchPattern	62	68	71
MoonPhases	165	381	95

Table 4. Subject programs

4.4 Genetic Algorithm Parameter Settings

The selected values for the proposed genetic algorithm parameters are given as follows after experimenting with various values:

- Probability of crossover is 0.7 as this value is a commonly used value in genetic algorithms, often considered a good balance between allowing the algorithm to explore new solutions and exploiting existing good solutions.
- Probability of mutation is 0.07 since a lower probability of mutation allows the algorithm to introduce slight randomness without disturbing the main characteristics of the individuals. This value helps to escape local optima.
- Maximum number of iterations is selected as 1000, because there was no change to the results after this number.
- Mutant ratio is 4 %. During experiments, a mutant ratio 4 % provided a good trade-off between computational efficiency and algorithm performance in terms of the fitness function.
- Chromosome size is equal to the number of lines of code of the subject program, which effectively captures the essential characteristics of the subject program and aids in optimization.

The experiments were performed 30 times on each subject program to validate whether there were changes in each trial or not. After 30 trials, the results show no change and the experiments were terminated.

5 RESULTS

This section presents the results calculated by averaging 30 experiments conducted on each of the seven test programs.

Table 5 answers RQ1: *"Which selection method generates the least number of equivalent higher order mutants?"* It presents the ratio of generated equivalent mutants for each test subject compared with random search and versions of GA using four selection strategies; roulette wheel, tournament selection, rank selection, and truncation selection. Each algorithm generated equivalent mutants; however, the ratio for each is different. Among all trials, the random search was the least effective in terms of the number of equivalent mutants, with an average of 26.8 %. For the proposed genetic algorithm, the truncation selection version has a slightly lower percentage than the rank approach with 18 % and performed the best among others.

MatchPattern had a ratio of 33 % equivalent mutants using random search, while HashTable had a ratio of 11 % equivalent mutants using GA with the roulette wheel selection strategy.

Table 6 answers RQ2: *"Which selection method has the highest and lowest execution cost?"* The aim of this research question is to find out which selection strategy performs faster than the others in terms of execution duration on average. Random search seems to be the least effective performer in this experiment with 13.5 minutes on average. The next worst one is GA with rank selection with 12.9 minutes. GA with tournament selection performed 0.4 minutes slower than GA with roulette

Project Name	Random Search	GA with Roulette Wheel Selection	GA with Tournament Selection	GA with Rank Selection	GA with Truncation Selection
TriangleType	30 %	25 %	25 %	21.2 %	14 %
PrintPrimes	30.3 %	28.4 %	24.1 %	25.5 %	21 %
CalculateDays	23.2 %	15.2 %	18.8 %	15 %	20 %
HashTable	16 %	11 %	15 %	13 %	15 %
CocktailSort	22.5 %	18 %	12.3 %	16.3 %	14.2 %
MatchPattern	33 %	20.7 %	17.6 %	22.5 %	20.4 %
Moon Phases	28.1 %	21.5 %	25 %	20.1 %	18 %
Average	26.8 %	20 %	19.7 %	19.1 %	18 %

Table 5. Ratio of generated equivalent mutants

wheel selection. In general, the fastest algorithm is GA with truncation selection of 9.5 minutes on average.

CalculateDays was the fastest program to execute, with 7.3 minutes using GA with truncation selection, and the slowest experiment recorded was CocktailSort with 16.1 minutes using random search.

Project Name	Random Search	GA with Roulette Wheel Selection	GA with Tournament Selection	GA with Rank Selection	GA with Truncation Selection
TriangleType	12.1	10.2	10.4	13	9.4
PrintPrimes	15.4	11	11.3	13.2	10.2
CalculateDays	9	8.2	8.5	10.5	7.3
HashTable	15.8	10.2	10.6	12.3	10
CocktailSort	16.1	14	15.3	16	12.3
MatchPattern	11.6	8.5	9	10	6.3
MoonPhases	14.5	12.7	12.9	15	11.1
Average	13.5	10.7	11.1	12.9	9.5

Table 6. Execution cost of different selection strategies

Table 7 presents the percentage of higher order mutants from each mutation order including second order, third order, and fourth order mutants to answer RQ3: *"What is the percentage of higher order mutants generated for each mutation order?"*. The motivation for this experiment is to assess whether a higher order is capable of producing a reduced number of mutants that can produce better results. The table clearly shows that when the order is increased, the number of generated mutants is reduced. For this experiment, all subject programs applied with random search and all different variants of the algorithm, and the average results are shown in the table. For instance, the ratio of higher order mutants in the second order mutation using GA with truncation selection is 50.1 %, however, when the fourth order mutation is applied, the average number of higher order mutants is reduced to 14.2 %. Therefore, the proportion of mutant ratio is decreased by 35.9 % in general.

Project Name	Random Search	GA with Roulette Wheel Selection	GA with Tournament Selection	GA with Rank Selection	GA with Truncation Selection
SecondOrder	55.5 %	46.3 %	55.2 %	64.2 %	50.1 %
ThirdOrder	37.6 %	29.4 %	26.1 %	23.1 %	29.2 %
FourthOrder	30.2 %	18.1 %	15.2 %	15.5 %	14.2 %

Table 7. Percentage of the generated higher order mutants from each mutation order

Overall, these research questions and experiments collectively explore the intricacies of mutation testing, focusing on the optimization of selection strategies, resource efficiency, and the structural characteristics of higher order mutants. They reflect a comprehensive approach to improving the effectiveness and efficiency of mutation testing practices.

6 CONCLUSION

In this study, higher order mutation is discussed and a search-based mutation testing method using genetic algorithms is proposed. Four different versions of the genetic algorithm were proposed, each of which involves different selection methods; roulette wheel, tournament, rank, and truncation selection. The proposed solution is applied to the first order mutants to create higher order mutants. The goal is to address the equivalent mutant problem, and the results show that the number of equivalent mutants can be reduced using the proposed method when higher order mutants are formed.

We implemented our proposed algorithm using the C# programming language. To rigorously assess the efficacy and robustness of our approach, we selected seven different test programs, each equipped with relative test suites specifically designed to target and kill higher order mutants. These test environments provided a comprehensive landscape on which to evaluate the performance of the proposed algorithm.

To further substantiate the merits of our method, we performed a comparative analysis against a random search algorithm. Empirical findings convincingly demonstrated that our genetic algorithm-based approach outperformed the random search algorithm in key performance metrics. Specifically, the genetic algorithm not only reduced the occurrence of equivalent mutants, but also accomplished this task in a shorter execution time, thus enhancing computational efficiency.

In future work, we identify several promising directions. The application of alternative selection strategies and different search-based optimization techniques, such as ant colony optimization or particle swarm optimization, can be applied to this study. Comparative analyses under the same test conditions could yield insightful data, enabling us to fine-tune our method and perhaps discover a universally more effective approach for dealing with the challenges posed by higher order mutants and the equivalent mutant problem.

REFERENCES

[1] DeMillo, R. A.—Lipton, R. J.—Sayward, F. G.: Hints on Test Data Selection: Help for the Practicing Programmer. Computer, Vol. 11, 1978, No. 4, pp. 34–41, doi: 10.1109/C-M.1978.218136.

[2] Usaola, M. P.—Mateo, P. R.: Mutation Testing Cost Reduction Techniques: A Survey. IEEE Software, Vol. 27, 2010, No. 3, pp. 80–86, doi: 10.1109/MS.2010.79.

[3] Jia, Y.—Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, Vol. 37, 2011, No. 5, pp. 649–678, doi: 10.1109/TSE.2010.62.

[4] Papadakis, M.—Kintis, M.—Zhang, J.—Jia, Y.—Le Traon, Y.—Harman, M.: Chapter Six – Mutation Testing Advances: An Analysis and Survey. In: Memon, A. M. (Ed.): Advances in Computers. Vol. 112, 2019, pp. 275–378, doi: 10.1016/bs.adcom.2018.03.015.

[5] Vercacmmen, S.—Borg, M.—Demeyer, S.: Validation of Mutation Testing in the Safety Critical Industry Through a Pilot Study. 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2023, pp. 334–343, doi: 10.1109/ICSTW58534.2023.00064.

[6] Nayyar, Z.—Rafique, N.—Hashmi, N.—Rashid, N.—Awan, S.: Analyzing Test Case Quality with Mutation Testing Approach. 2015 Science and Information Conference (SAI), 2015, pp. 902–905, doi: 10.1109/SAI.2015.7237249.

[7] Ojdanic, M.—Soremekun, E.—Degiovanni, R.—Papadakis, M.—Le Traon, Y.: Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution. ACM Transactions on Software Engineering and Methodology, Vol. 32, 2023, No. 1, Art. No. 14, doi: 10.1145/3530786.

[8] Noemmer, R.—Haas, R.: An Evaluation of Test Suite Minimization Techniques. In: Winkler, D., Biffl, S., Mendez, D., Bergsmann, J. (Eds.): Software Quality: Quality Intelligence in Software and Systems Engineering (SWQD 2020). Springer, Cham, Lecture Notes in Business Information Processing, Vol. 371, 2020, pp. 51–66, doi: 10.1007/978-3-030-35510-4_4.

[9] Palomo-Lozano, F.—Estero-Botaro, A.—Medina-Bulo, I.—Núñez, M.: Test Suite Minimization for Mutation Testing of WS-BPEL Compositions. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18), 2018, pp. 1427–1434, doi: 10.1145/3205455.3205533.

[10] Jehan, S.—Wotawa, F.: An Empirical Study of Greedy Test Suite Minimization Techniques Using Mutation Coverage. IEEE Access, Vol. 11, 2023, pp. 65427–65442, doi: 10.1109/ACCESS.2023.3289073.

[11] Kim, J.—An, G.—Feldt, R.—Yoo, S.: Learning Test-Mutant Relationship for Accurate Fault Localisation. Information and Software Technology, Vol. 162, 2023, Art. No. 107272, doi: 10.1016/j.infsof.2023.107272.

[12] Pearson, S.—Campos, J.—Just, R.—Fraser, G.—Abreu, R.—Ernst, M. D.—Pang, D.—Keller, B.: Evaluating and Improving Fault Localization. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 609–620, doi: 10.1109/ICSE.2017.62.

[13] Barani, M.—Labiche, Y.—Rollet, A.: On Factors That Impact the Relationship Between Code Coverage and Test Suite Effectiveness: A Survey. 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2023, pp. 381–388, doi: 10.1109/ICSTW58534.2023.00071.

[14] Leotta, M.—Paparella, D.—Ricca, F.: Mutta: A Novel Tool for E2E Web Mutation Testing. Software Quality Journal, Vol. 32, 2024, No. 1, pp. 5–26, doi: 10.1007/s11219-023-09616-6.

[15] Kintis, M.—Papadakis, M.—Papadopoulos, A.—Valvis, E.—Malevris, N.—Le Traon, Y.: How Effective Are Mutation Testing Tools? An Empirical Analysis of Java Mutation Testing Tools with Manual Analysis and Real Faults. Empirical Software Engineering, Vol. 23, 2018, No. 4, pp. 2426–2463, doi: 10.1007/s10664-017-9582-5.

[16] Uzunbayir, S.—Kurtel, K.: An Analysis on Mutation Testing Tools for C# Programming Language. 2019 4th International Conference on Computer Science and Engineering (UBMK), IEEE, 2019, pp. 439–443, doi: 10.1109/UBMK.2019.8907222.

[17] Jia, Y.—Harman, M.: Higher Order Mutation Testing. Information and Software Technology, Vol. 51, 2009, No. 10, pp. 1379–1393, doi: 10.1016/j.infsof.2009.04.016.

[18] Akinde, A. O.: Using Higher Order Mutation for Reducing Equivalent Mutants in Mutation Testing. Asian Journal of Computer Science and Information Technology, Vol. 2, 2012, No. 3, pp. 13–18.

[19] Ghiduk, A. S.—Girgis, M. R.—Shehata, M. H.: Higher Order Mutation Testing: A Systematic Literature Review. Computer Science Review, Vol. 25, 2017, pp. 29–48, doi: 10.1016/j.cosrev.2017.06.001.

[20] Sarhani, M.—Voss, S.—Jovanovic, R.: Initialization of Metaheuristics: Comprehensive Review, Critical Analysis, and Research Directions. International Transactions in Operational Research, Vol. 30, 2023, No. 6, pp. 3361–3397, doi: 10.1111/itor.13237.

[21] Hussain, K.—Mohd Salleh, M. N.—Cheng, S.—Shi, Y.: Metaheuristic Research: A Comprehensive Survey. Artificial Intelligence Review, Vol. 52, 2019, No. 4, pp. 2191–2233, doi: 10.1007/s10462-017-9605-z.

[22] Sadeeq, H. T.—Abdulazeez, A. M.: Giant Trevally Optimizer (GTO): A Novel Metaheuristic Algorithm for Global Optimization and Challenging Engineering Problems. IEEE Access, Vol. 10, 2022, pp. 121615–121640, doi: 10.1109/ACCESS.2022.3223388.

[23] Jafari, M.—Salajegheh, E.—Salajegheh, J.: Elephant Clan Optimization: A Nature-Inspired Metaheuristic Algorithm for the Optimal Design of Structures. Applied Soft Computing, Vol. 113, 2021, Art. No. 107892, doi: 10.1016/j.asoc.2021.107892.

[24] Abualigah, L.—Abd Elaziz, M.—Sumari, P.—Geem, Z. W.—Gandomi, A. H.: Reptile Search Algorithm (RSA): A Nature-Inspired Meta-Heuristic Optimizer. Expert Systems with Applications, Vol. 191, 2022, Art. No. 116158, doi: 10.1016/j.eswa.2021.116158.

[25] Khoshniat, N.—Jamarani, A.—Ahmadzadeh, A.—Haghi Kashani, M.—Mahdipour, E.: Nature-Inspired Metaheuristic Methods in Software Testing. Soft

Computing, Vol. 28, 2024, No. 2, pp. 1503–1544, doi: 10.1007/s00500-023-08382-8.

[26] Rani, S.—Suri, B.: Mutation Based Test Generation Using Search Based Social Group Optimization Approach. Evolutionary Intelligence, Vol. 15, 2022, No. 3, pp. 2105–2114, doi: 10.1007/s12065-021-00618-6.

[27] Jatana, N.—Rani, S.—Suri, B.: State of Art in the Field of Search-Based Mutation Testing. 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), IEEE, 2015, pp. 1–6, doi: 10.1109/ICRITO.2015.7359256.

[28] Silva, R. A.—Senger de Souza, S. d. R.—Lopes de Souza, P. S.: A Systematic Review on Search Based Mutation Testing. Information and Software Technology, Vol. 81, 2017, pp. 19–35, doi: 10.1016/j.infsof.2016.01.017.

[29] Arasteh, B.—Gharehchopogh, F. S.—Gunes, P.—Kiani, F.—Torkamanian-Afshar, M.: A Novel Metaheuristic Based Method for Software Mutation Test Using the Discretized and Modified Forrest Optimization Algorithm. Journal of Electronic Testing, Vol. 39, 2023, No. 3, pp. 347–370, doi: 10.1007/s10836-023-06070-x.

[30] Grün, B. J. M.—Schuler, D.—Zeller, A.: The Impact of Equivalent Mutants. 2009 International Conference on Software Testing, Verification, and Validation Workshops, 2009, pp. 192–199, doi: 10.1109/ICSTW.2009.37.

[31] Offutt, A. J.: Investigations of the Software Testing Coupling Effect. ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 1, 1992, No. 1, pp. 5–20, doi: 10.1145/125489.125473.

[32] Polo, M.—Piattini, M.—García-Rodríguez, I.: Decreasing the Cost of Mutation Testing with Second-Order Mutants. Software Testing, Verification and Reliability, Vol. 19, 2009, No. 2, pp. 111–131, doi: 10.1002/stvr.392.

[33] Madeyski, L.: Impact of Pair Programming on Thoroughness and Fault Detection Effectiveness of Unit Test Suites. Software Process: Improvement and Practice, Vol. 13, 2008, No. 3, pp. 281–295, doi: 10.1002/spip.382.

[34] Mateo, P. R.—Usaola, M. P.—Alemán, J. L. F.: Validating Second-Order Mutation at System Level. IEEE Transactions on Software Engineering, Vol. 39, 2013, No. 4, pp. 570–587, doi: 10.1109/TSE.2012.39.

[35] Mohammadi, A.—Sheikholeslam, F.—Mirjalili, S.: Nature-Inspired Metaheuristic Search Algorithms for Optimizing Benchmark Problems: Inclined Planes System Optimization to State-of-the-Art Methods. Archives of Computational Methods in Engineering, Vol. 30, 2023, No. 1, pp. 331–389, doi: 10.1007/s11831-022-09800-0.

[36] Rahman, N. A. A.: A Review on Search-Based Mutation Testing. Academia of Information Computing Research, Vol. 3, 2022, No. 1.

[37] Srivastava, P. R.—Ramachandran, V.—Kumar, M.—Talukder, G.—Tiwari, V.—Sharma, P.: Generation of Test Data Using Meta Heuristic Approach. TENCON 2008 – 2008 IEEE Region 10 Conference, 2008, pp. 1–6, doi: 10.1109/TENCON.2008.4766707.

[38] Sörensen, K.—Glover, F. W.: Metaheuristics. In: Gass, S. I., Fu, M. C. (Eds.): Encyclopedia of Operations Research and Management Science. Springer, US, Boston,

MA, 2013, pp. 960–970, doi: 10.1007/978-1-4419-1153-7_1167.

[39] Kumar, A.—Bawa, S.: A Comparative Review of Meta-Heuristic Approaches to Optimize the SLA Violation Costs for Dynamic Execution of Cloud Services. Soft Computing, Vol. 24, 2020, No. 6, pp. 3909–3922, doi: 10.1007/s00500-019-04155-4.

[40] Bottaci, L.: A Genetic Algorithm Fitness Function for Mutation Testing. Proceedings of the SEMINALL-Workshop at the 23rd International Conference on Software Engineering, Toronto, Canada, 2001.

[41] Kapoor, S.: Test Case Effectiveness of Higher Order Mutation Testing. International Journal on Computer and Technology Application (IJCTA), Vol. 2, 2011, No. 5, pp. 1206–1211.

[42] Madeyski, L.—Orzeszyna, W.—Torkar, R.—Jozala, M.: Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. IEEE Transactions on Software Engineering, Vol. 40, 2013, No. 1, pp. 23–42, doi: 10.1109/TSE.2013.44.

[43] Wong, C. P.—Meinicke, J.—Chen, L.—Diniz, J. P.—Kästner, C.—Figueiredo, E.: Efficiently Finding Higher-Order Mutants. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), 2020, pp. 1165–1177, doi: 10.1145/3368089.3409713.

[44] Oh, S.—Lee, S.—Yoo, S.: Effectively Sampling Higher Order Mutants Using Causal Effect. 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2021, pp. 19–24, doi: 10.1109/ICSTW52544.2021.00017.

[45] Langdon, W. B.—Harman, M.—Jia, Y.: Efficient Multi-Objective Higher Order Mutation Testing with Genetic Programming. Journal of Systems and Software, Vol. 83, 2010, No. 12, pp. 2416–2430, doi: 10.1016/j.jss.2010.07.027.

[46] Alshraideh, M.—Bottaci, L.: Search-Based Software Test Data Generation for String Data Using Program-Specific Search Operators. Software Testing, Verification and Reliability, Vol. 16, 2006, No. 3, pp. 175–203, doi: 10.1002/stvr.354.

[47] Alshraideh, M.—Mahafzah, B. A.—Al-Sharaeh, S.: A Multiple-Population Genetic Algorithm for Branch Coverage Test Data Generation. Software Quality Journal, Vol. 19, 2011, No. 3, pp. 489–513, doi: 10.1007/s11219-010-9117-4.

[48] Mala, D. J.—Ruby, E.—Mohan, V.: A Hybrid Test Optimization Framework – Coupling Genetic Algorithm with Local Search Technique. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 133–164.

[49] Omar, E.—Ghosh, S.: An Exploratory Study of Higher Order Mutation Testing in Aspect-Oriented Programming. 2012 IEEE 23rd International Symposium on Software Reliability Engineering, 2012, pp. 1–10, doi: 10.1109/ISSRE.2012.6.

[50] Wedyan, F.—Ghosh, S.: On Generating Mutants for AspectJ Programs. Information and Software Technology, Vol. 54, 2012, No. 8, pp. 900–914, doi: 10.1016/j.infsof.2011.12.001.

[51] Omar, E.—Ghosh, S.—Whitley, D.: Constructing Subtle Higher Order Mutants for Java and AspectJ Programs. 2013 IEEE 24th International Symposium

on Software Reliability Engineering (ISSRE), 2013, pp. 340–349, doi: 10.1109/ISSRE.2013.6698887.

[52] Omar, E.—Ghosh, S.—Whitley, D.: Subtle Higher Order Mutants. Information and Software Technology, Vol. 81, 2017, pp. 3–18, doi: 10.1016/j.infsof.2016.01.016.

[53] Derezinska, A.—Halas, K.: Experimental Evaluation of Mutation Testing Approaches to Python Programs. 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, 2014, pp. 156–164, doi: 10.1109/ICSTW.2014.24.

[54] Arasteh, B.—Imanzadeh, P.—Arasteh, K.—Gharehchopogh, F. S.—Zarei, B.: A Source-Code Aware Method for Software Mutation Testing Using Artificial Bee Colony Algorithm. Journal of Electronic Testing, Vol. 38, 2022, No. 3, pp. 289–302, doi: 10.1007/s10836-022-06008-9.

[55] Nishtha, J.—Bharti, S.—Shweta, R.: Systematic Literature Review on Search Based Mutation Testing. e-Informatica Software Engineering Journal (EISEJ), Vol. 11, 2017, No. 1, pp. 59–76, doi: 10.5277/e-Inf170103.

Serhat Uzunbayir received his B.Sc. degree in software engineering (2012), his M.Sc. degree in intelligent computing systems (2015), and his Ph.D. degree in computer engineering (2024) from the Izmir University of Economics. He is currently working as Research Assistant Doctor at the Izmir University of Economics, Software Engineering Department. His primarily research interests are software testing, Big Data, and optimization algorithms.

Kaan Kurtel received his M.Sc. degree from the Computer Engineering Department, Ege University (2005), and his Ph.D. degree in computer science from the Trakya University (2009). His main interests are software verification and validation, software quality, software maintenance, and context-aware systems. He is currently working as Assistant Professor in the Software Engineering Department, at the İzmir University of Economics.