International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

EvoColony: A Hybrid Approach to Search-Based Mutation Test Suite Reduction Using Genetic Algorithm and Ant Colony Optimization

Serhat Uzunbayir[1*], Kaan Kurtel[2]

Submitted: 26/08/2023 **Revised**: 12/10/2023 **Accepted**: 25/10/2023

Abstract: The increasing complexity of software systems requires robust and efficient test suites to ensure software quality. In this context, mutation testing emerges as an invaluable method for evaluating a test suite's the fault detection capability. Traditional approaches to test case generation and evaluation are often inadequate, particularly when applied to mutation testing, which aims to evaluate the quality of a test suite by introducing minor changes or mutations to the code. As software projects increase in scale, there is greater computational cost of employing exhaustive mutation testing techniques, leading to a need for more efficient approaches. Incorporating metaheuristics into the realm of mutation testing offers a synergistic advantage in optimizing test suites for better fault detection. Especially, combining test suite reduction methods with mutation testing produces a more computationally efficient approach compared to more exhaustive ones. This study presents a novel approach, called EvoColony, which combines intelligent search-based algorithms, specifically genetic algorithms and ant colony optimization, to reduce test cases and enhance the effectiveness of the test suit for mutation testing. Integrating both metaheuristic techniques, the research aims to optimize existing test suites, and to improve mutant detection with fewer test cases, thus improving the overall testing quality. The results of experiments conducted were compared with traditional methods, demonstrating the superior effectiveness and efficiency of the proposed hybrid approach. The findings show a significant advancement in test case reduction when using the hybrid algorithm with mutation testing methodologies, and thus ensure the quality of test suites.

Keywords: *software testing, mutation testing, search-based mutation, genetic algorithms, ant colony optimization, metaheuristics*

1. Introduction

Software testing is an essential part in the software development process with the goal of delivering a reliable product. To do this, the testing team needs to create an effective test suite that aims to catch potential flaws in all aspects of the program, including each statement, branching condition, and possible execution path. However, the growing complexity of functional requirements and program specifications makes this a time-consuming task. The most basic approach is to perform exhaustive testing that uses all possible data combinations to test the application. Since there may be an infinite number of test inputs, this approach becomes impractical. Therefore, deterministic properties of source code entail the necessity of various code coverage criteria. Two such criteria are, condition coverage, and statement coverage. These criteria can also be used to measure and evaluate the quality of the system. Thus, code coverage criteria are considered a more effective approach for assessing a software system's confidence level. However, it can be challenging to achieve 100% code coverage, not only in terms of computational resources, but also from the perspective of human efforts. Specifically, these difficulties have three causes [1]:

- Certain coverage criteria may entail a broad spectrum of testing prerequisites.
- The process of generating test cases to meet these testing requirements cannot be fully automated.
- Test conditions that are not addressable through automation may require manual intervention.

Mutation testing is a testing method that focuses on identifying faults and offers a specific metric known as the mutation adequacy score. This metric assesses the ability of a test suite to identify errors, serving as an indicator of its efficacy. In a study by Chekam et al. [2] they empirically compared mutation, statement, and branch coverage, concluding that the mutation testing criterion outperformed other coverage techniques in fault detection within the test suite. Furthermore, Chen et al. [3] delved into the impact of the size of the test set and proposed a methodology to manage the test size while evaluating the test adequacy criteria. Given these considerations, it is imperative to reduce the size of the test suite without losing its effectiveness.

Since its introduction in 1978, mutation testing has been a subject of considerable academic inquiry [4]. Numerous studies have endeavored to transition mutation analysis from abstract theories into functional applications [5][6][7]. Recently, there has been a notable shift in research towards the integration of artificial intelligence techniques (AI) into

[1] *Department of Software Engineering, Izmir University of Economics Izmir, TURKIYE*
ORCID ID: 0000-0002-5139-5247
[2] *Department of Software Engineering, Izmir University of Economics Izmir, TURKIYE*
ORCID ID: 0000-0001-8614-0925
** Corresponding Author Email: uzunbayir.serhat@ieu.edu.tr*

mutation testing. AI has the capacity to greatly enhance mutation testing by automating tasks such as mutant generation, execution, and result evaluation. Machine learning models, trained on extensive test datasets, can discern trends in the behavior of mutants and identify those with a higher likelihood of inducing errors [8]. Moreover, metaheuristics offer considerable advantages. Such strategies focus on test suite optimization, harnessing the principles of natural evolution or swarm intelligence to pinpoint the most effective test cases for revealing induced mutations. It is evident that these innovations hold promise for enhancing the reliability of software testing by refining mutation testing methodologies and software quality.

Search-based mutation testing uses advanced metaheuristic optimization methods to improve mutation testing processes. These methods include bat algorithm, ant colony optimization, artificial bee colony, genetic algorithms, and other techniques [9]. These optimization methods are good at solving difficult problems by searching through many possible solutions quickly. One major challenge in this type of testing, however, is dealing with a large number of potential changes to the code, which can substantially slow the process. Therefore, it is useful to apply these techniques to mutation testing since the search space of the problem is large considering the number of mutants. Recent studies show that the use of these advanced methods can accelerate the process by reducing the number of code changes and test cases needed.

The research has two primary aims, first, to propose EvoColony, a combination of two metaheuristic approaches, genetic algorithms, and ant colony optimization for mutation testing, and second, to evaluate the effectiveness of the algorithm with traditional methods. Genetic algorithms can provide a good initial solution, but ant colony optimization can further refine it and escape local optima. The ant colony maintains a balance between exploring new paths and exploiting known good paths, potentially alleviating the premature convergence problem seen in genetic algorithms. Fusing two methods allows each to complement the disadvantages of the other, thus finding an optimal solution. The validation of the proposed approach is compared to a random search algorithm, two variants of a genetic algorithm using single and double crossovers, and traditional ant colony optimization. The results show that EvoColony successfully achieves better solutions than traditional approaches.

The rest of the paper is organized as follows. Section 2 presents preliminaries of mutation analysis, search-based mutation testing, genetic algorithms, ant colony optimization, and a review of the literature. Section 3 outlines research questions and elaborates the proposed method. Section 4 discusses experimental design preparation including the configuration of the test environment, the test data employed, and the benchmark algorithms considered. Section 5 shows and evaluates the experimental results. Finally, Section 6 concludes the article and discusses future work.

2. Related Work

This section offers an overview of mutation analysis and its procedures, introduces foundational ideas related to search-based mutation testing, discusses genetic algorithms and ant colony optimization, and provides a literature review based on earlier research on the topic.

2.1. Mutation Analysis

The concept of mutation analysis first came to light in 1971 [10]. Early shaping of the research area of mutation testing was largely influenced by the contributions of DeMillo et al. [4] and Hamlet [11] during the late 1970s. Mutation testing is characterized by the application of mutation analysis in formulating new test scenarios to scrutinize existing software evaluations. Primarily used in the context of unit testing, this white-box testing approach involves syntactic alterations to specific statements within the source code. The aim is to gauge the efficacy of the test suite by ascertaining its ability to detect faults in the program. Fundamentally, mutation testing serves as a tool for evaluating the adequacy of a testing strategy to ensure that the software meets quality standards.

Formally, given an original program P, mutation testing uses mutation operators to generate a set of mutants M for P. Mutation operators are used to apply syntactical transformation rules to generate mutants. These operators usually correspond to regular programmer mistakes. Each mutant $m \in M$ is identical to the original program P, with the sole exception that an altered program statement using a specific mutation operator. Then, all mutants in the set of M are executed using the test suite T of P. At the end of this procedure, the effectiveness of T is evaluated using a mutation adequacy score, comparing the results of execution between the mutants and the original program. A mutant m is killed by the test case $t \in T$ in P if its outcome is different; otherwise, m survives [12][13].

Mutation testing is a reliable method for identifying suitable test cases that can detect actual faults, however, due to the vast numbers involved, it is not feasible to generate mutations for all potential faults in a program. Consequently, mutation testing typically focuses on a subset of faults that are closest to the correct version of the program, under the assumption that this subset is able to provide an adequate representation of all faults.

The traditional mutation analysis process in which the mutants are generated, executed, and evaluated with the test suite involves the following six steps:

1) The original program P is modified using mutation operators to generate a range of mutants *M*. A mutant is a changed and faulty version of the original program. Table 1 shows an example of a simple mutation. Generated mutants can contain one or more faults, such as changed operators, operand position changes, or deletion of statements. This phase can be fully automated through tools that employ designated mutation operators on *P*.

Table 1: Original Program and Mutant

Original Program	Mutant m
Read v1, v2	Read v1, v2
if v1 != v2	if v1 != v2
v1 = v1 * v2	v1 = v1 + v2
end if	end if
end	end

2) The original program *P* and the mutants *M* are executed against test suite *T*.

3) To verify the accuracy of the output, the original program *P* is run using the test suite *T*. If the output is incorrect, corrections must be made to *P* before proceeding.

4) When the output of the original program is correct, the test suite *T* is executed with each living mutant. Then, its output is compared to the original program output to identify which mutants should be killed. The mutant survives if the original program and the mutant give the same results, otherwise it is killed [14] and eliminated from the process.

5) Following the execution of a mutant, the mutation adequacy score is calculated using formula (1) which represents the proportion of mutants killed to the total number of mutants that can be killed. The procedure ends when all mutants have been killed.

6) If any mutants survive, they must be inspected manually to decide if they are equivalent, or if the test cases are not adequate to kill them. Equivalent mutants yield identical outcomes to the original program and are invincible; that is, while they differ in syntax, they are functionally identical to the original program. If equivalent mutants are detected, they are eliminated. If there are no equivalent mutants that remain undetected, additional test cases should be added to test suite *T*, and the procedure resumes at step 4.

Mutation analysis assesses the test suite's quality through the use of a calculated metric called the mutation adequacy score [5]. The final objective for the tester would be to elevate the mutation adequacy score to 1, ensuring that the test suite *T* is capable of identifying all errors as indicated by the mutants. Thus, mutation testing provides a structured and effective way to measure the adequacy of the test adequacy [15].

$$\text{Mutation Adequacy Score} = \frac{\text{Killed Mutants}}{\text{All Mutants - Killed Mutants}} \quad (1)$$

2.2. Search-Based Mutation Testing

Search-based mutation testing combines the fundamentals of mutation testing with search-based optimization techniques [16]. It uses metaheuristic algorithms that can solve complex problems, such as genetic algorithms, ant colony optimization, or particle swarm optimization to autonomously create test cases capable of uncovering code mutations, thereby enhancing the overall quality of the test suite. The approach has two connected goals: to reduce the number of necessary test cases, and to boost their ability to detect mutants. Two other advantages are provided by this method: increasing the efficiency of the test process by reducing the time and computational resources needed to create and maintain a comprehensive test suite; and automating test case generation process, which reduces the manual effort required. The application of metaheuristics in testing procedures shows promise due to the large volume of inputs, specifically mutants, that need to be examined during the testing phase [17].

Various metaheuristic algorithms can be used with search-based mutation but each has some drawbacks. For example:

- Ant colony optimization is difficult to analyze theoretically. Its time of convergence is uncertain because of the nature of the search.
- Genetic algorithms have limitations such as converging too quickly, slow processing speed, lengthy iteration periods, and inefficiency in generating test cases.
- Particles in particle swarm optimization may get stuck in local minimum and the performance of the algorithm is sensitive to its parameters and coefficients.
- Simulated annealing may be slow to convergence when applied on complex problems. Also, the cooling schedule depends on the nature of the problem, it is critical to select it carefully.
- Tabu search may be computationally expensive and requires memory structure.

To address the drawbacks, a multifaceted strategy is often required. One common approach is to use hybrid models that combine the strengths of different algorithms, enhancing both reliability and efficiency. Parameter tuning can also be crucial; automated methods such as grid search or even dynamic adaptation during the algorithm's run can help determine the most effective settings. Additionally, benchmarking the algorithm against both synthetic and real-world scenarios can provide valuable insight into its performance.

2.3. Genetic Algorithms

Genetic algorithms draw inspiration from the natural selection processes observed in biological systems, grounded in Darwin's theory of evolution and Mendel's contributions [18]. They are a type of problem-solving method used to find answers sufficient to complex search optimization problems with a large number of candidate solutions. Finding the best solution among all is often considered as a time-consuming task; using the concept of the survival of the fittest the best solution is eventually found [19].

A typical genetic algorithm starts with a population of randomly generated individuals called chromosomes, where each individual represents a possible solution to the problem. These individuals are then evaluated based on a fitness function that quantifies how well they solve the problem. Then, parent individuals are selected and their offspring are produced using crossover and mutation operators, which are expected to be better solutions than the previous generations. After performing the procedure until the stopping condition is reached, the solutions converge to the optimal one.

One key feature of genetic algorithms involves mapping the problem onto chromosomes and establishing a domain-specific fitness function.

Chromosome: A chromosome serves as an individual solution to the problem. It signifies a specific location in the overall solution space and consists of a sequence of "genes," each of which corresponds to a particular attribute or parameter of the solution. The chromosome is structured in a way that enables its manipulation through the processes of the algorithm.

Fitness Function: A fitness function evaluates the effectiveness of a chromosome in addressing the problem by assigning it a performance-based score. Higher scores indicate more effective solutions. This function guides the algorithm to determine which solutions should be retained and which should be modified as it seeks the optimal solution.

A basic genetic algorithm is processed with six steps (Fig. 1):

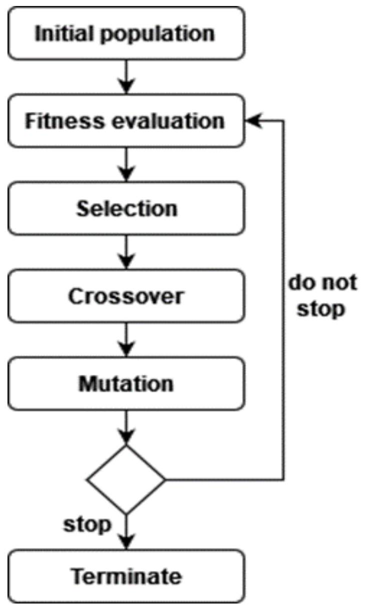

Fig 1: Traditional Genetic Algorithm Steps.

1) **Initial population:** The beginning of the algorithm involves randomly generated chromosomes to produce the initial population. The size of the population depends on the dataset and the problem domain.

2) **Fitness evaluation:** The fitness of the chosen population is evaluated according to the pre-defined fitness function set by the user.

3) **Selection:** Individuals are chosen based on their fitness levels to generate subsequent generations. If the fitness value is high, chromosomes are expected to be included in the new generation; if not they can be eliminated. Selection can be made by using different methods, such as roulette-wheel, rank, or tournament selection.

4) **Crossover:** Two parent individuals are used to create offspring by combining their genes. By combining the best traits of each parent, the algorithm can produce children that are potentially better solutions to the problem. There are possible different crossover methods, such as single-point, multi-point, uniform, or arithmetic crossover.

5) **Mutation:** Mutation is applied to the new offspring after crossover with a low probability. By adding small changes to an individual, it helps to maintain genetic diversity and explore new areas of the solution space.

6) **Terminate:** The algorithm is terminated when the best solution is found and returned. The termination condition is critical when controlling the computation time of the algorithm. There are different possible options to set as termination

conditions, such as fitness threshold, where either the maximum level of fitness is reached, or a predefined number of generations is produced.

Genetic algorithms offer a powerful approach to optimizing software testing processes, improving test effectiveness, and providing flexibility to introduce new genetic operators or mutation operators for specific testing scenarios.

2.4. Ant Colony Optimization

Ant colony optimization was introduced by Marco Dorigo [20] and inspired by the behavior of ants. In a natural habitat, ants can find the shortest route from a food source to their nest through a self-organized mechanism. When ants are released to an environment that contains a food source, they create trails to find food, and release a chemical called pheromone on the way back to the nest. Ants can sense this chemical and start to follow paths that have a higher intensity of pheromones. At the same time, the pheromone evaporates on the unused ant paths. Eventually, all the ants start to use the path that has the most intense level of pheromones, which is also the shortest path between the nest and the food source. Since the intensity of the pheromone is directly related to the probability of the path followed, this approach is considered a probabilistic method [21].

The ant colony algorithm requires a graph representation to formalize the problem being solved. Nodes in the graph represent a possible state or decision point, and the edges between nodes represent transitions between these states. The graph can be directed or undirected, depending on the problem being addressed.

A basic ant colony algorithm is processed with six steps (Fig. 2):

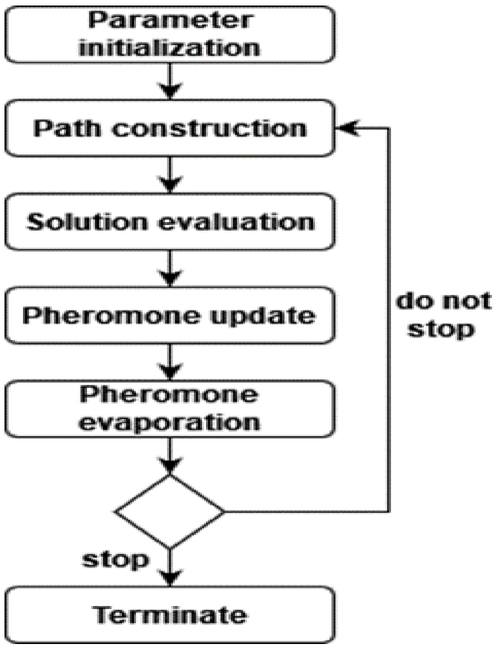

Fig. 2: Traditional Ant Colony Algorithm Steps.

1. **Parameter initialization:** At the start, the pheromone levels on the paths of the graph representation are initialized to zero to indicate that there are no pheromones. Ant colony optimization requires specific parameters to control the behavior of the algorithm. The common parameters can be defined as follows:

- *Number of ants:* This parameter represents the number of artificial ants that will traverse the graph to construct the solutions. The greater the number, the better the potential solutions. However, computational resources and time will be increased.
- *Pheromone evaporation rate (ρ):* This parameter is between 0 and 1, and represents the speed with which the pheromone markings fade. A high rate allows for quick adaptation, but may lead to instability.
- *Pheromone intensity (Q):* This parameter sets the starting pheromone concentrations on the paths. Striking a balance is crucial, as excessive or insufficient initial pheromone can affect the effectiveness of the algorithm.
- *Pheromone influence factor (α):* This parameter determines the weight given to the pheromone trails when the ants decide their routes.
- *Heuristic information (β):* This parameter is optional and sets the significance of any heuristic data available to guide the ants.
- *Termination criteria*: This parameter sets the total number of algorithm executions or other conditions that signal that the algorithm should stop.

2. **Path construction:** Ants create random trails on the graph and construct solutions at the start. In the next iterations, the solutions are updated based on a probabilistic decision-making process influenced by the pheromone levels on the paths and optional heuristic information.
3. **Solution evaluation:** Each ant trail is evaluated based on an objective function specific to the problem domain. For example, this would be the length of each path if the problem is to find the shortest path.
4. **Pheromone update:** Pheromone levels are increased based on the quality of solutions found. Generally, better solutions result in more pheromone being deposited. In some ACO variants, only the best ant or a set of elite ants can update the pheromone levels.
5. **Pheromone evaporation:** Pheromone levels on no-longer-used are reduced to simulate natural evaporation. This is usually done by multiplying the current pheromone level by (1−ρ).

6. **Terminate:** The algorithm terminates after a pre-defined stopping criteria is met. This could be a set number of iterations, a threshold value for the objective function, or some other custom condition.

Ant colony optimization offers several advantages that make it useful for solving optimization problems. In the context of mutation testing, which involves a large number of mutants, the ant colony can help navigate this vast space efficiently. The algorithm can prioritize mutants that are more likely to reveal faults, thereby reducing the number of test cases needed.

2.5. Literature Review

This section presents a review of related work on search-based mutation testing, and test case reduction using metaheuristics.

Haga and Suehiro [22] proposed a genetic algorithm that automatically produces test cases using mutation analysis. Their hybrid approach combines random generation and refinement. They used mutation adequacy score and experimented with a C-based project. In this approach, test cases are initially produced at random and subsequently fine-tuned through the use of a genetic algorithm. Their proposed genetic algorithm produced higher quality test suites. This study has some drawbacks, however, the use of only numeric data experimentation with, three mutation operators, and therefore, the impossibility of generating complex coverage measures. Unlike this study, our proposed solution involves more than one metaheuristic algorithm.

Sahoo and Ray [23] conducted a systematic review of the literature on metaheuristic search-based techniques for test case generation. The authors discussed the advantages and drawbacks of many reviewed metaheuristics techniques, such as genetic algorithms, bee colony optimization, tabu search, and ant colony optimization for software testing. This article provides useful insights on many metaheuristics within the context of test case evaluation.

Arora and Baghel [24] proposed a hybrid approach that fuses a particle swarm optimization and a genetic algorithm to increase the quality of the test suite by reducing the number of test cases. The algorithm is tested with six programs and randomly generated test cases. The results show that the number of tests can be halved using the proposed approach. This study used random test cases and was not applied in mutation testing.

Palak [25] introduced a hybrid approach using genetic algorithms and ant colony optimization to choose optimal test cases with the aim of recognizing the development time and cost. This approach focused on component-based software projects and selected a subset of test cases from the test suite. The results showed that the proposed approach is able to find 100% faults using 33% of the test cases. This study shows potential, but it has not been applied to large-scale programs involving mutation testing.

Suri and Singhal [26] worked on the selection and prioritization of test cases. They showcased an execution of an ant colony optimization algorithm that had been previously described. Their proposed technique takes as input the fault identification data and the time required to run the regression test suite. In this algorithm, the time required to run each test case serves as its execution cost. The goal was to maximize the total number of faults detected while minimizing the overall execution cost. The results showed that the method produces near optimal solutions. This study shows that ant colony optimization can be used to select and prioritize test cases.

Jatana and Suri [27] presented an improved version of the crow search to automate the test suite generation. Using the principles of mutation testing, they simulated the behaviors of crows and incorporated the Cauchy distribution. This was aimed at overcoming the shortcomings of the original crow search algorithm, which frequently becomes trapped in local searches. By using Cauchy random numbers, the algorithm's ability to explore the solution space increased. They used mutation sensitivity testing to define the fitness function for this approach. The experiments showed that the proposed approach produces better results than other search-based mutation testing algorithms.

In a recent study by Tsagaris et al. [28], a genetic and ant colony algorithm is combined and applied to a travel salesman problem. Their aim was to reduce the total distance thus; reducing the travel time. Their solution achieved up to 40% path optimization compared to a traditional genetic algorithm. Furthermore, the authors also verified their approach by using a real coordinate measuring machine. This study shows good potential results; however, it is less relevant since our approach is applied in mutation testing rather than path optimization problem.

According to the discussed studies above, metaheuristics can be used to experiment with test case problems. Our novel approach is applied in search-based mutation testing compared to others, and using a hybrid approach has been found effective in reducing test cases and increase the quality of testing procedures.

3. Methodology

In this section, research questions are defined and the proposed method is explained.

3.1. Research Questions

The primary objective of this study is to propose and evaluate the effectiveness of a hybrid approach called EvoColony in mutation testing. The research aims to answer the following questions (RQs):

- **RQ1:** Is EvoColony algorithm able to reduce test cases in a test suite while preserving a strong mutation adequacy score?
- **RQ2:** How well does EvoColony perform compared to traditional search-based approaches considering random search, genetic algorithms, and ant colony optimization?

By answering these research questions, our aim is to offer a comprehensive view of the benefits of incorporating metaheuristic algorithms into search-based mutation testing.

3.2. EvoColony: A Hybrid Approach

EvoColony combines genetic algorithm with ant colony optimization. The reasons for this merger are the drawbacks of genetic algorithms, and how ant colony optimization's capacity to overcome these. For example, solutions of genetic algorithms may converge to local optima rather than continuing the search through global optima. On the other hand, ant colony algorithms adapt better to dynamic changes in the problem landscape, potentially avoiding premature convergence to local optima. Fine-tuning of genetic algorithm parameters such as crossover rate, mutation rate, and selection strategy can be cumbersome, compared to ant colony optimization, which requires fewer and easier-to-tune parameters. Therefore, ant colony optimization can improve the results of the genetic algorithm by taking these parameters as input, and distributing the initial pheromones based on these while initializing the graph.

The proposed approach can be divided into four parts (Fig. 3):

1) **Input Preparation:** The first part is the preparation of the inputs. EvoColony requires three inputs; a test program, its test suite, and the mutant programs.

2) **Genetic Algorithm:** In part two, the genetic algorithm processes start. In the initialization, test cases from the test suite are encoded as chromosomes, a fitness function is determined using equation (3), and the parameters of the genetic algorithm are defined. The fitness function is defined as:

$$\boldsymbol{f(x) = M(x) + (1 - S(x))} \tag{3}$$

where $\boldsymbol{f(x)}$ is the fitness of test suite $\boldsymbol{x}$, $\boldsymbol{M(x)}$ represents mutation adequacy score for test suite $\boldsymbol{x}$, and $\boldsymbol{S(x)}$ is the normalized size of test suite $\boldsymbol{x}$.

The fitness of the initial population is evaluated. Then, the parents of the next generation are selected using roulette-wheel selection. Larger fitness function values increase the likelihood that chromosomes will be selected. The mating process involves applying, single-point crossover and mutation based on a certain probability, and a new solution is created. The processes iterate until the best solution is found.

3) **Ant Colony Optimization Algorithm:** The third part is the ant colony optimization. Taking the best solution from part two as input, the ant colonyparameters are initialized, the ant colony graph is created, and the initial pheromones are distributed to the paths based on the genetic algorithm solution. The ants start moving on the paths and the pheromones are updated on the trails. If the ants stop following a particular path, the pheromone evaporates. The solution is evaluated and the processes are repeated until the best solution is found.

4) **Output:** In the fourth part, the best solution is returned. The expected output is a reduced test set by keeping the best mutation score.

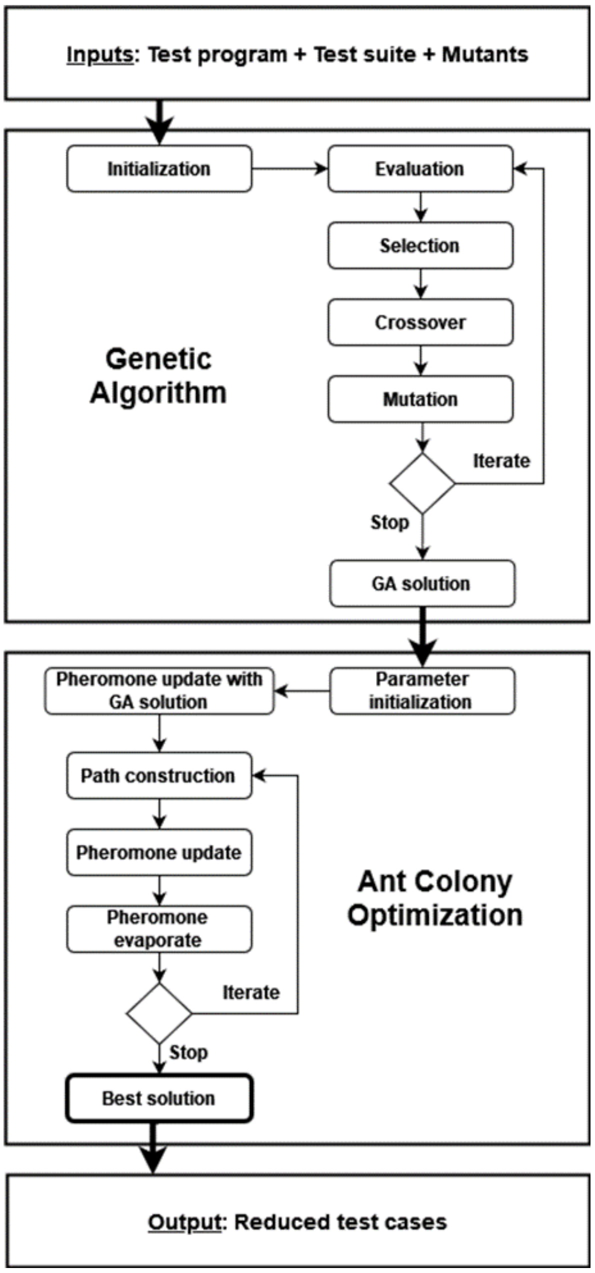

Fig. 3: The EvoColony Algorithm Steps.

4. Experimental Design

In this section the test environment is explained, test data is presented, and benchmark algorithms are discussed.

4.1. Test Environment

The tests were carried out on ten distinct programs, all of which were coded in the C# language. Some programs were selected from the software testing literature [29] [30], whereas others are selected from frequently used mutation testing experiments. They are converted into C# when implementation was not available. To increase diversity, the subject program set includes various characteristics: mathematical calculations, array operations, and complex branching conditions. Each subject program differs in program sizes. Test suites for each program were initially generated using IntelliTest and later manually fine-tuned to confirm that the program being tested is free from errors and that the experiments are not influenced by any runtime exceptions. Mutants were generated using VisualMutator for each program (Fig. 4). We have selected five standard mutation operators that VisualMutator supports since the mutation adequacy score was almost identical when the rest operators were also selected. The selected operators for this study are as follows:

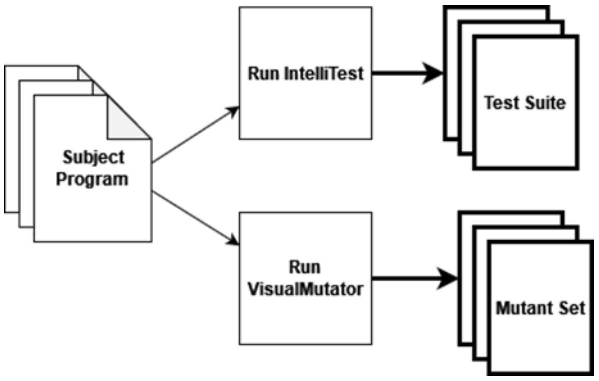

Fig. 4: Test Data Setup.

- *Arithmetic Operator Replacement:* Substitutes one arithmetic operator for another (+, -, *, /).
- *Relational Operator Replacement:* Changes relational operators (<, <=, >, >=, ==, !=).
- *Logical Operator Replacement:* Substitutes logical operators (&, |, ^).
- *Logical Connector Replacement:* Substitutes logical connectors (&&, ||).
- *Operator Deletion:* Produces two mutants from each operation (+, -, >, <=, %).

Moreover, four object-oriented mutation operators are selected to represent object-oriented features. These are the following:

- *Accessor Method Change:* Changes the accessor methods for a class's properties or fields.
- *Accessor Modifier Change:* Alters the accessibility level of a property or field accessor.
- *Member Variable Initialization Deletion:* Removes the initialization of member variables.
- *Member Call from Another Inherited Class:* Mutates a method call to a member of the current class (or a base class) to a call to a member of another class that shares the same base class.

To perform a comparative analysis, five different algorithms were implemented; a random search, two variants of a traditional genetic algorithm with single-point crossover and double-point crossover, a traditional ant colony algorithm, and the proposed EvoColony algorithm. These algorithms were used to execute each subject program. The execution was performed on a desktop computer running Windows 11 Pro operating system with an Intel i7 9700k 2.8 GHz processor.

4.2. Test Data

The details of the subject programs are given below (Table 2) with their size in lines of code (LOC) and the number of mutants. Total number of code size is 1037 and the total number of mutants experimented on the study is 1227. Initial mutation adequacy scores for all programs are calculated as 1.000.

Table 2: Subject Program Details

Subject Program	Size in LOC	# of Mutants
BubbleSort	93	85
Calendar	115	114
TriangleType	123	149
ArrayOperations	113	176
TemperatureConverter	104	95
QuadraticSolver	73	88
HashTable	107	151
BinarySearch	64	164
BankAccount	76	110
AutoDoor	169	195
Total	**1037**	**1227**

4.3. Benchmark Algorithms

Selected benchmark algorithms for the experiments are as follows:

- *Random Search (RS):* Random search is used to reduce test cases in mutation testing by iteratively and randomly selecting subsets of test cases, evaluating their mutation adequacy scores, and then comparing those scores that are in an acceptable range. The process continues until a stop target mutation adequacy score is reached.

- *Traditional Genetic Algorithm with Single-Point Crossover (GA-SP):* A genetic algorithm is employed for test case reduction by initializing a population with random subsets of test cases and then applying a single-point crossover and mutation to create new generations. The fitness function based on the mutation adequacy score is used to evaluate each subset of test cases. The algorithm iteratively refines the population to optimize the fitness while reducing the number of test cases, ending when a maximum generation is reached.
- *Traditional Genetic Algorithm with Double-Point Crossover (GA-DP):* In this version of a genetic algorithm, two crossover points are selected within the test case sequence, and portions of parent sequences between these points are swapped to produce offspring. The fitness of each offspring is then evaluated using the mutation adequacy score. Double-point crossover can encourage greater diversity in the test case subsets and potentially result in a more efficient and effective reduced set. The algorithm iteratively applies double-point crossover, along with other operations such as mutation, to evolve the population until it reaches a (near) optimal fitness level.
- *Traditional Ant Colony Optimization Algorithm (ACO):* Each test case is represented as a node in a graph. Then, the process of selecting a set of test cases as a path through this graph is specified. Initially, pheromones are evenly distributed across all paths. During each iteration, several artificial ants construct solutions by navigating the graph based on a probabilistic function influenced by pheromone levels. After completing a path, each ant evaluates the fitness of its selected test case subset based on the mutation adequacy score. Pheromones are then updated: paths that lead to higher fitness values are reinforced, while others evaporate. The algorithm iterates until a maximum number of iterations is met.
- *EvoColony:* The proposed algorithm from Section 3.2 is implemented, and its results are compared with other algorithms to evaluate the effectiveness of the test suite.

Genetic Algorithm Specific Parameters	
Selection type:	Roulette-wheel selection
Crossover probability:	0.8
Mutation probability:	0.06
Population size:	Twice the number of initial test cases
Chromosome size:	Number of test cases
Maximum iteration:	200 generations
Ant Colony Algorithm Specific Parameters	
α:	2
β:	2
ρ:	0.02
Q:	Based on the genetic algorithm results
Number of ants:	100
Maximum iteration:	1000

Table 3: Genetic and Ant Colony Parameters of EvoColony

5. Results and Evaluation

In this section, experimental results are presented and evaluated. Experiments were conducted based on the research questions in Section 3.1.

RQ1: *Is EvoColony algorithm able to reduce test cases in a test suite while preserving a strong mutation adequacy score?*

This research question aims to answer whether the EvoColony algorithm can minimize the number of tests without sacrificing the quality or effectiveness of those tests in detecting defects.

Table 3 shows the parameters of genetic and ant colony algorithm parts of EvoColony. These parameters were selected based on a combination of experimentation, empirical evidence, computational efficiency considerations.

Subject Program	Initial # of Test Cases	Reduced Test Suite Size	Reduction Ratio
BubbleSort	105	52	50.48%
Calendar	130	70	46.15%
TriangleType	126	89	29.37%
ArrayOperations	149	81	45.64%
TemperatureConverter	115	63	45.22%
QuadraticSolver	108	72	33.33%
HashTable	119	65	45.38%
BinarySearch	123	69	43.90%
BankAccount	152	94	38.16%
AutoDoor	165	102	38.18%
Total	**1192**	**757**	**36.49%**

Table 4: EvoColony Results

Table 4 provides a comprehensive view of test results of EvoColony across subject programs. The comparative results have the following information: initial number of test cases, reduced test suite size, and reduction ratio. Mutation adequacy score calculated for all experiments still remain perfect even after the reduction. Reduction ratio is calculated by using the following formula (3):

$$\text{Reduc. Ratio} = \frac{\text{Initial \# of test cases - Reduced \# of test cases}}{\text{Initial number of test cases}} \text{x}100 \quad (3)$$

One of the standout observations is the significant reduction in the number of test cases across the board. For instance, *BubbleSort* saw its test cases drop by a substantial 50.48%, while maintaining a perfect mutation adequacy score. This implies that not only were the test cases reduced, but they were optimized so that they were still able to cover the necessary conditions and branches for effective mutation testing. However, it is important to note the variation in reduction percentages. *TriangleType* had the smallest reduction at 29.37%, suggesting room for improvement. Despite these reductions, the mutation adequacy scores remain perfect for all programs. This high level of mutation adequacy indicates that the sets of test cases, although reduced, are nevertheless of high quality, and effective in identifying potential faults.

The summary statistics at the bottom of the table offer a macro-level view, showing that 1192 initial test cases were reduced to 757, a considerable overall reduction of 36.49%. Yet, despite this number of reduced test cases, the mutation adequacy score for all subject programs remains perfect at 1.000. This demonstrates that the reduction in test cases does not mean sacrificing test adequacy, and therefore, the useless test cases are removed.

The test results successfully illustrate that EvoColony can be used to significantly reduce the number of test cases without compromising the quality of the testing, as evidenced by perfect mutation adequacy scores.

RQ2: *How well does EvoColony perform compared to traditional search-based approaches considering random search, genetic algorithms, and ant colony optimization?*

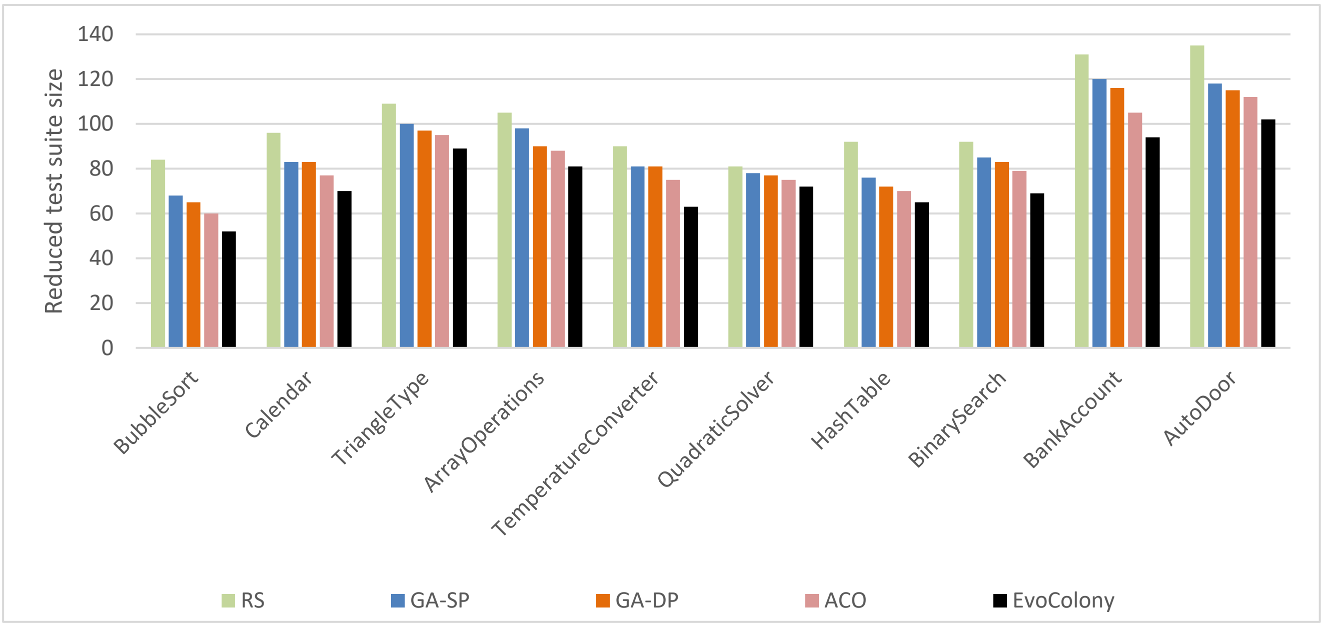

Fig. 5: Reduced Test Suite Size Comparison.

This research question aims to answer how well the EvoColony algorithm in comparison with other common methods used to optimize software tests. The goal is to determine whether the performance of EvoColony is better, worse, or similar in terms of enhancing the quality of test suites.

According to the results in Table 5, RS consistently produces the smallest reductions and performs worst when compared to other approaches. The comparatively poor performance of RS is most likely due to its inherent lack of optimization capabilities. RS operates without an iterative optimization process, unlike more advanced algorithms, such as GA variants (GA-SP and GA-DP) and ACO. Furthermore, it does not adapt or learn from previous iterations, and its random selection process misses the opportunity for more informed decision making, further compromising its effectiveness. GA variants yield moderately effective reductions compared to RS and often produce very similar results. Their effectiveness is relatively consistent across various subject programs, suggesting that they are robust and reliable methods for this task. The two variants perform similarly, with only slight differences in their reduction percentages, depending on the specific program under test. The efficacy of GA variants appears to be sensitive to the specific program being tested. For example, both algorithms perform the same amount of reduction for the *Calendar* and *TemperatureConverter* with 36.15% and 29.57% respectively, but vary in effectiveness for other subject programs. Additionally, GA variants not only excel compared to RS but also hold their ground against ACO and EvoColony.

While GA variants offer robust and reliable performance, ACO generally shows better test case reduction for subject programs such as *BubbleSort* and *Calendar*, for which it achieves reductions of 42.86% and 40.77%, respectively. Therefore, the results indicate that this is the second-best performer.

		RS		GA-SP		GA-DP		ACO		EvoColony	
Subject Program	**Initial # of Test Cases**	**Reduced Test Suite Size**	**Reduction Ratio**	**Reduced Test Suite Size**	**Reduction Ratio**	**Reduced Test Suite Size**	**Reduction Ratio**	**Reduced Test Suite Size**	**Reduction Ratio**	**Reduced Test Suite Size**	**Reduction Ratio**
BubbleSort	105	84	20.00%	68	35.24%	65	38.10%	60	42.86%	52	50.48%
Calendar	130	96	26.15%	83	36.15%	83	36.15%	77	40.77%	70	46.15%
TriangleType	126	109	13.49%	100	20.63%	97	22.99%	95	24.60%	89	29.37%
ArrayOperations	149	105	29.53%	98	34.23%	90	39.60%	88	40.94%	81	45.64%
TemperatureConverter	115	90	21.74%	81	29.57%	81	29.57%	75	34.78%	63	45.22%
QuadraticSolver	108	81	25.00%	78	27.78%	77	28.70%	75	30.56%	72	33.33%
HashTable	119	92	22.69%	76	36.13%	72	39.50%	70	41.18%	65	45.38%
BinarySearch	123	92	25.20%	85	30.89%	83	32.52%	79	35.77%	69	43.90%
BankAccount	152	131	13.82%	120	21.05%	116	23.68%	105	30.92%	94	38.16%
AutoDoor	165	135	18.18%	118	28.48%	115	30.30%	112	32.12%	102	38.18%

Table 5: Comparative Test Results

EvoColony appears provide the greatest benefits in test case reduction of other four approaches. It consistently shows superior performance in minimizing the number of test cases across all subject programs. Moreover, Fig. 5 illustrates the comparison of the test suite reduction with respect to the results in Table 5. Our results indicate that EvoColony is able to reduce, on average, one-third of all test cases, making it the best-performing algorithm for the experiments conducted in this research.

6. Conclusions and Future Work

Software testing is an indispensable component of the software development lifecycle, serving as a cornerstone for quality assurance. Mutation testing helps developers to assess the effectiveness of the existing test suite at detecting faults in terms of mutation adequacy score. This research introduced EvoColony, a novel hybrid approach that employs both genetic algorithms and ant colony optimization for search-based mutation test suite reduction. It is shown that the algorithm is able to reduce test cases in a test suite by maintaining a perfect mutation score. The effectiveness of the proposed approach was evaluated with a comparative analysis.

Our experimental findings validate the greater efficiency of EvoColony in comparison to four other methods: random search, two variants of genetic algorithms, and ant colony optimization. Notably, the proposed hybrid approach significantly outperformed the other methods in test case reduction, while maintaining the quality of test suites. This achievement underscores the utility of leveraging complementary algorithms. EvoColony offers several advantages, most notably, its dual optimization strategy. Using genetic algorithms for the initial search and ant colony optimization for refinement allows the leveraging of the strengths of both methods, circumventing their individual weaknesses.

While this study marks a promising advancement in the realm of mutation testing, further research is warranted. Future work could focus on extending the capacity of EvoColony to process more complex types of mutations or integrating it with other machine learning techniques for more adaptive testing strategies. The results may be compared with other search-based methods such as particle swarm optimization, tabu search, simulated annealing, or hybrid approaches.

Acknowledgements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for profit sectors.

Author contributions

Serhat Uzunbayir: Conceptualization, Methodology, Software, Field study, Data collection, Visualization, Investigation, Coding, Writing-Original draft preparation, Editing.

Kaan Kurtel: Conceptualization, Methodology, Visualization, Supervision, Writing-Reviewing.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] M. Kintis, “Effective methods to tackle the equivalent mutant problem when testing software with mutation,” Ph.D. dissertation, Department of Informatics, Athens University of Economics and Business, Athens, Greece, 2016.

[2] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, “An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption,” *2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, Institute of Electrical and Electronics Engineers Inc., Jul. 2017, pp. 597–608. doi: 10.1109/ICSE.2017.61.

[3] Y. T. Chen et al., “Revisiting the relationship between fault detection, test adequacy criteria, and test set size,” *2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, Institute of Electrical and Electronics Engineers Inc., Sep. 2020, pp. 237–249. doi: 10.1145/3324884.3416667.

[4] R. A. DeMillo, R. J. Lipton and F. G. Sayward, “Hints on test data selection: help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34-41, April 1978, doi: 10.1109/C-M.1978.218136.

[5] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5. pp. 649–678, 2011. doi: 10.1109/TSE.2010.62.

[6] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, “Mutation testing in the wild: findings from GitHub,” *Empirical Software Engineering*, vol. 27, no. 6, Nov. 2022, doi: 10.1007/s10664-022-10177-8.

[7] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, “Mutation testing advances: an analysis and survey,” *Advances in Computers*, vol. 112, pp. 275–378, Jan. 2019, doi: 10.1016/bs.adcom.2018.03.015.

[8] A. Panichella and C. C. S. Liem, “What are we really testing in mutation testing for machine learning? a critical reflection,” *Proceedings of the 43rd International Conference on Software Engineering: New Ideas and Emerging Results*, pp. 66–70, May 2021, doi:10.1109/ICSE-NIER52604.2021.00022.

[9] N. Jatana, B. Suri, and S. Rani, “Systematic literature review on search based mutation testing,” *E-Informatica Software Engineering Journal*, vol. 11, no. 1. Politechnika Wroclawska, pp. 59–76, Jan. 01, 2017. doi: 10.5277/e-Inf170103.

[10] A. J. Offutt, “Mutation 2000: uniting the orthogonal,” *Wong, W.E. (eds) Mutation Testing for the New Century, The Springer International Series on Advances in Database Systems*, vol 24., 2001, Springer, Boston, MA. doi:10.1007/978-1-4757-5939-6_7.

[11] R. G. Hamlet, “Testing programs with the aid of a compiler,” *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279-290, July 1977, doi: 10.1109/TSE.1977.231145.

[12] Y. Jia and M. Harman, “Higher order mutation testing,” *Information Software Technology*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009, doi: 10.1016/j.infsof.2009.04.016.

[13] L. Chen and L. Zhang, “Speeding up mutation testing via regression test selection: an extensive study,” *2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, Institute of Electrical and Electronics Engineers Inc., May 2018, pp. 58–69. doi: 10.1109/ICST.2018.00016.

[14] I. Leontiuc, “Continuous mutation testing in modern software development,” M.S. thesis, Department of Software Technology, Faculty EEMCS, Delft University of Technology, Delft, the Netherlands, 2017.

[15] Y.-S. Ma and J. Offutt, “Description of method-level mutation operators for java,” George Mason University, 2005.

[16] R. A. Silva, S. do R. Senger de Souza, and P. S. Lopes de Souza, “A systematic review on search based mutation testing,” *Information Software Technology*, vol. 81, pp. 19–35, Jan. 2017, doi: 10.1016/j.infsof.2016.01.017.

[17] N. Jatana, B. Suri, and S. Rani, “Systematic literature review on search based mutation testing,” *E-Informatica Software Engineering Journal*, vol. 11, no. 1. Politechnika Wroclawska, pp. 59–76, Jan. 01, 2017. doi: 10.5277/e-Inf170103.

[18] V. Sharma, D. Rakesh Kumar, D. Sanjay Tyagi, P. Scolar, and A. P. Dcsa, “A review of genetic algorithm and mendelian law,” *International Journal of Science Engineering Research*, vol. 7, 2016, [Online]. Available: http://www.ijser.org

[19] S. Uzunbayir, “A genetic algorithm for the winner determination problem in combinatorial auctions,” *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, Sarajevo, Bosnia and Herzegovina, 2018, pp. 127-132, doi: 10.1109/UBMK.2018.8566446.

[20] M. Dorigo, M. Birattari and T. Stutzle, “Ant colony optimization,” *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28-39, Nov. 2006, doi: 10.1109/MCI.2006.329691.

[21] S. Uzunbayir, “Reverse ant colony optimization for the winner determination problem in combinatorial auctions,” *International Conference on Computer Science and Engineering, UBMK 2022*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 19–24. doi: 10.1109/UBMK55850.2022.9919488.

[22] H. Haga and A. Suehiro, “Automatic test case generation based on genetic algorithm and mutation analysis,” *2012 IEEE International Conference on Control System, Computing and Engineering*, Penang, Malaysia, 2012, pp. 119-123, doi: 10.1109/ICCSCE.2012.6487127.

[23] R. R. Sahoo and M. Ray, “Metaheuristic techniques for test case generation: a review,” *Research Anthology on Agile Software, Software Development, and Testing*, vol. 2, IGI Global, 2021, pp. 1043–1058. doi: 10.4018/978-1-6684-3702-5.ch052.

[24] D. Arora and A. S. Baghel, “A hybrid meta-heuristics technique for finding optimal path by software test case reduction,” *International Journal of Hybrid Information Technology*, vol. 8, no. 4, pp. 35–40, Apr. 2015, doi: 10.14257/ijhit.2015.8.4.05.

[25] Palak and P. Gulia, “Hybrid swarm and ga based approach for software test case selection,” *International Journal of Electrical and Computer Engineering*, vol. 9, no. 6, pp. 4898–4903, 2019, doi: 10.11591/ijece.v9i6.pp49898-4903.

[26] B. Suri and S. Singhal, “Implementing ant colony optimization for test case selection and prioritization,” *International Journal on Computer Science and Engineering*, pp. 1924-1932, Nov. 2011.

[27] N. Jatana and B. Suri, “An improved crow search algorithm for test data generation using search-based mutation testing,” *Neural Processing Letters*, vol. 52, no. 1, pp. 767–784, Aug. 2020, doi: 10.1007/s11063-020-10288-7.

[28] A. Tsagaris, P. Kyratsis, and G. Mansour, “The integration of genetic and ant colony algorithm in a hybrid approach,” *International Journal of Intelligent Systems and Applications in Engineering*, vol. 11, no. 2, pp. 336 –, Feb. 2023.

[29] M. B. Bashir and A. Nadeem, “Improved Genetic Algorithm to Reduce Mutation Testing Cost,” *IEEE Access*, vol. 5, pp. 3657-3674, 2017, doi: 10.1109/ACCESS.2017.2678200.

[30] F. Wedyan, A. Al-Shishani, and Y. Jararweh, “GaSubtle: a new genetic algorithm for generating subtle higher-order mutants,” *Information (Switzerland)*, vol. 13, no. 7, Jul. 2022, doi: 10.3390/info13070327.