Transactions on Machine Intelligence 7(3) (2024) 208-222

ISSN Online: 2821-1693

Transactions on Machine Intelligence

Journal Homepage: https://tmachineintelligence.ir/index.php/TMI

Increasing Error Detection in Software Testing Using Cuckoo Algorithm and Gravity Search Algorithm

S. Meydani[1], F. Sadeghi[2,*]

[1] Bahmanyar Institute of Higher Education, Kerman, Iran
[1] Assistant Professor, Bahmanyar Institute of Higher Education, Kerman, Iran

ARTICLE INFO

Article History:
Received 26 May 2024
Received in revised form 8 August 2024
Accepted 20 September 2024
Available online 25 September 2024

Keywords:
Software Testing; Cuckoo Search Optimization Algorithm; Gravity Search Algorithm; Prioritizing Test Items

ABSTRACT

In the realm of software testing, one of the most critical challenges faced by development teams is the limitation of resources and time. As software systems grow in complexity and size, the number of test cases increases significantly, making it impractical to re-execute the entire suite of tests in each testing cycle. Consequently, there arises a need for effective strategies to select and prioritize test cases in a way that ensures the most valuable and error-prone parts of the software are tested early. Prioritizing test cases not only accelerates the detection of defects but also enhances the efficiency of the testing process by focusing efforts on areas with higher potential for failure. In this study, two powerful nature-inspired metaheuristic algorithms Cuckoo Search Optimization Algorithm and Gravitational Search Algorithm are employed to address the problem of test case prioritization. These algorithms are used to prioritize test cases based on coverage criteria, particularly aiming for maximum fault detection coverage. By optimizing the sequence of test execution, the proposed method improves both the fault detection rate and the overall effectiveness of the testing process. This approach contributes to the timely identification of critical defects and supports faster and more reliable software releases.

1. INTRODUCTION

Software testing constitutes a significant portion of the overall cost in software development. Each minor modification in the codebase typically results in a new version of the software, necessitating a comprehensive validation process. This involves both designing new test cases and re-executing previously conducted tests—a process known as regression testing. However, due to limitations in time and computational resources, executing the full suite of test cases in every iteration becomes impractical. To address this challenge, it is essential to implement test case selection and prioritization strategies that ensure the most critical test cases are executed early in the testing cycle, aligning with the primary objectives of the test process [1].

* Corresponding Author: sattar.meydani@gmail.com
Bahmanyar Institute of Higher Education, Kerman, Iran

http://dx.doi.org/10.47176/TMI.2024.208

The complexity of regression testing arises not only from the volume of test cases but also from the time-consuming nature and constrained budget typically allocated to it. In this study, a hybrid approach combining the Cuckoo Search Optimization Algorithm and the Gravitational Search Algorithm is proposed for prioritizing test cases within a regression test suite. Test case prioritization is performed based on code coverage criteria—test cases that cover a larger portion of the code are scheduled for earlier execution. This approach enhances both the effectiveness and efficiency of fault detection, significantly reducing the time required to uncover critical software defects. Given the vast search space associated with optimizing regression test suites, evolutionary algorithms offer a promising solution due to their heuristic search capabilities. The proposed method was implemented and evaluated across various performance metrics. Comparative analysis demonstrates its superiority over baseline approaches and conventional prioritization techniques [2].

Software testing, broadly defined, is the process by which the quality and reliability of a software system are assessed. While testing typically involves executing a program to identify errors, its scope extends to ensuring that the software meets specified requirements and operates reliably within the intended hardware and environmental conditions. Considering the high financial investment in software products, software failures can result in substantial sometimes irreversible economic losses for both developers and users. Studies have shown that more than one-third of these losses could be mitigated through more effective testing processes [3].

When modifications are made to a software system, they may introduce new errors or inadvertently affect other components. Therefore, previously tested parts must be revalidated to ensure no new faults have been introduced this is the essence of regression testing. Since this process is repeated frequently during development, it tends to significantly increase testing time. Prioritizing test cases in regression testing is thus a critical strategy to enhance fault detection efficiency and reduce overall testing duration. The goal is to confirm that prior bugs have been resolved and that new changes have not negatively impacted existing functionalities [4].

2. THE IMPORTANCE AND NECESSITY OF CONDUCTING RESEARCH

The increasing size of regression test suites and the complexity of their selection processes present significant challenges, particularly due to time constraints and limited testing budgets. As the demand for high-quality software products continues to rise, comprehensive testing has become a crucial phase in the software development lifecycle. Regression testing is performed following code modifications to ensure that recent changes do not adversely affect existing functionality. However, the testing process is often hindered by constraints such as limited resources, tight development schedules, ambiguous or evolving requirements all of which negatively impact the effectiveness of regression testing [5].

In this context, leveraging code coverage information for the selection and prioritization of test cases can substantially accelerate fault detection. Establishing a systematic prioritization based on defined criteria enables testing teams to achieve their quality objectives while reducing the overall testing cycle duration [6]. Since software testing is both resource-intensive and critical to the software lifecycle, regression testing serves as a key activity to mitigate the risk of software degradation during maintenance. Although many organizations rely on executing the complete system test suite to validate changes, this approach is costly and inefficient. To optimize testing efforts, it is widely recommended to use a reduced yet effective regression test set that can still provide assurance of software integrity [7].

Given these considerations, this research aims to develop an automated method for prioritizing regression test suites. The goal is to generate an optimal subset of tests using an efficient algorithm that maximizes fault detection within minimal execution time. In recent years, test process automation has gained significant momentum, and numerous commercial tools and academic studies have emerged in this area.

The prioritization problem has been extensively studied, and various methodologies have been proposed. One common approach is the general defect detection method [8], which seeks to uncover all existing faults in the modified codebase. Several algorithmic strategies have been employed to enhance the prioritization process. Greedy algorithms [9] prioritize test cases by iteratively selecting the most beneficial cases first. Evolutionary algorithms aim to evolve a high-performing test suite over iterations, while non-evolutionary algorithms [10] focus on goal-driven prioritization strategies. Ant colony optimization algorithms [11] mimic the foraging behavior of ants to identify optimal test execution orders.

For companies seeking cost-effective solutions, version-specific algorithms [12] are particularly suitable, as they focus on prioritizing only the modified portions of the software. Additionally, variable analysis algorithms [13] have been proposed to assess the interdependencies between altered variables and their influence on the system, offering another dimension to the prioritization task.

One emerging and promising optimization approach is Bee Colony Optimization (BCO). BCO has been successfully applied in various domains, including general optimization problems [14], the NP-hard Traveling Salesman Problem (TSP) [15], and several scheduling challenges such as job-shop scheduling, resource allocation, and project planning [16]. Furthermore, BCO has also found applications in improving search engine performance through more effective web search optimization techniques [17].

3. SOFTWARE TESTING

effective test data methods, the software engineer must understand the basic principles that guide software testing. In this regard, a set of principles have been proposed, which we will use in the following:

- All tests must be traceable to the customer's requirements. As we have seen, the purpose of software testing is to detect errors. That is, most of the severe defects (from the customer's point of view) are those that make the program unable to meet his needs. The test should be planned long before the start of the test.

- Test planning can begin as soon as the requirements model is completed. Detailed definition of test data can begin as soon as the design model is consolidated. Therefore, all tests can be planned and designed before producing any code.

- [The "Pareto" principle applies to software testing. In simple terms, the "Pareto" principle states that 80% of all errors discovered during testing are probably detectable in 20% of all program components. The problem is to isolate the suspect components and test them completely.

- The test should start in small dimensions and expand to larger dimensions. The first tests are performed on each of the components. With the progress of the test, errors are found in a set of complex components andthen in the whole system.

- Full testing is not possible. The number of possible routes for the average program is also large. Therefore, it is not possible to implement any combination of routes, but it is possible to cover the program sufficiently [3].

In order for the test to be most efficient, it should be performed by an impartial third party. The most efficient means that it will find the errors with the highest probability. For the reasons mentioned earlier in this chapter, the software engineer who created the system has done, he is not the best person who should do all the tests [3].

4. THEORY AND BACKGROUND

The size and selection process of regression test suites introduce considerable complexity, primarily due to their time-consuming execution and the budget constraints typically faced by testing teams. As the demand for high-quality software continues to grow, rigorous testing has become an indispensable component of the software development lifecycle. Regression testing is particularly vital following software modifications, ensuring that changes do not introduce new faults or compromise existing functionality. However, test teams often operate under constraints such as limited resources, compressed testing schedules, and evolving or ambiguous requirements, all of which can hinder the effectiveness of regression testing.

To address these challenges, the selection and prioritization of test cases using code coverage metrics can significantly enhance the efficiency and speed of fault detection during the testing process [4]. Establishing a clear execution order and assigning priorities to test cases based on predefined criteria enables teams to meet testing objectives more effectively while concurrently minimizing the duration of the test cycle [4].

Software testing represents one of the most critical and costly phases in the software lifecycle. Regression testing plays a crucial role in mitigating the adverse effects of software changes during maintenance. Many organizations continue to rely on full re-execution of their system-level test suites to validate changes. However, this approach is both expensive and time-intensive. To optimize resource usage and reduce testing overhead, a smaller yet targeted regression test suite is recommended for validating the modified sections of the software [5].

Several approaches have been proposed for test case prioritization, one of which involves utilizing the historical execution data of test cases. For instance, a method that considers both the fault detection history and the age of test cases has shown promising results. Experimental evaluations indicate that this approach improves fault detection efficiency and provides more stable outcomes compared to random prioritization techniques. Another history-aware strategy involves collecting regression test histories and employing a genetic algorithm to identify and prioritize the most impactful test cases [5].

5. REGRESSION TEST

After implementing software changes designed to enhance performance or fix errors, new issues may arise. Regression testing involves re-executing a subset of previously successful tests to verify that the software has not unintentionally introduced errors into previously functioning areas. The primary objective of regression testing is to detect new bugs or regressions, whether in previously functioning parts or in areas that were not previously active, following changes such as optimizations, patches, or configuration updates. Its goal is to ensure that these modifications do not inadvertently introduce defects or disrupt the software's established functionality [5].

A key motivation for regression testing is to determine whether a change in one area of the system affects other parts of the software. Traditional regression testing involves rerunning tests that were previously successful before the changes, allowing for an assessment of whether the software's behavior has changed post-modification. Additionally, regression testing helps identify whether defects that were previously fixed have re-emerged [5].

Regression testing can be applied at any level of the software, from unit tests to system tests, and can be automated for greater efficiency. Specific types of regression tests, such as health tests and smoke tests, are employed for particular purposes. The primary focus of regression testing is on retesting after changes, with traditional regression tests repeating previously designed tests. However, in risk-oriented regression testing, new tests are employed for previously tested areas, with increasing complexity over time. The purpose of regression testing is to mitigate two key risks: [6]

- The risk that changes introduce unintended side effects.
- The risk that a change intended to resolve one issue inadvertently creates a new problem, either by failing to resolve the original issue or by introducing new defects [6].

6. GRAVITY SEARCH ALGORITHM

The Gravitational Search Algorithm (GSA) is an innovative metaheuristic optimization technique inspired by Newton's law of universal gravitation. According to this principle, every object in the universe exerts an attractive force on all other objects, with the intensity of this force directly proportional to their masses and inversely proportional to the square of the distance between them. Importantly, no object is exempt from the influence of this universal force [18]. The gravitational force F12 represents the pull exerted by mass m2m_2m2 on mass m1, where heavier and closer objects exert stronger attractions. Furthermore, Newton's second law of motion states that an object will accelerate in the direction of the applied force, with the acceleration proportional to the magnitude of that force.

GSA adopts this concept to simulate a system in which all agents (particles) attract each other based on their fitness-derived masses. Heavier particles, which represent better solutions due to their higher fitness values, exhibit slower movement, while lighter particles move more quickly. This mechanism enables a dynamic and collective search strategy, where particles are progressively drawn toward heavier (i.e., fitter) solutions. Over time, this interaction guides the population of solutions toward the global optimum in the search space [18], [19].

Each agent in the system is modeled as a particle with a mass that is directly associated with its fitness. The mass of a particle iii at time ttt is calculated using the following equation:

$$P_i(t) = \frac{\text{Fitness}_i(t) - \text{Worst}(t)}{\text{Best}(t) - \text{Worst}(t)} \tag{1}$$

where $Fitness_i(t)$ denotes the fitness value of particle i at time t. For minimization problems, the smallest fitness value is selected as the best, and the largest as the worst. In maximization problems, this selection is reversed.

Another important element of GSA is the gravitational constant $G(t)$, which governs the strength of attraction and decays over time to fine-tune the balance between exploration and exploitation. It is defined as:

$$G(t) = G(t_0) \times \left(\frac{t_0}{t} \right) \tag{2}$$

where $G(t_0)$ is the initial gravitational constant at the beginning of the search (i.e., time t_0), and $G(t)$ represents its value at a later time t. This time-dependent decay reflects a diminishing influence of gravitational attraction as the search progresses, which helps prevent premature convergence.

The pseudocode of the GSA is illustrated in Figure 1, and its operational flow is depicted in Figure 2 [18].

1- Determining the system environment and initial setting.
2- Initial location of objects.
3- Evaluation of objects.
4- Update the values, G, best, worst
5- Calculate the mass of each agent (M).
6- Calculate the force on each object.
7- Calculate the acceleration and speed of each object.
8- Updating the position of objects.
9- If the stopping condition is not met, we return to step 3. Otherwise, the best answer seen so far is given to the output and the algorithm stops.

Fig. 1. The pseudocode for the gravity search algorithm

7. CUCKOO SEARCH ALGORITHM

The Cuckoo Search (CS) algorithm is a meta-heuristic optimization method that adopts an evolutionary approach to finding optimal solutions. Proposed in 2009, this method is inspired by the unique behavior of cuckoo bird species, particularly their egg-laying strategy. Cuckoos are known for laying their eggs in the nests of other bird species, tricking the host birds into caring for their offspring. Some cuckoo species mimic the appearance of the host bird's eggs and chicks, increasing the likelihood that the host will unknowingly raise the cuckoo egg. In the event that the host bird detects the intruding egg, it may destroy it or eject it from the nest. This intriguing behavior has been leveraged to develop the Cuckoo Search algorithm, which mimics this reproductive strategy to find solutions to optimization problems.

The algorithm operates by maintaining a set of "host nests," each containing one solution (analogous to an egg). The best solution seeks to generate a new solution by exploring the existing ones and making changes to certain features [20]. The cuckoo bird's strategy, which optimizes survival with minimal effort, provides a robust framework for an optimization method that can effectively "outsmart" other solutions in the competitive landscape.

Much like other evolutionary algorithms, the Cuckoo Search method begins with an initial population (a set of cuckoos). Each cuckoo lays its eggs in the nests of host birds, with the eggs resembling the host bird's own eggs to improve the chances of survival. Eggs that are detected by the host are discarded, while those that successfully blend in are allowed to develop. The more eggs that survive and are cared for, the higher the fitness value of that region. The ultimate goal of the Cuckoo Search algorithm is to optimize the area where the largest number of eggs are preserved [21].

The steps of the Cuckoo Search algorithm are outlined in Table 1.

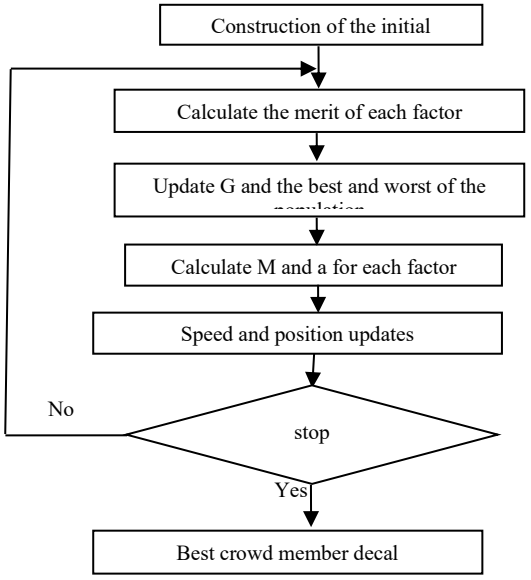

Fig. 2. Flowchart of gravity search algorithm

Table 1. Steps of the cuckoo search algorithm

Step 1	Creating an initial population of cuckoos.
Step 2	Each cuckoo creates one egg at a time and chooses a nest randomly and lays its egg in it.
Step 3	The number of available hosts or nests is fixed and the egg laid by the cuckoo will be discovered by the host bird with probability p.
Step 4	The best nest based on the high quality of the egg inside it is transferred to the next generation and the worst nests are removed.
Step 5	Steps ٢ to ٤ are repeated until the desired quality is reached or a certain number of steps are completed.
Step 6	Return existing nests as result.

Cuckoos instinctively seek out the most favorable regions that maximize the survival rate of their offspring. Once hatched and matured, these young cuckoos form flocks, each occupying a distinct territory. Periodically, all flocks migrate toward the territory that currently offers the greatest reproductive advantage that is, the region where the most eggs have successfully incubated. Upon arrival, each flock establishes its nests in proximity to this optimal site. Based on both the number of eggs laid by each cuckoo and their distance from the optimal region, a "nesting radius" is computed for each bird. Within this radius, cuckoos randomly deposit their eggs into host nests. Over successive iterations, the collective movements and randomized egg-laying converge on the habitat that yields the highest egg survival rate, effectively identifying the globally optimal solution [20].

Empirical studies have demonstrated that the standard Cuckoo Search (CS) algorithm often outperforms classical evolutionary approaches such as Differential Evolution—and swarming algorithms like Particle Swarm Optimization (PSO), particularly in high-dimensional search spaces. Enhancements to CS include the Modified Cuckoo Search, which incorporates inter-egg interactions to further refine the search process and achieve performance on par with, or exceeding, PSO in complex function optimization. For multi-objective problems, the Multi-Objective Cuckoo Search extends the original algorithm by permitting multiple eggs per nest, thereby generating a uniformly distributed Pareto front of solutions. This extension has been shown to rival or surpass other multi-objective techniques in real-world scenarios with diverse constraints [24].

Beyond theoretical benchmarks, CS has been successfully applied to a variety of domains, including training feedforward and spiking neural networks, embedded system design optimization, game-playing agent development, and combinatorial programming tasks. Its strengths derive from the efficient implementation of Lévy-flight–inspired random walks, a minimal set of control parameters, and the algorithm's inherent simplicity—attributes that collectively contribute to its robust performance across different problem classes [21]. The overall workflow of the Cuckoo Search algorithm is depicted in the block diagram in Figure 3.

8. OBJECTIVE FUNCTION OF EVOLUTIONARY ALGORITHMS

In this article, two standard evaluation criteria are used as objective function and competence in the evolution process of gravity search algorithm and cuckoo search optimization, and these two criteria are used as follows:

8.1. Average percentage of fault detection (APFD)

Coverage-based prioritization techniques rely on test case coverage data over the program code, typically represented in binary form. Incorporating detailed coverage information can be highly effective for achieving more efficient prioritization of test cases. In this study, the Average Percentage of Fault Detection (APFD) is employed as the primary evaluation metric [3], [4].

The APFD metric is widely used to assess the performance of a test suite T in detecting faults in a program P. It measures how quickly faults are detected when test cases are executed in a specific order.

The APFD is defined as follows:

$$\mathrm{APFD}(T,P) = 1 - \left(\frac{Tf_1 + Tf_2 + \cdots + Tf_m}{n \cdot m} \right) + \frac{1}{2n} \tag{3}$$

Where:

- m: the total number of faults in the program,
- n: the number of test cases in the test set T,
- $Tf_1, Tf_2, \ldots, Tf_m$: the positions of the first test cases in T that detect faults 1 through m.

This metric evaluates how early faults are detected across the sequence of test cases. A higher APFD value indicates that more faults are found earlier in the execution sequence, which generally reflects better prioritization effectiveness.

8.2. Average Percentage Condition Coverage (APCC)

An important metric for evaluating test case selection and prioritization methods is the Average Percentage Condition Coverage (APCC). This metric assesses the extent to which the prioritized execution order of test cases successfully exercises the conditional branches within a program. The significance of APCC arises from the fact that certain commands reside within conditional statements and are only executed under specific circumstances. If such commands contain faults, these may only manifest during rare or exceptional execution paths, making them difficult to detect without proper test coverage.

To effectively uncover such faults prior to deployment, it is essential that test cases be designed and ordered in a way that maximally triggers these conditions and simulates the corresponding execution paths. This ensures thorough testing of the conditional logic and facilitates early error detection.

Formally, given a set of test requirements TR and a coverage criterion C, a test set is said to satisfy C if every requirement in TR is addressed by at least one test case within the set. The APCC metric quantitatively expresses this coverage, providing a standardized means to compare the effectiveness of different prioritization strategies [5].

$$\mathrm{APCC}(T,P) = \left[1 - \left(\frac{TC_1 + TC_2 + \ldots + TC_m}{m \cdot n} \right) + \frac{1}{2n} \right] \tag{4}$$

that here,

T: The set of examined test cases
m: the number of conditions in the program under test P
n: the total number of test items in the test set
TC_i: The location of the first test item in T that checks the i-th condition.

9. GRAVITY SEARCH ALGORITHM USED

Software testing plays a vital role in ensuring software quality. A software system can only be considered reliable if its functionalities are thoroughly tested and verified. When modifications are introduced to a system, it becomes essential to validate that these changes have not inadvertently impacted other parts of the codebase. Regression testing addresses this concern by re-evaluating the software to confirm that existing functionalities remain intact. Typically, regression testing is conducted iteratively throughout the software development life cycle.

Despite its importance, regression testing is often resource-intensive and time-consuming. Consequently, a considerable body of research has focused on optimizing this process particularly through test case prioritization techniques, which aim to improve the efficiency of fault detection by strategically ordering test execution.

Among the many strategies proposed, the Gravitational Search Algorithm (GSA) stands out as a novel and effective metaheuristic approach. GSA has been widely applied to diverse optimization problems, including test case prioritization, due to its adaptability and ease of customization. One of its key advantages is the flexibility to tailor its operators to specific problem domains, making it especially suitable for test suite optimization.

This study introduces a test case prioritization approach based on the Gravitational Search Algorithm, with the goal of achieving 100% code coverage. To demonstrate the methodology, the well-known triangle classification problem is utilized, as it is frequently cited in software testing literature. The decision graph associated with this problem, illustrated in Figure 4, is adapted from standard software testing practices. This graph visualizes the conditions covered by each path, serving as a reference for evaluating the effectiveness of the proposed method [18].

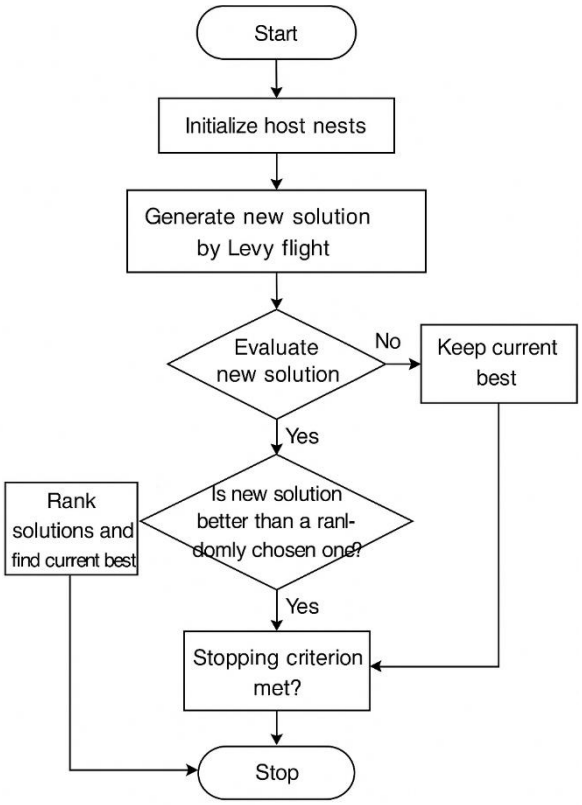

Fig. 3. Flowchart of the cuckoo search algorithm

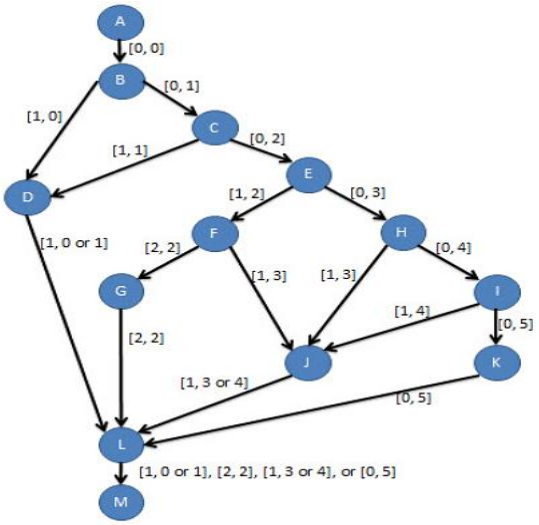

Fig. 4. Decision graph for triangle detection problem

This graph is made based on the commands in the program. Each node represents a line of the program and the edges represent the sequence of program execution. Nodes that have two output edges represent conditional commands (If) where the execution path from the first edge represents the execution node if the condition is true and the execution path from the other edge represents the execution node if the condition is false. According to the execution conditions, each of the test cases goes through one of the existing paths from the start node to the end, and the quality of each entry is determined based on the quality of the related path in terms of covering nodes and also covering conditions. The test set is randomly generated based on different criteria, each of which satisfies a part of the total coverage. A fitting function is determined for each path. In this article, the effectiveness of the gravity search algorithm approach and five other approaches are analyzed using the average percentage of code coverage (APCC)[6].

10. MAPPING THE GRAVITY SEARCH ALGORITHM TO THE PROBLEM OF PRIORITIZING TEST CASES

This study introduces a binary version of the Gravitational Search Algorithm (GSA) for addressing the discrete nature of the test case prioritization problem, which differs fundamentally from the original GSA designed for continuous optimization. While the traditional GSA operates within a continuous search space, many real-world optimization problems such as test case prioritization in software testing require exploration within a discrete binary space.

In the binary variant, agents navigate a search space composed solely of binary values (0 and 1). Conceptually, this space can be visualized as a hypercube, where each vertex represents a candidate solution defined by a unique combination of binary values. Movement within this space involves transitioning between the corners (i.e., flipping binary values), and each dimension of a particle's position is expressed as either a 0 or a 1. A particle "moves" in a given dimension by flipping its binary state, and this movement is guided by a probabilistic function derived from its velocity. Specifically, the particle changes its state in a given dimension with a probability determined by this function.

In the binary GSA, the computation of gravitational force, velocity, and particle position updates largely mirrors those in the continuous version of the algorithm. However, a key distinction lies in the use of Hamming distance—rather than Euclidean distance to measure inter-particle proximity in the binary space. Additionally, the gravitational constant which governs the strength of attraction between particles evolves over time, decreasing linearly rather than exponentially as in the original continuous GSA. This adjustment ensures smoother changes in bit values, which is essential since the gravitational influence in binary space affects individual bits rather than continuous dimensions.

Consequently, a more gradual decay in the gravitational constant is more appropriate for maintaining convergence stability in discrete binary environments [19].

$$G(t) = G_0(1 - \frac{t}{T}) \quad (5)$$

In the binary algorithm v_i^d , it is converted into a probability function and is limited to the interval [0-1]. This function should be defined in such a way that as the particle velocity increases, the probability of changing the position of the object increases. At speeds close to zero, the probability of changing the situation is close to zero[18].

$$S\left(V_i^d(t)\right) = |\tanh(vd_i(t))| \quad (6)$$

It should be noted that for the proper convergence of the algorithm, it should be limited to a suitable interval. In other words. The value equal to 6 is considered.

After calculating the above probability function, the object moves in each dimension according to the equation below. According to this relationship, the object changes its position with a probability in one dimension. The higher the speed of the object in one dimension, the greater the probability of the object moving in that dimension. Changing the position of an object in one dimension of binary space means changing its value from zero to one or vice versa. It is a random number with a uniform distribution in the interval[19].

In this section, how to use the discrete gravity search algorithm for the problem of prioritizing test items and how to map each operator to the upcoming problem is explained. The crime structure in the proposed method is in the form of permutations of test items to show the order of their implementation. The absorption operation is shown in Figure 5 The bit string is swapped after this point. Also, this figure shows the mass obtained from the absorption operation, which received half of the values from the first mass and the other half from the second mass in a combined form. As a result, the first child becomes a new set because it covers all independent paths[19].

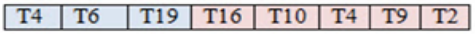

Fig. 5: Mass structure in the proposed method

11. CUCKOO ALGORITHM MAPPING IN PRIORITIZING TEST ITEMS

In this study, the proposed algorithm aims to minimize the overall cost of regression testing by reducing the number of executed test cases without compromising fault detection effectiveness. The algorithm is a hybrid approach, integrating the Cuckoo Search (CS) algorithm with the Gravitational Search Algorithm (GSA). This section elaborates on the Cuckoo Search component of the optimization process. The algorithm identifies an optimal subset of test cases that can detect all potential faults while minimizing execution time.

In the context of the Cuckoo Search algorithm, the test case prioritization problem is mapped such that each cuckoo's nest represents a candidate solution, and the eggs within a nest symbolize the sequence of prioritized test cases for that solution.

To reduce the number of test cases, the proposed method incorporates regression test selection alongside prioritization strategies. The Cuckoo Search is executed using a randomized exploration strategy, designed to identify the smallest subset of test cases capable of detecting all known faults with minimal runtime. This process is guided by the Average Percentage of Fault Detection (APFD) metric, which serves as the primary evaluation criterion. Hence, the CS algorithm is applied to efficiently identify the minimum set of high-priority test cases that maximizes fault detection effectiveness, as reflected by a high APFD value [20].

The following assumptions are considered for the proposed algorithm:

- The initial population of test cases is randomly generated. The population is represented as TS = {tc1, tc2, …, tct}.
- The fitting function is based on the APFD criterion.
- In the presented problem, the fitting function is APFD and APCC criteria. The stopping criterion is all

possible errors covered in the minimum time or the algorithm processed for the maximum number of given iterations.

- In the main test suite, each test case (tc1, tc2, ..., tcn) covers some errors.
- The set of errors is shown as F = {f1, f2, ..., fm}.
- Test cases are represented in binary with m bits (m is the total number of errors).
- Each bit of the test item depends on the error detection capability. Bit 1 indicates error detected and bit 0 indicates no error.
- The input is the number n of the sequence of host nodes, n is the number of test cases and m is the number of possible detected errors.
- Here the sequence number of host nodes is equal to the number of test cases.
- The output is the sequence of traversing the test cases and meeting the stopping criteria. Here, the stop criterion is to reach the predetermined number of generations and iterations.

12. PRIORITIZATION OF TEST ITEMS BASED ON THE OPTIMIZATION ALGORITHM OF GRAVITY SEARCH AND CUCKOO SEARCH

The Gravitational Search Algorithm (GSA) operates through multiple stages of exploration. However, one of its notable limitations is its inadequacy in local search, particularly during the final iterations of the optimization process. In these stages, the algorithm may converge prematurely and become trapped in local optima. While GSA is effective in global exploration continually seeking solutions with better fitness values across the search space it lacks the mechanisms necessary to escape local minima. Consequently, the algorithm may experience a decline in convergence speed as optimization progresses.

To address this issue, the Cuckoo Search (CS) algorithm is incorporated as a complementary strategy to improve local search capability. In the proposed hybrid model for prioritizing software regression test cases, GSA serves as the primary global search algorithm, while CS functions as a population refinement tool. There are two main motivations for integrating these algorithms: (1) the need for a population-based method that can thoroughly explore the solution space associated with test case prioritization, and (2) the requirement for a rapid convergence mechanism that can efficiently adapt to changes and provide optimal combinations of test cases.

Despite its limitations in local search, GSA is computationally efficient for global exploration. When enhanced with CS, the hybrid method becomes robust in both global and local search phases. Due to the high computational cost of CS operations when applied to each GSA particle, the CS algorithm is only activated on the final population obtained from the last iteration of the GSA. This final GSA output acts as the initial solution for the CS, thereby optimizing the local search without excessive overhead during the iterative process [21].

One of the primary challenges in this hybrid optimization lies in how solutions are represented and transferred between the two algorithms. Specifically, the integration mechanism must determine how potential solutions are encoded, how optimal solutions are selected, and how CS nests (representing candidate solutions) interact with GSA objects during the optimization process. In the proposed method, each cuckoo nest corresponds to a possible solution and is represented by an array of n elements. Each element in the array is randomly initialized as an integer between 1 and m, representing possible test case configurations.

The flowchart of the proposed hybrid algorithm which combines GSA and CS for optimal test case prioritization in software regression testing is depicted in Figure 6 [18, 21].

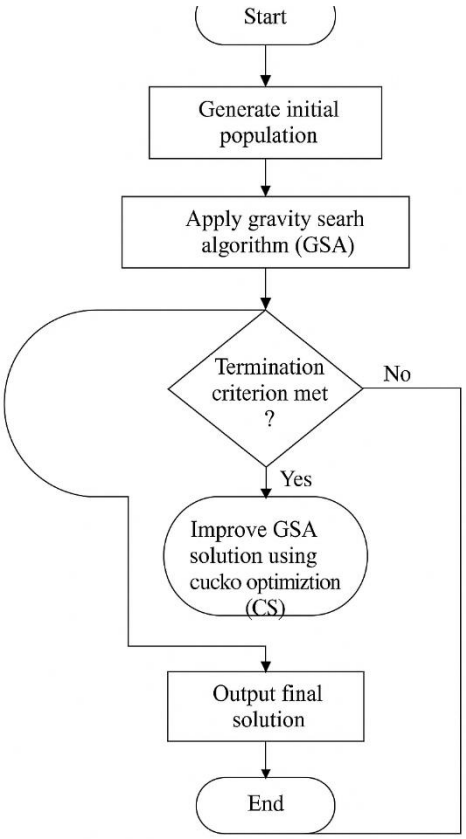

Fig. 6. Combined flowchart of gravity search algorithm and cuckoo optimization

In this study, the Gravitational Search Algorithm (GSA) is employed as the initial phase of the prioritization process within a regression test suite, aiming to achieve complete fault coverage. The prioritization leverages the Average Percentage of Condition Coverage (APCC) as a guiding metric. To overcome the GSA's inherent weakness in local search and to prevent entrapment in local optima, the Cuckoo Search (CS) optimization algorithm is integrated into each GSA iteration. This hybrid approach utilizes GSA-generated solutions as the foundational structure for nests and eggs within the CS algorithm. The Average Percentage of Fault Detection (APFD) metric is employed to evaluate and guide the optimization process of the CS algorithm, thereby integrating both APCC and APFD as dual evaluation criteria in the proposed framework.

As illustrated in Figure 6, after each iteration of GSA, the solutions it produces are transformed into corresponding representations within the CS framework, where objects become nests and their components are modeled as eggs. The CS algorithm then conducts a global exploration to enhance these candidate solutions. The two algorithms operate in an alternating manner—GSA initiating the search, followed by CS refining the solutions repeating this sequence to progressively converge on an optimal prioritization of test cases.

Following the initial GSA execution, control transitions to the CS algorithm. Here, randomized nest updates generate new candidate solutions, which are assessed using the stated performance criteria. The best solution from each CS iteration is compared against the best solution from all previous iterations, and if superior, it replaces the former as the current global optimum. The CS process iterates for a pre-defined number of cycles, refining the solution set. Subsequently, control returns to the GSA, where standard gravitational operations—such as object attraction and positional updates resume.

This iterative integration of GSA and CS continues until the predefined number of GSA iterations is reached. Upon completion, the most effective solution obtained across all iterations of both algorithms is returned as the final output. The effectiveness of the proposed hybrid method will be evaluated in the next chapter through two key metrics: APFD, for fault detection efficiency, and APCC, for condition coverage performance [6].

13. CONCLUSION

Software testing represents one of the most critical and cost-intensive phases in the software development life cycle. Within this domain, regression testing plays a key role in mitigating the adverse effects of changes introduced during software maintenance. To ensure software reliability, organizations often re-execute the entire system test suite after modifications. However, this comprehensive re-testing is both resource-intensive and time-consuming. To address these challenges, researchers have proposed executing a smaller, more focused regression test suite that still ensures sufficient validation of the modified software.

Given these considerations, there is a clear need for automated test case prioritization techniques that can generate an efficient subset of tests capable of detecting the maximum number of faults in minimal time. The growing interest in automated testing has led to the emergence of numerous commercial tools and a wealth of academic literature in this area.

Several approaches have been explored for test case prioritization using heuristic algorithms, such as ant colony optimization, genetic algorithms, and fuzzy logic. In this article, for the first time, a hybrid approach combining the Gravitational Search Algorithm (GSA) and the Cuckoo Search (CS) algorithm is introduced. This novel combination is specifically designed to prioritize test cases in regression testing scenarios.

The proposed method targets modified code segments and aims to maximize test efficiency through this dual-algorithmic strategy. Implementation results are thoroughly analyzed and compared against various existing prioritization techniques, revealing the effectiveness and superiority of the proposed method across several performance metrics.

This hybrid model draws inspiration from the natural foraging behavior of cuckoos and the gravitational attraction mechanism in physics, using them to navigate the solution space effectively. Due to the inherent complexity and time-intensive nature of selecting and executing regression tests within a constrained budget, this hybrid approach aims to prioritize test cases for full fault coverage with minimal execution cost.

The algorithm's effectiveness is evaluated using two widely accepted metrics: the Average Percentage of Fault Detection (APFD) and the Average Percentage of Condition Coverage (APCC). Results demonstrate a significant improvement over traditional techniques, indicating the method’s suitability for real-world testing scenarios.

The performance of the proposed algorithm was further validated using a standard case study: the triangle classification problem, which is commonly used in software testing literature to evaluate prioritization methods. The experimental findings confirmed that the prioritization generated by this method closely approximates the optimal prioritization and outperforms several existing approaches.

Given the manual implementation limitations, the current evaluation was restricted to test suites with a small number of test cases. Therefore, to enable scalability and practical deployment in larger software systems, automating the extraction of test cases from source code and integrating this with the proposed prioritization algorithm is recommended for future work. The development of a comprehensive automation tool could facilitate broader adoption and fully realize the potential of this algorithm in industrial environments.

Declaration

We acknowledge that we used ChatGPT to enhance the academic writing of our manuscript while ensuring the originality and integrity of our work.

Transparency Statement

The data supporting this study are available upon reasonable request to the corresponding author, subject to ethical and confidentiality considerations.

Acknowledgments

We would like to express our gratitude to all individuals who contributed to this project.

Declaration of Interest

The authors declare that they have no competing interests.

Funding

This research received no specific grant from any funding agency, commercial, or not-for-profit sectors.

REFERENCES

[1] Chi, J., Wang, Q., Yang, Z., & Li, X. (2020). Relation-based test case prioritization for regression testing. Journal of Systems and Software, 163, 110539. https://doi.org/10.1016/j.jss.2020.110539

[2] Lima, J. A. P., & Vergilio, S. R. (2020). Test case prioritization in continuous integration environments: A systematic mapping study. Information and Software Technology, 121, 106268. https://doi.org/10.1016/j.infsof.2020.106268

[3] Liu, Y., Li, J., Zhang, L., & Zhou, J. (2023). More precise regression test selection via reasoning about semantics-modifying changes. Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. https://doi.org/10.1145/3597926.3598086

[4] Khaleel, S. I., & Anan, R. (2023). A review paper: Optimal test cases for regression testing using artificial intelligent techniques. International Journal of Electrical & Computer Engineering, 13(2), 1803–1816. https://doi.org/10.11591/ijece.v13i2.pp1803-1816

[5] Torbunova, A., Strandberg, P. E., & Porres, I. (2024). Dynamic test case prioritization in industrial test result datasets. arXiv preprint arXiv:2402.02925. https://doi.org/10.1145/3644032.3644452

[6] Sebastian, A., Naseem, H., & Catal, C. (2024). Unsupervised machine learning approaches for test suite reduction. Applied Artificial Intelligence, 38(1), 2322336. https://doi.org/10.1080/08839514.2024.2322336

[7] Pan, R., Gao, J., Deng, Y., & Zhang, Z. (2022). Test case selection and prioritization using machine learning: A systematic literature review. Empirical Software Engineering, 27(2), 29. https://doi.org/10.1007/s10664-021-10066-6

[8] Luo, Q., Xu, Y., Liu, Y., Yang, B., & Luo, J. (2020). Automated visual defect detection for flat steel surface: A survey. IEEE Transactions on Instrumentation and Measurement, 69(3), 626–644. https://doi.org/10.1109/TIM.2019.2963555

[9] Qasim, M., Alenezi, M., & Mahmoud, S. A. (2021). Test case prioritization techniques in software regression testing: An overview. International Journal of Advanced and Applied Sciences, 8(5), 107–121. https://doi.org/10.21833/ijaas.2021.05.012

[10] Marois, A., Mattei, N., Pesant, G., & Rossi, F. (2021). Evaluation of evolutionary algorithms under frugal learning constraints for online policy capturing. 2021 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA). IEEE. https://doi.org/10.1109/CogSIMA51574.2021.9475930

[11] Raimond, K., & Lovesum, J. (2019). A novel approach for automatic remodularization of software systems using extended ant colony optimization algorithm. Information and Software Technology, 114, 107–120. https://doi.org/10.1016/j.infsof.2019.06.002

[12] Bhanushali, A., Gupta, K., & Sharma, M. (2024). Innovative approaches for machine-driven software testing using neural networks. International Journal of Intelligent Systems and Applications in Engineering, 12(2), 724–732.

[13] Mistry, J. (2024). Impact of model selection on pulmonary effusion diagnosis using prediction analysis algorithms. Journal of Xidian University, 18(1), 611–618.

[14] Seyyedabbasi, A. (2023). A reinforcement learning-based metaheuristic algorithm for solving global optimization problems. Advances in Engineering Software, 178, 103411. https://doi.org/10.1016/j.advengsoft.2023.103411

[15] Hernández-Sabaté, A., Albarracín, L., & Sánchez, F. J. (2020). Graph-based problem explorer: A software tool to support algorithm design learning while solving the salesperson problem. Mathematics, 8(9), 1595. https://doi.org/10.3390/math8091595

[16] He, Y., & Wang, M. (2024). An improved chaos sparrow search algorithm for UAV path planning. Scientific Reports, 14(1), 366. https://doi.org/10.1038/s41598-023-50484-8

[17] Asaad, M. M. O. F., Wahid, J., & Rahmat, A. R. (2024). Employing artificial bee colony algorithm to optimize the artificial neural network in heart disease prediction. AIP Conference Proceedings, 2895(1). AIP Publishing. https://doi.org/10.1063/5.0192144

[18] Ramesh, R., & Karthic, M. J. (2024). Optimizing cryptocurrency price prediction: A hybrid approach with resilient stochastic clustering and gravitational search algorithm. International Journal of Intelligent Systems and Applications in Engineering, 12(17s), 239–248.

[19] Momeni, E., Momeni, R., & Sadrekarimi, A. (2021). An efficient optimal neural network based on gravitational search algorithm in predicting the deformation of geogrid-reinforced soil structures. Transportation Geotechnics, 26, 100446. https://doi.org/10.1016/j.trgeo.2020.100446

[20] Al-Abaji, M. A. (2021). Cuckoo search algorithm: Review and its application. Tikrit Journal of Pure Science, 26(2), 137–144. https://doi.org/10.25130/tjps.v26i2.130

[21] Rath, P. K., Singh, A., Tripathy, B. K., & Satapathy, S. C. (2024). CSOFS: Feature selection using cuckoo search optimization algorithm for software fault detection. 2024 International Conference on Emerging Systems and Intelligent Computing (ESIC). IEEE. https://doi.org/10.1109/ESIC60604.2024.10481641