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

Critical Path-Aware Deep Learning Architecture for Efficient Test Case Prioritization and Minimization

Vinita Tomar[1], Mamta Bansal[2], and Pooja Singh[3]

Submitted: 05/05/2024 **Revised**: 18/06/2024 **Accepted**: 25/06/2024

Abstract: Test case prioritization and minimization are essential practices to augment the efficiency of the testing process in software testing. However, conventional methods often struggle with large-scale software systems due to their inability to effectively handle the critical path, leading to suboptimal prioritization and minimal test coverage This research paper proposes an Enhanced Electric-Eel with Critical Path-Aware Foraging Optimization (EECPFO) algorithm tailored for test case minimization and prioritization in software testing. The algorithm is designed to address the challenges of minimizing redundancy, prioritizing critical paths, and maintaining diversity in test suites. To achieve this, modifications including a Dynamic Fitness Function, Redundancy-aware Foraging, Critical Path Sensitivity, and Diversity Maintenance are integrated into the EEFO framework. The proposed algorithm is evaluated for its effectiveness using three open-source Java programs (JTopas, Ant, and JMeter) from the Software Infrastructure Repository (SIR), employing well-known metrics such as Average Percentage of Fault Detection (APFD) and Average Percentage of Fault Detection with Cost (APFDc). Experimental results demonstrate significant improvements in test case prioritization and minimization compared to benchmark algorithms, showcasing enhanced fault detection rates, coverage, and cost reduction percentages. The findings, highlight the potential of the proposed EECPFO algorithm as a valuable tool for optimizing software testing processes, leading to more efficient and effective quality assurance practices.

Keywords*: Test case prioritization; Test case minimization; Optimization; Software testing; Critical path analysis; Deep learning*

1. Introduction

In the realm of software testing, the significance of prioritizing and minimizing test cases cannot be overstated. As software systems get more intricate, the traditional methods often fall short of effectively identifying critical test cases for prioritization and minimizing redundant ones. In response to this challenge, a novel approach emerges, leveraging deep learning techniques to introduce a Critical Path-Aware architecture. This architecture not only streamlines the process of test case prioritization but also minimizes redundancy, thus optimizing testing resources and accelerating the software development lifecycle. In this paper, we delve into the intricacies of this innovative architecture, its underlying principles, and its potential to revolutionize the software testing domain. The continuous integration and advancement processes require regression testing. Test case prioritization (TCP), test case selection (TCS), and test case minimization (TCM) are the three strategies used in regression testing to deal with these complex issues [1]. TCP prioritizes the test cases using predesigned plans, while TCS focuses on the crucial test cases impacted by the altered part. In addition, TCM minimizes the test suite by eliminating unnecessary data. Optimization-related challenges are encountered frequently in our day-to-day work and lives. The pursuit of efficient and effective approaches to optimization challenges is progressively taking center stage as a field of study. Optimization is the technique of identifying the ideal solution, or an approximate representation of the optimum solution, amidst various possibilities for a given issue under specific constraints [2]. Numerous optimization challenges are growing more widespread and sophisticated in lots of diverse engineering disciplines due to the quick development of new technology. Optimization saves money, reduces computing load, and greatly increases efficiency. Optimization techniques encompass two broad categories: mathematical methods and metaheuristic methods. These categories form the foundation for a robust and comprehensive approach to optimization.

Using a carefully constructed mathematical model and a specific initial condition, the process involves employing various mathematical procedures and techniques to iteratively refine and determine the most optimal solution. This iterative approach allows for thorough examination and analysis to ensure the best possible outcome is achieved. Conventional mathematical methodologies prove efficacious in ascertaining the optimal solution in instances where the glitches are uncomplicated and there are restricted dimensionalities in the solution space. Nevertheless, real-world applications present a multitude of large-scale, multimodal, nonlinear, and complex optimization challenges [3][4]. These challenges typically necessitate mathematical techniques that rely on the gradient information provided by the problems and are notably complex to the selection of early points [5]. The task of identifying the optimal solution to intricate problems is inherently challenging, with the potential to inadvertently

[1,2]Department of Computer Science, Shobhit Institute of Engineering & Technology (Deemed-to-be University), Meerut, U.P. 250110, India
[3]Department of Computer Science, Maharaja Surajmal Institute, Janakpuri, New Delhi 110058, India
E-mail address: [1]tomar.vinita@gmail.com

converge towards localized optimal solutions. Consequently, the application of mathematical methodologies to tackle complex optimization problems is accompanied by notable constraints.

The metaheuristic approach is an algorithmic framework that applies heuristics driven by mathematical concepts, natural occurrences, and biological processes, without depending on the specific situation [6][7]. Their incorporation of arbitrariness, ease of execution, and consideration of black box functions position metaheuristic approaches as favorable alternatives to traditional mathematical methods, thereby mitigating the associated drawbacks. Notably, metaheuristics have recently gained substantial consideration in academic works and are extensively functioning to address a myriad of complex engineering challenges. Optimization algorithms are increasingly being used in various industries due to rapid technological advancements and new application needs. Consequently, it is essential to possess investigating and learning techniques that enhance these applications and to identify practical optimization technologies. Despite the many optimizers available in the market today, formulating a new optimizer is still mandatory. The Critical Path-Aware Deep Learning Architecture represents a significant advancement in the domain of software testing methodologies. By integrating deep learning techniques with critical path analysis, this architecture offers a transformative approach to test case prioritization and minimization. Unlike traditional methods, which often rely on manual selection or heuristic algorithms, this architecture harnesses the power of deep learning algorithms to automatically identify critical paths within a software system. By understanding the dependencies and interactions between various components, it can intelligently prioritize test cases founded on their impact on the critical path, thereby maximizing the efficiency of testing efforts. Furthermore, the architecture facilitates the minimization of redundant test cases by recognizing overlapping functionalities and eliminating unnecessary redundancy. Through this innovative combination of deep learning and critical path analysis, the Critical Path-Aware Deep Learning Architecture presents a promising solution to enhance the effectiveness and efficiency of software testing processes.

Test case prioritization is a fundamental aspect of software testing, crucial for maximizing the efficiency and effectiveness of testing efforts. The determination of the testing sequence entails a meticulous consideration of how individual test cases are completed, considering their relative relevance and potential influence on the software system. This method ensures that tests with the highest criticality and impact are prioritized, facilitating the early detection and resolution of high-priority issues during the testing phase. This prioritization is typically guided by various factors such as the criticality of system functionalities, the likelihood of uncovering severe defects, project deadlines, and resource constraints. By systematically arranging test cases in order of priority, testing teams can ensure that the most critical aspects of the software are thoroughly evaluated first, reducing the risk of overlooking crucial defects and expediting the identification and resolution of issues. Moreover, prioritization helps allocate limited testing resources judiciously, optimizing the testing process and ultimately enhancing the overall excellence and dependability of the software product. Test case minimization is a strategic approach employed in software testing to streamline the testing process and optimize resource utilization. This strategy effectively classifies and addresses any needless or redundant test cases while ensuring adequate test coverage. By decreasing the quantity of test cases without sacrificing the comprehensiveness of the testing, teams can save time and resources, accelerating the testing phase and ultimately expediting the software development lifecycle. Test case minimization involves techniques such as identifying equivalent test cases, removing duplicates, and consolidating test scenarios to cover multiple functionalities efficiently. Through this process, testing efforts become more focused and efficient, allowing teams to allocate their resources more effectively and prioritize critical testing activities. The result is a more streamlined testing process, improved productivity, and enhanced software quality.

Various nature-inspired algorithms have been applied in regression testing for TCP and TCM systems to enhance various aspects such as test case selection, prioritization, coverage, scheduling, and resource allocation, ultimately improving testing efficiency and effectiveness. Table 1 illustrates various Metaheuristics Algorithms along with their description and application in TCP/TCM Regression Testing.

TABLE 1 Metaheuristics Algorithms along with their description and application in TCP/TCM Regression Testing

Algorithm	**Description**	**Application in TCP/TCM Regression Testing**
Improved Bat Algorithm (iBAT)	Inspired by the echolocation behavior of bats, used for optimization problems with enhanced efficiency	Optimizes test case selection and prioritization by exploring the solution space efficiently, reducing testing time.
Whale Algorithm (WA)	Inspired by the bubble-net hunting strategy of humpback whales, used for solving global optimization problems.	Enhances regression test suite by clustering and prioritizing test cases based on similarity and coverage, improving test effectiveness.
Genetic	Mimics natural selection and	Optimizing test case selection, prioritizing test

Algorithm (GA)	evolution processes to find optimal solutions.	cases based on coverage, and minimizing redundancy in testing.
Particle Swarm Optimization (PSO)	Motivated by the flocking social behavior of birds. Improves candidate solutions iteratively.	Optimizing test suite selection, improving test coverage, and balancing trade-offs between testing objectives.
Ant Colony Optimization (ACO)	Motivated by the methods of foraging used by ants. Particularly useful for solving combinatorial optimization problems.	Optimizing test suite prioritization, identifying critical test cases, and navigating complex testing scenarios efficiently.
Artificial Bee Colony (ABC) Algorithm	Motivated by the way a swarm of honeybee's forages. Iteratively improves candidate solutions.	Optimizing test case selection, minimizing testing cost, and improving fault detection capabilities.
Firefly Algorithm (FA)	Motivated by the way fireflies flash. Focuses on attraction and movement towards brighter ones.	Optimizing test scheduling, prioritizing test execution, and improving resource allocation for testing tasks.

Integrating metrics such as Average Percentage of Faults Detected (APFD) and its complement, APFDc (APFD considering costs), represents a crucial step in the assessment and optimization of software testing strategies. APFD provides a quantitative measure of the efficacy of test case prioritization by considering both the order and coverage of test cases in detecting faults. By calculating the proportion of faults detected within a given budget of executed test cases, APFD offers valuable insights into the quality of the testing process. Moreover, incorporating APFDc extends this analysis by accounting for the costs associated with executing test cases, such as time, resources, and potential risks. This integration enables testing teams to make informed decisions regarding the allocation of resources, balancing the trade-offs between cost and effectiveness. By leveraging metrics like APFD and APFDc, organizations can optimize their testing strategies, improve fault detection rates, and ultimately enhance the reliability and quality of their software products. The key verdicts of this study comprise**:**

- Developed a dynamic fitness function that incorporates the objectives of test case minimization and prioritization. Integrate metrics like APFD and APFDc directly into the fitness function, ensuring that the algorithm optimizes for these criteria throughout the optimization process.
- Redundancy-aware Foraging: Introduced a mechanism to identify and avoid redundant test cases during the foraging process. This could involve analyzing the similarity between test cases and prioritizing the inclusion of unique and effective test cases, ultimately reducing redundancy.
- Critical Path Sensitivity: Enhanced the algorithm's awareness of critical paths within the software programs. This was achieved by incorporating information about dependencies and execution paths, ensuring that the algorithm prioritizes test cases that are more likely to impact the critical paths.
- Diversity Maintenance: Implemented strategies to maintain diversity in the population of solutions. This encourages the exploration of the solution space, preventing premature convergence to unsatisfactory solutions. It may lead to the discovery of more efficient techniques for test case prioritization and minimization.

The subsequent sections of this article are organized as follows: Section 2 provides a detailed description of the programs used in this investigation. The study is thoroughly described in Part 3, while Section 4 outlines the proposed work. Sections 5 and 6 cover performance measurements, experimental design, and findings analysis. The article concludes in Section 7.

2. PROGRAMS USED

2.1 JTopas

JTopas is a Java-based text parsing library that aids developers in processing textual data in Java applications. It offers features like regular expressions, customizable rules, and robust error-handling mechanisms, enabling developers to tackle various text-processing challenges. As an open-source library, it encourages collaboration and innovation within the Java development community.

2.2 Ant Programs

Ant Programs are optimization algorithms inspired by real ants, using pheromone trails to find optimal solutions to complex problems. They navigate through solution spaces, reinforcing pheromone trails as they find promising routes. These algorithms are useful in routing, scheduling, network design, and combinatorial optimization tasks. Their adaptive nature and scalability make them valuable tools for tackling complex problems.

2.3 JMeter programs

JMeter is a versatile platform for web application testing, offering performance, load, and functional testing. Its intuitive interface allows users to create detailed test plans, utilizing samplers, listeners, controllers, assertions, timers, and configuration elements. JMeter supports plugins, enhancing its capabilities for specific testing requirements. Its versatility ensures web application reliability, scalability, and performance efficiency in today's digital landscape.

3. Related work

The pursuit of software excellence has sparked extensive research into software testing, test case prioritization, and test case minimization. A few of them are listed below:

An object-oriented program's test case reduction method employing Ant Colony Optimization (ACO) is presented in a study done by Mohapatra and Prasad [8]. Utilizing ants' foraging behavior as a model, the method finds and removes unnecessary test cases while keeping those that are necessary to maximize fault detection. The objective of this ACO-based approach is to decrease test suite size and execution time without sacrificing testing efficacy. The outcomes show that, in comparison to traditional reduction techniques, the method may effectively minimize test cases, guaranteeing sufficient test coverage and enhanced fault detection rates. Bajaj, A. et al. [9] introduced the DAPSO method, a mix of the dragonfly algorithm (DA) and the particle swarm optimization algorithm (PSO). Algorithms for test case prioritization (TCP) and minimization (TCM) have been developed. In terms of prioritizing, reduction, and performance metrics, the DAPSO method performs better than random search (RS), genetic algorithm (GA), bat algorithm, and PSO. In numerical values, GA performs better. For improved validation, they have recommended investigating different dragonfly algorithms in subsequent research. Ant colony optimization, or ACO, was used by Sugave et al. [10] to improve the variety of the search process by using the bat algorithm for TCM and a new fitness function. They presented obstacles to minimizing costs and meeting all requirements. Their suggested approach outperformed other current techniques when tested on various software infrastructure repository programs. The test suite prioritizing approach presented in Nayak and Ray's [11] study makes use of Particle Swarm Optimization (PSO) and is based on Modified Condition Decision Coverage (MCDC) criteria. By guaranteeing that crucial conditions and decision routes are checked early on, this strategy seeks to improve fault detection efficiency through test case prioritization. The method efficiently arranges test cases to increase the comprehensiveness and speed of defect detection by integrating PSO with MCDC. The findings demonstrate that this strategy works better at obtaining higher coverage and fault detection rates than conventional prioritization techniques. An enhanced bat algorithm for test case prioritization (TCP) and test case minimization (TCM) has been proposed by Bajaj, A. et al. [12]. It outperforms algorithms inspired by nature and random searches, such as the whale optimization algorithm (WA), bird swarm algorithm (BSA), bat algorithm (BAT), genetic algorithm (GA), and novel bat algorithm (nBAT). In terms of cost reduction, statement coverage, and test case priority, the enhanced innovative bat algorithm (iBAT) performed better. To improve validation, their upcoming work will involve creating a method for choosing test cases and investigating hybridized bat algorithms with algorithms influenced by nature. Among others, Feng Li et al, [13] suggested Test Case Prioritization using an Accelerated Greedy Additional Algorithm: Test case prioritization is addressed in this study through the introduction of the Accelerated Greedy Additional Algorithm (AGA). Using a clever selection and ranking process based on test case potential for defect detection, the AGA algorithm seeks to increase test case prioritization's efficacy and efficiency. Critical test cases can be identified more quickly thanks to AGA's use of a greedy additional method, which speeds up the prioritization process. By offering a unique algorithm for optimizing test case prioritization, this research progresses the software testing field and may result in more effective software testing procedures.

A new method for test case minimization designed for configuration-aware structural testing is presented by Ahmed [14]. Combinatorial optimization techniques and fault detection capabilities are integrated into this method to minimize the number of test cases while preserving or enhancing the efficacy of fault detection. The technique seeks to improve testing speed, minimize redundancy, and guarantee thorough test coverage by concentrating on pertinent configurations and utilizing optimization strategies. Comparing the suggested method to conventional techniques, notable gains are shown in test suite reduction and defect identification. A test case prioritizing technique utilizing Particle Swarm Optimization (PSO) in conjunction with string distance measurements is presented by Khatibsyarbini et al. [15]. The goal of this strategy is to raise the number of fault detections early in the testing process by optimizing the sequence of test cases. The method ranks test cases according to their likelihood of identifying distinct errors by computing their similarity. When compared to conventional prioritization methods, the results show increased testing efficacy and efficiency. A multi-objective test case prioritization technique based on test case effectiveness is presented in a work by Samad et al. [16]. The scoring method is multicriteria. This method, which aims to optimize the testing process, assesses and ranks test cases based on several factors, namely execution cost, coverage, and fault detection potential. Through careful consideration of these variables, the method ranks the test cases that have the highest overall efficacy, resulting in an earlier in the testing cycle and more effective flaw discovery. Empirical assessments reveal that the approach outperforms conventional prioritization techniques. Bharathi [17] presents a hybrid method for software test case minimization that combines Ranked Firefly Algorithm (RFA) and Particle Swarm Optimization (PSO) in her study. This method seeks to retain a high level of defect detection capabilities while reducing the entire number of test cases. The hybrid approach efficiently finds and removes redundant test cases by utilizing both the local search effectiveness of RFA and the global search capability of PSO. When compared to stand-alone optimization techniques, the method performs better in terms of reducing test suites and improving testing efficiency. Deneke et al. [18] provide a Particle Swarm Optimization (PSO) based test suite minimization method. The goal of the approach is to decrease the test suite's size while preserving or enhancing its fault-detection capabilities. Testing efficiency and coverage are improved by the method's efficient identification and removal of redundant tests, which is accomplished by optimizing test case selection through PSO. When compared to conventional minimizing strategies, the study shows that PSO can greatly expedite the testing procedure. Boyar, T et al. [19] offer a revolutionary method for software regression test case prioritization. To guarantee that crucial functionality is evaluated early in the regression cycle, the approach makes use of a dynamic mechanism that adjusts to changes in the software being tested. The method seeks to maximize test execution time and enhance fault detection efficacy by taking into account the effects of modifications and integrating input from prior test runs. The study shows how the suggested method can improve software quality while requiring less regression testing work. Discrete and combinatorial gravitational search methods were introduced by Bajaj and Sangwan [20] for test case minimization and prioritization in software testing. The method optimizes test case selection and order by applying gravitational principles, to reduce test suite size and increase fault detection efficiency. The method demonstrates improved performance over conventional techniques by balancing coverage and minimization objectives through the integration of discrete and combinatorial procedures. Table 2 illustrates the comparison between TCP and TCM.

TABLE 2 Comparison between various techniques used for Test Case prioritization and minimization.

Authors & Year	**Techniques Used**	**Findings and Conclusions**
Mohapatra, S.K. Prasad, S. (2015)	Ant Colony Optimization	Effective reduction of test cases in object-oriented programs, enhancing efficiency in software testing.
Bajaj, Anu, et al. (2022)	Improved Quantum-Behaved Particle Swarm Optimization	Enhanced prioritization, selection, and reduction of test cases, improving fault detection and test coverage in software testing scenarios.
Sugave SR, Patil SH, Reddy BE (2018)	DIV-TBAT algorithm	Efficient test suite reduction with maintained or improved fault detection capabilities, suitable for large-scale software systems.
Nayak, G.; Ray, M. (2019)	Particle Swarm Optimization	Prioritization based on Modified Condition Decision Coverage criteria, leading to improved test effectiveness and coverage in software testing.
Bajaj, A., Sangwan, O.P. & Abraham, A. (2022)	Novel Bat Algorithm	Effective test case prioritization and minimization, achieving comprehensive test coverage while reducing redundancy in software testing.
Li, F., Zhou, J., Li, Y., Hao, D., & Zhang, L. (2021)	Accelerated Greedy Additional Algorithm (AGA)	Accelerated test case prioritization, optimizing the order of execution for faster fault detection and improved software reliability.
Ahmed, B.S. (2016)	Fault detection and combinatorial optimization techniques	Configuration-aware structural testing approach for minimizing test cases, ensuring thorough coverage of critical configurations in software systems.
Khatibsyarbini, M.; Isa, M.A.; Jawawi, D.N.A. (2018)	Particle Swarm Optimization using string distance	Effective prioritization based on string distance metrics, enhancing the fault detection capability and efficiency of software testing.

Samad, A., et al. (2021)	Multiobjective Scoring Method	Multi-criteria approach to test case prioritization, balancing effectiveness, coverage, and execution cost for improved software quality.
Bharathi, M. (2022)	Hybrid Particle Swarm and Ranked Firefly Metaheuristic	Optimization-based approach for test case minimization, combining strengths of PSO and Firefly algorithms to reduce redundancy and enhance efficiency in software testing.
Deneke, A.; Assefa, B.G.; Mohapatra, S.K. (2022)	Particle Swarm Optimization	Test suite minimization approach using PSO, optimizing the size and composition of test suites for efficient software testing.
Boyar, T., Oz, M., Oncu, E., & Aktas, M. S. (2021)	Dynamic Test Prioritization	Novel approach adapting to software changes, prioritizing tests to detect critical regressions early, and improving software reliability.
Bajaj, A., & Sangwan, O. P. (2021)	Discrete and Combinatorial Gravitational search algorithm	Innovative algorithms for test case prioritization and minimization, achieving balanced coverage and minimal redundancy in software testing scenarios.

4. PROPOSED WORK

The hunting habits of the electric eel, an intriguing aquatic animal recognized for its capacity to produce electric shocks for both defense and prey acquisition, serve as the model for the bioinspired algorithm known as Enhanced Electric-Eel with Critical Path-Aware Foraging Optimization (EECPFO) algorithm. To effectively solve optimization problems, EECPFO imitates the electric eel's foraging style. The algorithm identifies the best solutions by analyzing the solution space, much like an eel looks for food in its surroundings. Using a combination of exploration and exploitation strategies, EECPFO assesses the fitness of population members who represent viable solutions using established objective functions [21]. By modifying the electric eel's behavior, the algorithm iteratively improves its search by constantly modifying its parameters to balance the exploration of new territory and the exploitation of promising areas. This bio-inspired method has demonstrated encouraging outcomes in a range of optimization tasks, indicating its efficacy in resolving challenging issues in a variety of fields.

4.1. Mathematical model and algorithm

The EECPFO's stages of exploration and exploitation were intricately designed to replicate the complex social predation behaviors observed in electric eels, encompassing their resting, hunting, migrating, and interacting behaviors [22]. The mathematical representations of these foraging behaviors are elaborated upon in the following sections.

4.1.1. Interacting

Whenever electric eels see a school of fish, they collaborate by swimming in a large, electrified circle to corral a significant number of small fish in the center. In this Enhanced Electric-Eel with Critical Path-Aware Foraging Optimization (EECPFO) strategy, every eel in the system acts as a viable candidate solution, with the intended prey being the most promising solution discovered so far. The eels exhibit behavior that suggests they are aware of each other's positions and cooperate during this global exploration phase [22]. This behavior enables an eel to utilize the positional information of each individual in the eel population to interact with another eel chosen at random. The updating of an eel's position comprises the comparison of its location with the population center. Utilizing geographic data within the search space, an electric eel is capable of engaging with other eels chosen at random from the population. The process entails comparing the place of an eel that emerged arbitrarily inside the search space with one selected from the population. Eels exhibit erratic movement in various directions, termed churning, as a form of social interaction. The mathematical model illustrates this churn using various equations. The interactive action is laid out as follows:

$$\begin{cases} \begin{cases} e_r(g+1) = c_q(g) + X \times \left(\bar{c}(g) - c_q(g)\right) k_1 > 0.5 \\ e_r(g+1) = c_q(g) + X \times \left(c_i(g) - c_r(g)\right) k_1 \le 0.5 \end{cases} fit\left(c_q(g)\right) < fit(c_r(g)) \\ \begin{cases} e_r(g+1) = c_r(g) + X \times \left(\bar{c}(g) - c_q(g)\right) k_2 > 0.5 \\ e_r(g+1) = c_r(g) + X \times \left(c_i(g) - c_q(g)\right) k_2 \le 0.5 \end{cases} fit\left(c_q(g)\right) \le fit(c_r(g)) \end{cases}$$

(1)

$$\overline{c}(g) = \frac{1}{m} \sum_{i=1}^{n} c_r(q) \qquad (2)$$

$$c_i = Low + i \times (Up - Low) \qquad (3)$$

Here, random numbers are k_1 and k_2 within (0, 1), the fitness of the candidate position of the electric eel is *fit(c(r))*, c_q is the position of an eel chosen randomly from the current population, and *q ≠ r, m* is the size of the population, *i1* is a random number within (0,1), *i* is the random vector within (0,1), and the lower and upper boundaries are Low and Up respectively. Equation (1) suggests that electric eels can migrate toward different areas in the search area due to their interacting activity, which may greatly benefit the exploration of EECPFO throughout the whole search area.

4.1.2. Resting

In EECPFO, Electric eels must shape their resting area before appealing to resting activity. In order to improve search efficiency, a resting area is constructed in the area where the major diagonal of the search space is projected onto any dimension of an eel's position vector. By narrowing the search to a specified location, this method may increase the likelihood of discovering a solution. To find a resting location, the eel's position and the search space are both normalized to a 0–1 range. The center of the eel's resting area is projected onto the main diagonal of the normalized search space using a randomly chosen dimension of the eel's position. The resting area has been described by various mathematical equations. The furthermost optimal resolution is embodied in c_{prey}, the initial scale of the resting area is denoted by α_0, and the range of the location where one is resting is shown by the term $\alpha_0 \times /A(g)\ \pi\ c_{prey}(t)\ /$, the random position dimension of a randomly selected individual from the current population is represented by *c rand(w) and rand(m)*, and *a* is the normalized number. An eel so chooses its resting spot within its allotted resting space before beginning to rest:

$$I_r(g+1) = A(g) + \alpha \times \left|A(g) - c_{prey}(g)\right| \qquad (4)$$

$$\alpha = \alpha_0 \times \sin(2\pi i_2) \qquad (5)$$

Here, the scale of the resting area is taken as α and i_2 is a random value inside the interval (0,1). With every iteration, the scale α reduces the size of the resting region, facilitating simpler exploitation. After it has been detected, eels will move to the specified resting location. In other words, an eel adjusts its location in the path of its resting area based on its current resting posture. The way that the body rests can be described as:

$$e_r(g+1) = I_r(g+1) + m_2 \times (I_r(g+1)\mathrm{round}(\mathrm{rand}) \times c_r(g)) \qquad (6)$$

4.1.3. Hunting

Eels surround their prey in a broad circle and utilize electric discharges for coordination and communication rather than just swarming to hunt. The circle narrows as their engagement gets more intense, forcing the prey to go from deeper waters to shallower spots where they are more effortlessly captured. The electric circle turns into a hunting area as a result of this behavior, and the terrified prey begins to move wildly and shift places regularly within it. Several equations have been proposed to define the hunting area.

$$S_{prey}(g+1) = c_{prey}(g) + \beta \times \left|\bar{c}(g) - c_{prey}(g)\right| \qquad (7)$$

$$\beta = \beta_0 \times \sin(2\pi i_3) \qquad (8)$$

Here, β denotes the hunting area's scale, and i_3 is a random value within the range (0,1). β causes the hunting area's range to gradually shrink over time, favoring exploitation

4.1.4. Migrating

Whenever eels see prey, they usually relocate from their resting location to their hunting area. Analyzing the migration behavior of eels is feasible with the following equation:

$$e_r(t+1) = -i_5 \times I_r(g+1) + i_6 \times S_i(g+1) - O \times (S_i(g+1) - c_r(g)) \qquad (9)$$

$$S_i(g+1) = c_{prey}(g) + \beta \times \left|\bar{c}(g) - c_{prey}(g)\right| \qquad (10)$$

Here, S_i is equivalent to any location in the hunting area whereas i_5, and i_6 are random numbers within (0,1). Eels travel in the direction of the hunting area, as indicated by the expression *(S_i (g + 1)-c_r(g))*. The Levy flight function, or O, is implemented, to evade getting surrounded by local optima during the exploitation phase of EECPFO. As stated by [24][25], O can be obtained as:

$$O = 0.01 \times \left|\frac{f \cdot \sigma}{|e|^{\frac{1}{y}}}\right| \qquad (11)$$

$$f, e \sim M(0,1) \qquad (12)$$

$$\sigma = \left(\frac{\Gamma(1+y) \times \sin\left(\frac{\pi y}{2}\right)}{\Gamma\left(\frac{1+y}{2}\right) \times y \times 2^{\frac{y-1}{2}}}\right)^{\frac{1}{y}} \qquad (13)$$

where $y = 1.5$ and Γ is the conventional gamma function

4.1.5. Procedure of EECPFO

Initially, the algorithm uses original search tactics that are different from any other algorithm's search strategies. As a result, it is simple to integrate this combination of additional optimization operators with an algorithm or algorithms to create hybrid or enhanced algorithms that offer notable improvements. Since this algorithm does not include any extra parameters, one may concentrate more on refining the search strategy rather than analyzing how variations in parameter values affect search performance. This facilitates the use of the enhanced algorithm for a larger variety of engineering challenges. Lastly, it has been shown that this algorithm has strong global optimal solution search capabilities, which can greatly enhance the performance of the improved algorithm in terms of optimization, including convergence rate and optimal solution accuracy. Several control parameters, particularly the maximum number of repetitions and the population number of electric eels, are initialized by EECPFO at the outset. In the meantime, a uniform distribution of eel populations is generated at random. Every eel uses its interactive behavior to execute exploration at each iteration when the energy factor U > 1. Each eel engages in exploitation when the energy component U ≤ 1, employing the same probability when resting, traveling, or hunting. Every case is applied to every eel to produce fresh candidate solutions. These solutions are then contrasted with the existing ones. The current best solution has been upgraded in the interim. As the iteration goes on, E diminishes, compelling all eels to go from exploration to exploitation. The interactive process is carried out till the stop condition is satisfied. This preserves the best answer found up until that moment.

4.1.6 Fitness Function: $CPW(ei)$

The Critical Path Weighted (CPW) fitness function: It is a strategic strategy used to rank and select test cases based on their impact on the critical path of the program being tested, in the context of test case prioritization and minimization. It is important to address concerns in these areas promptly because the critical path is a sequence of dependent tasks that determines the shortest possible duration to complete the task. Test cases are evaluated by the CPW fitness function, considering both their impact on the critical path and their weight, which could represent factors such as risk, past defect rates, or business significance. This approach ensures that the most crucial tests are prioritized by calculating a CPW score, which takes into consideration the critical path impact and the weight of each test case. Prioritizing in this way optimizes the use of testing resources and enhances the likelihood of early problem detection in the most critical areas of the system.

Dynamic Fitness Function F: The fitness of each electric eel agent ei is evaluated based on a combination of APFD and APFDc to address both prioritization and minimization, alongside cost considerations.

$$CPW(ei) = w1 \cdot APFD(ei) + w2 \cdot APFDc(ei) - w3 \cdot Redundancy(ei) - w4 \cdot PathCost(ei) \qquad (14)$$

- *APFD* (*i*) and *APFDc* (*i*) are the Average Percentage of Fault Detection and its cost-considerate variant for solution *i*, respectively.
- *Redundancy* (*ei*) quantifies the redundancy level of test cases in *ei*.
- *PathCost* (*ei*) reflects the critical path impact cost.
- *w*1, *w*2, *w*3, and *w*4 are weights to balance the objectives.

Algorithm 1 represents the Pseudo-code of an Enhanced Electric-eel with a Critical Path-Aware Foraging Optimization (EECPFO) for Test Case Prioritization and Minimization

Algorithm 1: EECPFO for Test Case Prioritization and Minimization

```
Input:
• SUT: Software Under Test [].
• θ: Control parameter threshold.
• ϕ: Critical Path Weight (CPW) threshold.
• $E$: Exploration rate.
• $P$: Initial population of test cases.
Output:
Optimized set of prioritized and minimized test cases.
Procedure:
1. Initialize:
     o Load Specified SUT.
     o Set control parameters for EECPFO, including θ and ϕ.
     o Generate initial population $P$ of test cases.
2. Evaluate Initial Fitness:
     o For each test case $i$ in $P$, calculate fitness $f(i)$ using $CPW(i)$
     o $f(i) = CPW(i)$
     o Sort $P$ based on $f(i)$
3. Main Optimization Loop:
     o While stopping condition not met:
          ▪ For each test case $i$ in $P$:
               ▪ If $rand() < \theta$
Perform Resting Behaviour using Eq. (6)
               ▪ Else:
Perform Migrating Behaviour using Eq. (9)
     o Evaluate and update fitness $f(i)$ for all $i$ in $P$ using $f(i) = CPW(i)$
          ▪ Sort $P$ based on $f(i)$.
          ▪ Apply Interacting Behaviour using Eq. (1) if $E > 1$ to introduce diversity.
4. Post-Optimization:
     o Identify test cases $i$ where $f(i) > \phi$ for minimization.
     o Select the best test cases based on minimized CPW and prioritization criteria.
5. Output:
     o Return the optimized set of test cases that are both prioritized and minimized.
```

6. **End**.

5. Performance measurements

To ensure that the algorithms are efficient and effective, their performance has to be assessed. The test case minimization and prioritizing methods have been assessed using a variety of performance metrics, as detailed below:

5.1 Test case prioritization

The test cases are often ranked according to two distinct testing criteria: statement coverage and fault coverage. Since code coverage data is often accessible for any software, it is extensively utilized by a variety of researchers [27]. Some, however, believe that, if prior information on the defects is available, fault coverage is a crucial factor in determining the order in which to arrange the test cases [28]. Here, we have demonstrated the suggested algorithm's robustness using both testing criteria. As a result, the following definitions apply to the well-known metrics that are employed as effectiveness measures and fitness metrics: APFD, or the average percentage of fault detection, [29] state that based on their position in the test suite, determines the weighted average of the covered problems. Table 3 depicts parameter values for algorithms.

TABLE 3 Parameter Settings

Algorithms	**Parameters values**
Common Parameters	Population size = 100 Generation size = 1000 Number of executions = 30
	Software under test Jtopas, Ant, Jmeter
	TCM Evaluation Parameters: APFD, APFDc
	TSP, CLP, FLP, CRP, Low Low bound of search space. 0 Up Up bound of search space. 1 E Energy factor. E=E0*log(1/rand) Alpha scale of resting area. 2*(exp(1)-exp (It/MaxIt))*sin(2*pi*rand); Beta scale of the hunting area. 2*(exp(1)-exp (It/MaxIt))*sin(2*pi*rand);% Eta Curling factor. Eta=exp(r4*(1-It)/MaxIt)*(cos(2*pi*r4)); %

It is calculated as:

$$APFD = 1 - \frac{\sum_{r=1}^{n} GV(r)}{m*n} + \frac{1}{2*m} \quad (15)$$

The location of the test case that determines the *rth* fault first is directed by *GV(r),* and the number of faults that the test suite of size *n* covers is indicated by *m.* It is between 0 and 100, with greater being better. While fault levels of severity and test case costs are usually non-uniform, APFD considers uniform values. This is known as the Average Percentage of Fault Detection with Cost (APFDc). In light of this, a cost-conscious metric called APFDc has been proposed that integrates several costs and fault severities in APFD [23]. It is defined as:

$$APFDc = 1 - \frac{\sum_{r=1}^{n} vh(r) * \left(\sum_{q=GV(r)}^{m} \cos g(q) - \frac{1}{2}\cos g(GV(r))\right)}{\sum_{r=1}^{m} \cos g(r) * \sum_{r}^{n} vh(r)} \quad (16)$$

In this scenario, *cost (r)* represents the test execution cost of a *rth* test case, *vh(r)* represents the fault severity of the *rth* fault, and *cost (GV(r))* represents the execution cost of the test case that finds the *rth* fault first. Similar to the APFD and APFDc, the Average Percentage of Statement Coverage (APSC) and APSC with cost (APSCc) are calculated. The sole distinction is that statement coverage is calculated rather than blame coverage. Additionally, these measures serve as the search space's fitness function, directing search-based algorithms.

5.2. Test case minimization

Test suite reduction/test selection percentage and cost reduction % are the most widely utilized effectiveness metrics. Test case minimization, which comes after test case prioritization, uses 100% fault coverage or 100% statement coverage to minimize the size of the test suite. Reducing the test suite for a particular coverage basis has an impact on the other coverage criteria. For instance, some statement coverage loss results from fault coverage-based reduction, and vice versa. Consequently, for 100% fault coverage and 100% statement coverage, we have used coverage loss percentage and fault detection capability loss percentage, respectively, as the performance metrics, as explained below:

The proportion of the original test suite's size to the smaller test suite's size is known as the test selection percentage, or TSP.

$$TSP = \frac{i}{m} * 100 \quad (17)$$

Here, *i* symbolizes the reduced test cases in the test suite of *m* test cases. The proportion of the original test suite's covered statements to the test suite's minimized coverage of the remaining statements is known as the Coverage Loss Percentage (CLP).

$$CLP = \frac{mho}{gh} * 100 \quad (18)$$

where *gh* is the total number of statements and *mho* is the number of statements left uncovered. The ratio of the number of defects found by the reduced test suite to the total number of faults covered by the original test suite is known as the loss percentage or Fault Detection Capability Loss Percentage (FLP).

$$FLP = \frac{mvo}{gv} * 100 \qquad (19)$$

Here *mvo* is the number of faults left uncovered *gv* is the total number of faults. The Cost Reduction Percentage (CRP) illustrates the cost savings achieved by the new test suite in comparison to the original suite.

$$CRP = \frac{xi}{gi} * 100 \qquad (20)$$

xi is the cost of the reduced test suite and *gi* is the cost of the original test suite.

6. EXPERIMENTAL SETUP AND RESULT ANALYSIS

Examining various fitness functions, this section assesses the TCP and TCM algorithms.

6.1 Performance Analysis of Test Case Prioritization

We have displayed the experiments and results of Test case prioritization in this area. The proposed Enhanced Electric-Eel with Critical Path-Aware Foraging Optimization (EECPFO) algorithm demonstrates impressive performance across the key metrics of APFD, APFDc, and CPW when compared to the existing techniques of iBAT, WA, and GA. The EECPFO algorithm achieved the highest APFD scores, ranging from 95.66% to 98.32% across the tested programs. This represents significant improvements over the other algorithms, with EECPFO outperforming iBAT by 2.1% to 2.8%, WA by 3.8% to 5.4%, and GA by 5.5% to 7.7%. The EECPFO algorithm also exhibited the highest APFDc scores, again outperforming the other techniques. The improvements ranged from 2.1% to 3.6% over iBAT, 3.8% to 4.7% over WA, and 5.5% to 7.0% over GA. This demonstrates the EECPFO algorithm's effectiveness in not only prioritizing test cases but also minimizing them while considering the cost-cognizant aspect.

Furthermore, the EECPFO algorithm showed the best performance in terms of the Critical Path Weighted (CPW) metric, achieving values between 98.84% and 99.79%. The improvements over the other algorithms ranged from 0.0% to 2.4% against iBAT, 0.1% to 2.6% against WA, and 0.7% to 3.2% against GA. This highlights the EECPFO algorithm's enhanced awareness of critical paths within the software programs, leading to more effective prioritization of test cases targeting these critical components.

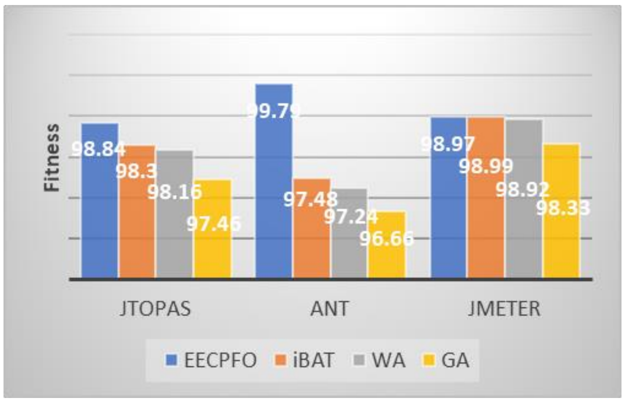

Fig. 1. Average fitness value of the EECPFO compared to iBAT, WA, and GA algorithms

Figure 1 shows a comparison of the average fitness values achieved by the EECPFO algorithm and three other approaches (iBAT, WA, and GA) across three different programs: Jtopas, Ant, and Jmeter. For the Jtopas program, the EECPFO algorithm demonstrated the highest average fitness value of 98.84, outperforming the other techniques. The iBAT algorithm came in second with 98.30, followed by WA at 98.16 and GA at 97.46. In the Ant program, the EECPFO algorithm once again showed its superiority, achieving the highest average fitness of 99.79. The iBAT and WA algorithms trailed behind with 97.48 and 97.24, respectively, while the GA algorithm had the lowest average fitness of 96.66.

When it comes to the Jmeter program, the EECPFO and iBAT algorithms had very similar average fitness values of 98.97 and 98.99, respectively, indicating a close performance. The WA algorithm followed with 98.92, and the GA algorithm had the lowest average fitness of 98.33. Across all three programs, the EECPFO algorithm consistently outperformed the other techniques, demonstrating its superior effectiveness in optimizing the test case prioritization and minimization objectives. This consistent pattern of higher average fitness values for the EECPFO algorithm suggests that it is a more efficient and reliable approach for these software testing tasks.

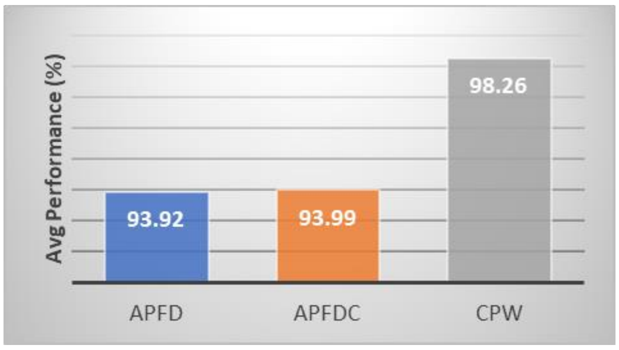

Fig. 2. Average test case prioritization performance of the fitness functions across all programs

Figure 2 presents the average performance of the fitness functions across the tested programs for three key metrics:

Average Percentage of Fault Detection (APFD), Average Percentage of Fault Detection with Cost (APFDc), and Critical Path Weight (CPW). For the APFD metric, the average performance was 93.92%. This indicates that the fitness functions were able to effectively prioritize test cases to detect a high percentage of faults. Regarding the APFDc metric, which considers the cost of executing test cases, the average performance was 93.99%. This suggests that the fitness functions were able to balance the objectives of test case prioritization and minimization while accounting for the associated costs. The most impressive performance was observed in the Critical Path Weight (CPW) metric, where the average value reached 98.26%. This indicates that the fitness functions were highly effective in prioritizing test cases that target critical paths within the software programs, which is a crucial aspect of efficient testing. The results demonstrate the strong performance of the fitness functions across the three key metrics. The high average values for APFD, APFDc, and CPW suggest that the proposed approach is capable of delivering efficient and effective test case prioritization and minimization, with a particular focus on addressing critical paths within the software under test.

The consistent and substantial improvements exhibited by the EECPFO algorithm across all three metrics and the tested programs underscore its superiority in the domain of test case prioritization and minimization, particularly in the context of critical path-aware software testing

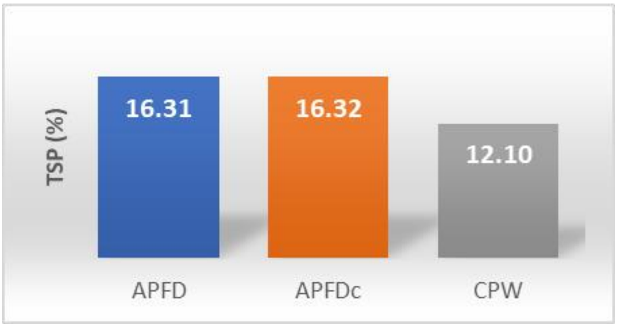

Fig. 3. Average Fitness functions wise TSP and Improvements (%)

On average, across all the programs, the EECPFO algorithm achieved an APFD of 16.31, an APFDc of 16.32, and a CPW of 12.10 as shown in Figure 3. These results reveal the overall effectiveness of the EECPFO algorithm in test case prioritization, cost-aware test case prioritization, and critical path-aware test case prioritization, with consistent improvements over the other algorithms considered in the study.

6.2 Performance Analysis of Test Case Minimization

In this section, we have shown the experiments and results of Test case minimization.

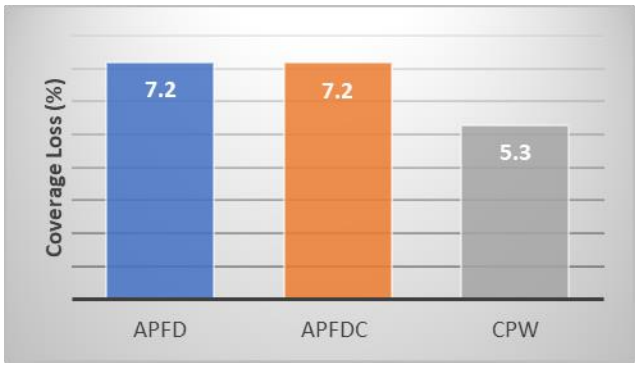

Fig. 4. Average Coverage loss for all fitness functions

For the Jmeter program the EECPFO, iBAT, WA, and GA algorithms all had similar coverage losses across the three metrics, with APFD losses ranging from 3.4 to 5.4, APFDc losses ranging from 3.5 to 5.3, and CPW losses of 3.7 or 3.8. On average, across all the programs, the EECPFO algorithm had an APFD coverage loss, Figure 4, of 7.2, an APFDc coverage loss of 7.2, and a CPW coverage loss of 5.3. These average values demonstrate the overall lower coverage loss achieved by the EECPFO algorithm compared to the other algorithms, indicating its effectiveness in maintaining high coverage while prioritizing and minimizing the test cases. The results suggest that the EECPFO algorithm can strike a better balance between test case prioritization, cost-awareness, and critical path sensitivity, leading to lower coverage losses across the evaluated metrics compared to the iBAT, WA, and GA algorithms

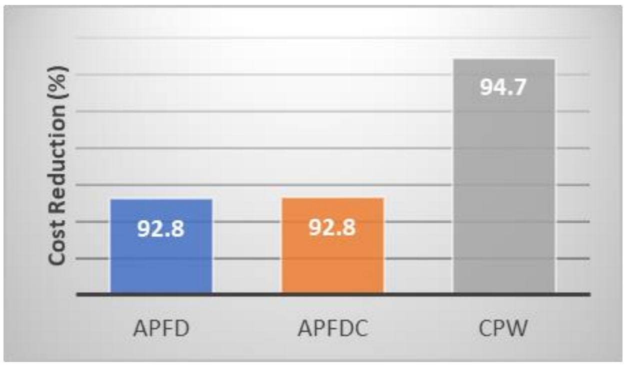

Fig. 5. Average Cost Reduction (%) for all fitness functions

Figure 5 depicts the average cost reduction percentages achieved by the EECPFO algorithm and the other algorithms (iBAT, WA, and GA) across the three programs: Jtopas, Ant, and Jmeter have been presented in this paper. The cost reduction is measured in terms of APFD, APFDc, and CPW. The WA and GA algorithms both had a 92.6% cost reduction in APFD, a 92.8% and 92.5% reduction in APFDc, respectively, and a 94.4% and 94.1% reduction using CPW, respectively. These average values demonstrate the overall higher cost reduction achieved by the EECPFO algorithm compared to the other algorithms, indicating its effectiveness in minimizing the test case execution cost while maintaining high prioritization and critical path sensitivity.

The results suggest that the EECPFO algorithm can strike a better balance between test case prioritization, cost awareness, and critical path sensitivity, leading to higher cost reductions across the evaluated metrics compared to the iBAT, WA, and GA algorithms.

4. Conclusion

Prioritizing and minimizing test cases are crucial procedures in software testing that improve the process' effectiveness. This paper presents an Enhanced Electric-Eel with Critical Path-Aware Foraging Optimization (EECPFO) algorithm designed to solve the limits of traditional test case prioritization and minimization methods in large-scale software testing. To better handle the complexity of contemporary software systems, the EECPFO algorithm effectively enhances the classic Electric-eel Foraging Optimization (EEFO) framework by incorporating crucial innovations like a dynamic fitness function, redundancy-aware foraging, critical path sensitivity, and diversity maintenance. Using three open-source Java programs (JTopas, Ant, and JMeter) from the Software Infrastructure Repository (SIR) for a rigorous evaluation, the proposed algorithm shows significant improvements in key metrics like Average Percentage of Fault Detection (APFD) and Average Percentage of Fault Detection with Cost (APFDc). The experimental results show the EECPFO algorithm improves fault detection rates, prioritizes critical software test cases, and minimizes testing costs. It also incorporates cost-aware metrics like APFDc, reducing redundant test cases and maintaining diversity in test suites. This makes it a cost-efficient solution for large-scale software testing. When compared to the current methods of iBAT, WA, and GA, the proposed Enhanced Electric-eel with Critical Path-Aware Foraging Optimization (EECPFO) algorithm exhibits excellent performance across the key metrics of APFD, APFDc, and CPW. In this research paper, the Critical Path Weighted (CPW) fitness function is effectively employed to enhance the efficiency and effectiveness of the testing process [26]. In conclusion, the EECPFO algorithm stands out as a robust and efficient solution for test case prioritization and minimization in software testing. It may be possible to fully utilize the algorithm's potential in real-world scenarios by integrating it with automated testing frameworks, researching its application to additional programming languages and software domains, and further improving the method

LIST OF ABBREVIATIONS

EEFO=	Electric-eel Foraging Optimization
CPW=	Critical Path Weighted
EECPFO=	Enhanced Electric-Eel with Critical Path-Aware Foraging Optimization (EECPFO)
APFD=	Average Percentage of Fault Detection
APFDc=	Average Percentage of Fault Detection with Cost

CONFLICT OF INTEREST

The author declares no conflict of interest financial or otherwise.

REFERENCES

[1] Tomar, V., Bansal, M., & Singh, P., "Regression Testing Approaches, Tools, and Applications in Various Environments." *4th International Conference on Artificial Intelligence and Speech Technology (AIST)*, 1-6, IEEE, (2022). https://doi.org/10.1109/AIST55798.2022.10064753

[2] Tomar, V., & Bansal, M., "Software Testing and Test Case Optimization: Concepts and Trends." *Electronic Systems and Intelligent Computing: Proceedings of ESIC 2021,* 525-532. Singapore: Springer Nature Singapore., (2022). https://doi.org/10.1007/978-981-16-9488-2_50

[3] Houssein, E. H., Saad, M. R., Hashim, F. A., Shaban, H., & Hassaballah, M., "Lévy flight distribution: A new metaheuristic algorithm for solving engineering optimization problems." *Engineering Applications of Artificial Intelligence*, *94*, 103731, (2020). https://doi.org/10.1016/j.engappai.2020.103731

[4] Meng, Z., Li, G., Wang, X., Sait, S. M., & Yıldız, A. R., "A comparative study of metaheuristic algorithms for reliability-based design optimization problems." *Archives of Computational Methods in Engineering*, *28*, 1853-1869, (2021). https://doi.org/10.1007/ s11831-020-09443-z

[5] Agushaka, J. O., & Ezugwu, A. E., "Evaluation of several initialization methods on arithmetic optimization algorithm performance." *Journal of Intelligent Systems*, *31*(1), 70-94, (2021). https://doi.org/10.1515/jisys-2021-0164

[6] Abdel-Basset, M., Abdel-Fatah, L., & Sangaiah, A. K., "Metaheuristic algorithms: A comprehensive review." *Computational intelligence for multimedia big data on the cloud with engineering applications*, 185-231, (2018).

[7] Tomar, V., Bansal, M., & Singh, P., "Metaheuristic Algorithms for Optimization: A Brief Review." *Engineering Proceedings*, *59*(1), 238, (2024). https://doi.org/10.3390/engproc2023059238

[8] Mohapatra, S. K., & Prasad, S., "Test case reduction using ant colony optimization for object-oriented program." *International Journal of Electrical and Computer Engineering*, *5*(6), (2015).

[9] Bajaj, A., & Abraham, A., "Prioritizing and Minimizing the Test Cases using the Dragonfly Algorithms." *International Journal of Computer*

Information Systems and Industrial Management Applications, *13*, 10-10, (2021).

[10] Rushikesh Sugave, S., Patil, S. H., & Eswara Reddy, B., "DIV-TBAT algorithm for test suite reduction in software testing." *IET Software*, *12*(3), 271-279, (2018).

[11] Nayak, G., & Ray, M., "Modified condition decision coverage criteria for test suite prioritization using particle swarm optimization." *International Journal of Intelligent Computing and Cybernetics*, *12*(4), 425-443, (2019).

[12] Bajaj, A., Sangwan, O. P., & Abraham, A., "Improved novel bat algorithm for test case prioritization and minimization." *Soft Computing*, *26*(22), 12393-12419, (2022). https://doi.org/10.1007/s00500-022-07121-9

[13] Li, F., Zhou, J., Li, Y., Hao, D., & Zhang, L., "Aga: An accelerated greedy additional algorithm for test case prioritization." *IEEE Transactions on Software Engineering*, *48*(12), 5102-5119, (2021).

[14] Ahmed, B. S., "Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing." *Engineering Science and Technology, an International Journal*, *19*(2), 737-753, (2016).

[15] Khatibsyarbini, M., Isa, M. A., & Jawawi, D. N. A., "Particle swarm optimization for test case prioritization using string distance." *Advanced Science Letters*, *24*(10), 7221-7226, (2018).

[16] Samad, A., Mahdin, H. B., Kazmi, R., Ibrahim, R., & Baharum, Z., "Multiobjective test case prioritization using test case effectiveness: multicriteria scoring method." *Scientific Programming*, *2021*(1), 9988987, (2021).

[17] Bharathi, M., "Hybrid particle swarm and ranked firefly metaheuristic optimization-based software test case minimization." *International Journal of Applied Metaheuristic Computing (IJAMC)*, *13*(1), 1-20, (2022).

[18] Deneke, A., Assefa, B. G., & Mohapatra, S. K., "Test suite minimization using particle swarm optimization." *Materials Today: Proceedings*, *60*, 229-233, (2022).

[19] Boyar, T., Oz, M., Oncu, E., & Aktas, M. S., "A novel approach to test case prioritization for software regression tests." *Computational Science and Its Applications–ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part VII 21*, 201-216. Springer International Publishing, (2021).

[20] Bajaj, A., & Sangwan, O. P., "Discrete and combinatorial gravitational search algorithms for test case prioritization and minimization." *International Journal of Information Technology*, *13*, 817-823, (2021).

[21] Zhao, W., Wang, L., Zhang, Z., Fan, H., Zhang, J., Mirjalili, S., & Cao, Q., "Electric eel foraging optimization: A new bio-inspired optimizer for engineering applications." *Expert Systems with Applications*, *238*, 122200, (2024).

[22] Bastos, D. A., Zuanon, J., Rapp Py-Daniel, L., & de Santana, C. D., "Social predation in electric eels." *Ecology and evolution*, *11*(3), 1088-1092, (2021). https://doi.org/10.1002/ ece3.7121

[23] Malishevsky, A. G., Ruthruff, J. R., Rothermel, G., & Elbaum, S., "Cost-cognizant test case prioritization." Technical report TR-UNLCSE-2006–0004, University of Nebraska-Lincoln, 97–106, (2006).

[24] Viswanathan, G. M., Afanasyev, V., Buldyrev, S. V., Murphy, E. J., Prince, P. A., & Stanley, H. E., "Lévy flight search patterns of wandering albatrosses." *Nature*, *381*(6581), 413-415, (1996). https://doi.org/10.1038/381413a0

[25] Zhao, S., Zhang, T., Ma, S., & Wang, M., "Sea-horse optimizer: A novel nature-inspired meta-heuristic for global optimization problems." *Applied Intelligence*, *53*(10), 11833-11860. (2023). https://doi.org/10.1007/s10489-022-03994-3

[26] V. Tomar, M. Bansal, and P. Singh, "Application of Gradient-Based Optimizer for Development of Enhanced Fitness Function with Critical Path Weights for Generating Test Data." International Journal of Intelligent Systems and Applications in Engineering, 12(21s), 4403, (2024). –. Retrieved from https://ijisae.org/index.php/IJISAE/article/view/6296

[27] Li, Z., Harman, M., & Hierons, R. M., "Search algorithms for regression test case prioritization." *IEEE Transactions on software engineering*, *33*(4), 225-237, (2007).

[28] Marchetto, A., Islam, M. M., Asghar, W., Susi, A., & Scanniello, G., "A multi-objective technique to prioritize test cases." *IEEE Transactions on Software Engineering*, *42*(10), 918-940, (2015).

[29] Elbaum, S., Malishevsky, A. G., & Rothermel, G., "Test case prioritization: A family of empirical studies." *IEEE transactions on software engineering*, *28*(2), 159-182, (2002).