Journal of Electronic Testing (2023) 39:659–677
https://doi.org/10.1007/s10836-023-06086-3

Efficient Fault Detection by Test Case Prioritization via Test Case Selection

J. Paul Rajasingh[1] · P. Senthil Kumar[2] · S. Srinivasan[3]

Received: 2 August 2023 / Accepted: 29 September 2023 / Published online: 22 November 2023

Abstract
One of the significant features of software quality is software reliability. In the testing phase, faults are identified and corrected by integrating them into software development, thus obtaining better reliability. Here, by utilizing the Elliptical Distributions-centric Emperor Penguins Colony Algorithm (ED-EPCA)-based Test Case Prioritization (TCP), an effectual Fault Detection (FD) technique is proposed using Fishers Yates Shuffled Shepherd Optimization Algorithm (FY-SSOA)-based Test Case Selection (TCS). Initially, for the incoming source code, the Test Case (TC) is created. Then, the significant factors needed for TCS and prioritization are identified. Next, by utilizing the Log Scaling-centered Generalized Discriminant Analysis (LS-GDA) model, the estimated factors are abated further to enhance the TCS along with prioritization for the Fault Detection Process (FDP). Then, using the FY-SSOA, the optimized TCs are selected. Subsequently, with the help of ED-EPCA, the TCs being selected are ranked as well as prioritized. Finally, to validate the proposed system's effectiveness, the model's performance is evaluated in the working platform of Java and analogized with the traditional methodologies. The results indicate that the test case prioritization-based fault detection method is robust with a 99.23% fault detection rate and a small amount of memory usage, which is only 8245475 kb by generating a large number of test cases.

Keywords Software testing · Test case · Prioritization · Selection · Entropy · Fault detection

1 Introduction

In the modern era, being an emerging paradigm, Information Technology (IT) acts as a backbone of the software industry. For software industrial growth, developing good quality software and maintaining its prominence is highly essential [9]. Software testing is a crucial but effort-intensive activity. In the software development life cycle, testing is a vital part. Moreover, mainly to enhance the quality, performance, reliability, and efficacy of the software, the testing process is performed [18]. Furthermore, in addition to maintaining product quality, testing certifies the proper functioning of the software's extended version. In the entire testing process, test cases are deliberated as the basic mechanisms. It is also prioritized to uncover a number of faults simultaneously or to detect severe faults at an early stage for achieving an objective function.

Permitting software testers to detect more crucial TCs is one of the effective strategies for handling risky components. Thus, certain faults related to those components are revealed. Faults pertinent to those hazardous components are also identified faster after detecting the TCs [20]. To gauge the program execution of a test on the program under test, the code coverage is utilized. Usually, with the aid of computer programming codes, which are already written to perform the required task, software applications are developed [22]. Generally, certain faulty instances might exist in these predefined codes. Thus, owing to software defects, it resulted in buggy software development. Next, several factors related to the software Fault Correction Process (FCP) are identified [10]. For developing the

Responsible Editor: B. Arasteh

✉ J. Paul Rajasingh
paulrajasingh2912@gmail.com

P. Senthil Kumar
drsenthilkumar2010@gmail.com

S. Srinivasan
ssn.cse@rmd.ac.in

[1] Information and Communication Engineering, Anna University, Chennai, India

[2] Information Technology, P. B. College of Engineering, Irungattukottai, Chennai 602117, India

[3] CSE, RMD Engineering College, Kavaraipettai, Tiruvallur District 601206, India

Software Reliability Growth Model (SRGM), the FDP is included in the FCP [27]. For an effectual TCP in minimum time, optimization methodologies are needed to be attained whilst retaining the software quality.

Conventionally, all activities of software testing are conducted by human testers (test engineers) manually. For every System Under Test (SUT), a set of test specifications is generated in the manual testing process. To prioritize the test cases, numerous nature-inspired algorithms, such as genetic algorithm, particle swarm optimization, ant colony optimization, et cetera are espoused [25]. For evaluating the fitness of varied models, various methodologies subsuming Factor analysis (FA) and regression are also employed [12]. To aid different classification tasks, Neural networks (NN) are also implemented in numerous software systems [28]. The former TCP was performed utilizing codes. On the contrary, for TCP, model-based mutation testing was employed in which valid information in terms of the objective function along with an automatic fault seeding system is desired [30].

In spite of all the efforts of researchers to enhance software testing quality, software testing's efficiency and effectiveness are still an open challenge and remain below the marginal line of customer's expectations. Testing is still largely ad hoc, expensive, and unpredictably effective even after being a widespread and mainstream validation technique in the software industry. In addition, the test case prioritization execution and analysis possess numerous challenges and problems. It is not just a simple reordering of the test cases; it needs deep insights and analysis procedures for getting the fruits of prioritization approaches.

The test suites were ordered on their fault detection ability with the costs of test suites by the prior works. However, they ignored the code coverage and other features of cost, such as test suite size and code size of the application under testing. A few methods incorporated the code-coverage TCP techniques, which may treat all faults equally. However, it is believed that one of the challenges faced by these techniques is the ability to monitor needs covered in the system test. In other words, prioritization must be cost-effective for testing. In most of the methods, problems like time complexity and low memory usage that might be utilized for tackling issues, such as selection-prioritization and selection across collections of test cases are not concentrated. It is possible that executing test cases that have more potential to cover the maximum possible paths of SUT first can alleviate the time complexity and low memory usage issues in test case selection and prioritization. Under this motivation, an effectual FD system by utilizing the ED-EPCA-based TCP has been proposed here utilizing FY-SSOA-based TCS to avoid the aforementioned issues.

1.1 Problem Statement

Even though there are various benefits in the prevailing methodologies, there exist certain disadvantages, which are listed below:

- TCP is a process, where to maximize certain conditions like the test cases' FD Rate, the TCs are ordered within the test suite of a SUT.
- The generated test cases were mitigated by the conventional TCP algorithm. Thus, a time complexity problem is caused whilst running the TCs being engendered.
- Poor computation efficacy was possessed by the risk-centric TCP utilizing a fuzzy expert system strategy.
- Handling larger input datasets was a complicated task for the input-based adaptive randomized TCP. In addition, it could not achieve cost-effective outcomes.
- Only a fixed strength was considered by the traditional TCP methodologies, which did not consider multiple strengths whilst selecting every single test case.

Thus, via an effectual FD system, the proposed model deterred the aforementioned drawbacks with the following contributions as,

- To augment the fault detection rate, the more significant test cases have been performed first via the ordering of test cases for prioritization.
- To minimize the time complexity of TCP, the prioritization is carried out only for the selected test cases using the FY-SSOA algorithm rather than running all the test cases generated.
- To improve the computation efficiency, the maximization criteria-based prioritization using the ED-EPCA method is employed.
- To accomplish more cost-effective results and to deal with large input data, factor identification and factor reduction using LS-GDA are utilized.
- Unlike conventional TCP methodologies, the test case selection under multiple strengths is carried out in the proposed method.

The paper's remaining parts are arranged as: some of the primitive works regarding the TCS and prioritization are reviewed in Section 2; the proposed technique is explicated in Fig. 3; the proposed model's superiority measure is demonstrated in Section 4; finally, the paper is winded up with the future work in Section 5.

2 Literature Survey

Gao [11] developed an SRGM framework via heterogeneous debuggers consideration. Here, to optimize the testing process, a cost analysis technique was employed. The FD

and fault correction were deemed to be a departure as well as an arrival process. Moreover, via a queuing system, these processes were illustrated. The simulation outcomes proved the presented model's effectiveness regarding performance metrics. However, the model was ineffective whilst working in a complex environment.

Xiao et al. [29] elaborated on a stepwise prediction system for the FDP and the FCP regarding the Artificial Neural Network (ANN). In the presented model, the greatest impact was possessed by the testing effort on FDP as well as the FCP. Thus, regarding software reliability along with total cost, the optimal release technique was attained. Conversely, the model's superiority measure was affected by a higher Mean Square Error (MSE).

Gokilavani and Bharathi [13] recommended a Principle Component Analysis (PCA) extraction along with a K-Means clustering-centric ranking model to investigate faults in the software. Here, for an effectual FDP, the model underwent (i) pre-processing, (ii) feature selection, (iii) clustering, and (iv) ranking process. Therefore, using the adjacency matrix betwixt the TCs and FDR, the sum of the faults was detected superiorly. However, information about the basic needs along with risks related to bugs was not provided by this model.

Lin et al. [19] presented a systematic mechanism to determine the FD and Diagnosis (FDD) model's supremacy. Here, from the dataset wielded, input scenarios as well as the input data samples were extricated. Additionally, by utilizing a condition-centric conventional system, the ground truth fault operational stage was proffered. Consequently, the model deterred unintentional lighting runtime complications. However, owing to fault intensities together with seasonal diversities, the dataset utilized was limited.

Cui et al. [8] modeled a fault localization methodology by the amalgamated usage of the spectrum as well as mutation. Regarding the program spectrum, the faults were localized by the Spectrum-Based Fault Localization (SBFL). Alternatively, concerning the mutant, faults were localized by the Mutation-Based Fault Localization (MBFL). At last, the ranking was performed. Hence, the utilization of the adopted ranking methodology enhanced the fault localization accuracy. However, the programs with multiple faults were not considered by this model.

Jahan et al. [15] proffered a semi-automatic risk-centric TCP model regarding software modification information together with the invocation relationship. Here, for risk analysis, the factors considered were complexity, the desired modification information, along with the model's size. The experiential outcomes demonstrated that when compared with the prevailing methodologies, the presented one spotted defects earlier with higher efficacy. Nevertheless, the potential errors were increased owing to the number of modified requirements, complexity, along with size.

Nagaraju et al. [23] developed a covariate SRGM along with formulating the optimal test activity allocation aimed at the maximization of FD. Here, regarding the discrete cox hazards methodology, a Poisson process was employed for SRGM. Therefore, by utilizing limited testing resources, the optimal test activity allocation problem was addressed. However, the presented covariate model's practical usage was not successful.

Nithya et al. [24] presented a gradient-centric methodology for the semi-automated selection of the software TC. For the TCS, a simulated annealing model was utilized. Similarly, for an effectual testing process, optimization methodologies were employed. Thus, the unwanted and redundant data were removed and more faults were detected. However, the convergence rate was affected by the usage of random permutations.

Choudhary et al. [7] introduced an effort-centered SRGM model for detecting along with correcting faults in software test cases by employing Multi-Attribute Utility Theory (MAUT). Furthermore, to estimate software failures, a Non-Homogenous Poisson Process (NHPP) was employed. Therefore, the total incurred cost was mitigated by the reduction in the launch time. However, the fault correction complexity was increased owing to the addition of faults in the fault removal process.

Mahdieh et al. [21] presented the coverage-centric TCP model aimed at fault proneness estimations. For defect prediction, the software's bug history was utilized. After that, fault-proneness regions of every single source code were analyzed as well as induced in the coverage-based TCP techniques. The experiential outcomes confirmed the presented model's effectiveness over the traditional methodologies. Nevertheless, the model provided an inaccurate prediction of faults due to the utilization of the lower number of bugs.

Bagherzadeh et al. [5] proffered a Reinforcement Learning (RL)-based model for TCP. Here, 3 alternative ranking methodologies were utilized since the TCP was regarded as the ranking problem. Therefore, higher ranking accuracy of the testing process was obtained from the analysis and the extraction of TC data as well as source code history. However, the model was time-consuming. Moreover, it provided potentially out-of-date models.

Chi et al. [6] defined an Additional Greedy method-centric Call sequence (AGC) framework for the regression testing process. Here, the first priority was given to the TCs with more call sequences. Then, it was deemed as a calling network regarding the program behaviors. Using the removal of redundant test cases, the test suite size was reduced by the TC minimization model. However, here, the testing process ceased prematurely at a certain arbitrary point.

Ali et al. [1, 2] developed Change Test cases and Failed Frequency (CTFF) based prioritization methodologies. Regarding the highest failure frequency, this model included 2 phases, comprising clustering along with the prioritization of parameters. Initially, using clustering, the TCs were prioritized. By employing the coverage condition, the test cases were prioritized if the test cases interfered with each other. Therefore, the model effectively detected more faults. However, the outcomes were not validated statistically.

Ali et al. [1, 2] suggested a pattern-centered verification model aimed at TCP. Here, by utilizing the observer patterns, the test cases were accessed often. Moreover, sorting, execution rate, and fault history were certain strategies via which the test cases were prioritized. Therefore, the presented model augmented the FDR together with random priority. However, reliability along with ambiguity issues occurred owing to multiple test cases with the same frequency.

Huang et al. [14] engendered Regression TCP (RTCP) model to test the developed software. The model was highly grounded on the combination coverage along with the code coverage concept. Furthermore, adaptive random, additional, total, and search-centric test prioritization RTCP sorts were also utilized. Consequently, when analogized with the traditional models, the presented model obtained comparable testing efficacy. However, the model was affected by the time overhead problem.

Arasteh et al. [3] intended to reduce the mutation test process's time and cost. The model identified the most fault-prone paths of a program using the Artificial Bee Colony algorithm and injected the mutation operators on the recognized fault-prone instructions and data. The experimental outcomes indicated that the method reduced generated mutants and the cost of mutation testing. Due to the improper exploitation ability of the algorithm, there occurred inaccurate identification of fault-prone paths.

Arasteh et al. [4] automated test data generation by developing a method Tracxtor using Imperialist Competitive Algorithms (ICA). The test data were generated under maximum branch coverage in a limited amount of time. As per the outcomes acquired from the experiments, the algorithm outperformed the other algorithms. However, the memory usage during test case generation was not considered.

Khari et al. [16] established an automated testing tool using the components of software testing. The test suit was generated utilizing the test suite generation approaches for the flow graph of the software under test. Further, this generated test suite was optimized utilizing the cuckoo search algorithm or artificial bee colony algorithm. When analogized to other algorithms, this technique can render a set of minimal test cases with maximum path coverage. But, due to the generation of test cases regardless of path, the path coverage achieved by the model was less.

Khari et al. [17] concentrated on heuristic algorithms' performance assessment to select the best-suited algorithm for path coverage-centric optimization. The algorithms were deployed to create test suites for the program under test and optimize them under path coverage and branch coverage generated by the test data. Results showed that the algorithms were well suited for path coverage-centric optimization techniques. As the heuristics were developed for the same problems, generating comprehensive test suites was difficult.

3 Proposed Efficient Fault Detection Methodology

In recent days, software has become highly significant in the daily routine of human life. Owing to complexity along with longevity, modern software systems encompass a larger set of requirements. Therefore, in a software engineering project, by utilizing the proposed FD model, the program's behavior is analyzed in the crucial stage of software testing. Figure 1 exhibits the proposed model's framework.

3.1 Test Case Generation

The process of generating test cases for the corresponding system from the model, which describes the system, is called test case generation. The generation of these test cases can't be executed manually, which increases the need for automating test case generation. Initially, for the selected source codes, the TC is created in the FDP. In the proposed model, the source code and test cases are collected from publicly available open source. When the test case is not available for the corresponding source code, the test cases are derived from the development of source code and requirements specifications manually.

Initially, for the selected source codes, the TC is created in the FDP. Information like the name of the individual (n_i) who generated the TC along with conducted the testing

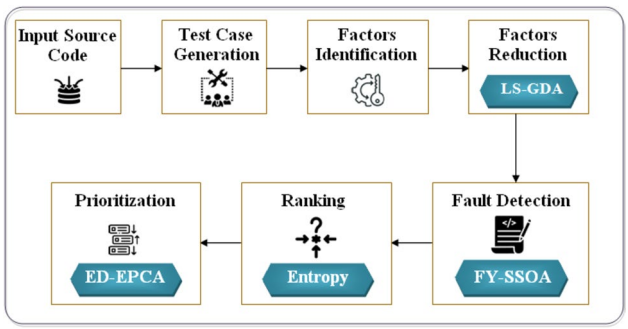

Fig. 1 Baseline Structure of proposed work

process, tested time (t_t), test case id (t_{id}), tested software version (*version*), expected result (E_r), and actual result (A_r) are encompassed in the test cases that are generated. Here, the input source code is specified as I_{sc}, and the TCs created are expressed as T_{case}. Since the generated test cases may or may not have a 100% fault detection rate, the test case selection has been done by factors identification and reduction.

3.2 Factors Identification

To maintain the software quality, certain significant factors [26] are detected or extracted for prioritizing the test cases after generating the test cases. The factors being identified are explicated below:

- Code coverage (C_c)
  The ratio of the program element by tests, which are engendered manually or automatically in source code, is proffered as code coverage. Line coverage, branch coverage, and method coverage are encompassed in program elements. It is expressed as,

$$C_c = \frac{lines\ of\ codes\ \text{covered by tests}}{total\ number\ of\ code} * 10 \tag{1}$$

- Data flow (D_f)
  The test case coverage of interactions betwixt the data items employed for software development purposes is defined as data flow. It is formulated as,

$$D_f = \frac{test\ data\ utilized}{Total\ number\ of\ test\ cases} \tag{2}$$

- Fault Proneness (F_P)
  It is utilized to evaluate the error-prone requirement. Regarding the number of field failures and in-house system test failures, which are already detected in the code, the F_p is measured as,

$$F_p = \frac{n_c}{n_p} \times 10 \tag{3}$$

  where, the number of test cases that are correctly predicted as faulty modules is specified as n_c, and the total number of predicted faulty modules is signified as n_p.
- Customer Assigned Priority (C_{prior})
  The priority assigned by the customer to the test case regarding the customer's need or necessity of the software being developed is termed customer-assigned priority. For every single requirement, the customer assigns a value betwixt 1 to 10. The higher customer priority is specified by 10.
- Implementation Complexity (I_c)
  The model's implementation complexity is measured by the total number of requirements. The methodology becomes more complicated with a larger number of requirements, thus leading to the production of the wrong product. It is expressed as,

$$I_c = \frac{N^m}{\max_{\forall m}\langle N^m \rangle} \tag{4}$$

  Here, the number of requirements or conditions in the proposed methodology is notated as N^m.
- Changes in requirement (R_c)
  The frequency of modification required in the development of the software project from the date of onset of the project is evaluated by R_c, which is formulated as,

$$R_c = \left(\frac{u}{v}\right) \times 10 \tag{5}$$

  where, the number of requirements is specified as v, and the requirement changes are signified as u.
- Fault Impact of requirements (I_{fault})
  The process of discovering the number of failed requirements in the development of the project as well as the number of in-house failures is defined as the fault impact of requirements. It is expressed as,

$$I_{fault} = \left(\frac{a}{b}\right) \times 10 \tag{6}$$

  where, the total number of requirements is notified as b and the number of in-house failures is signified as a.
- Completeness (ς)
  The condition for execution (g), the degree of success $(\aleph)$, and the limitations (h) in attaining the probable solution are explicated here. It is formulated as,

$$\varsigma = \left\{\frac{1}{\aleph}, \frac{g}{h}\right\} \tag{7}$$

- Traceability (T_r)
  The measure of the graphical representation (mapping) of the requirements (R) and test (T) is termed traceability. In this, it is verified whether the requirements are completed and tested or not. It is modeled as,

$$T_r \leftrightarrow \{R, T\} \tag{8}$$

- Execution time (T_e)
  The execution time is proffered as the time needed to accomplish the particular code (task). Therefore, the factors identified (F_I) are expressed as,

$$F_I = \left\{T_{case}(l), C_c, D_f, F_p, C_{prior}, I_c, R_c, I_{fault}, \varsigma, T_r, T_e\right\} \tag{9}$$

  where, the number of factors identified is denoted as $I = 1, 2, 3, ..., f$.

3.3 Factor Reduction Using LS-GDA

Here, without the loss of vital information, the identified factors are mitigated further to augment the accuracy of ranking test cases subsumed in the FDP. Generalized Discriminant Analysis (GDA), the most frequently utilized model, is utilized effectively for feature reduction. Here, via the estimation of eigenvalues by Eigen Value Decomposition (EVD) methodology, higher dimensional features are mitigated into lower dimensional ones. Nevertheless, the eigenvalue computation is quite complicated. Moreover, it is impossible to handle a larger number of datasets. The Log Scaling (LS) normalization model, which is termed LS-GDA, is utilized to abate the error rate. The steps included in the LS-GDA are given below:

Initially, the factors identified $\left(F_I\right)$ are inputted into the LS-GDA algorithm in m-dimensional search space. The LS-GDA maps the input vector $\left(F_I\right)$ in space (m) into the input vector $\phi(L^s)$ in space f. In the feature transformation process, the LS-GDA aims to minimize the between-class scatter due to the non-linearity of data. The wither class matrix $(\mathfrak{F}^w)$ and between-class scatter matrix $\left(\mathfrak{F}^b\right)$ for the mapped factors are formulated as,

$$\begin{cases} \mathfrak{F}^b = \sum_{I=1}^{D} F_I\left(\eta^I - \eta\right)\left(\eta^I - \eta\right)^T \\ \mathfrak{F}^w = \sum_{I=1}^{F_I} \left(\phi(L^s) - \eta^I\right)\left(\phi(L^s) - \eta^I\right)^T \end{cases} \tag{10}$$

Here, the mean value of the $i-th$ factor identified is specified as η_I, and the mean of all factors identified is symbolized as η. The LS-GDA finds the projection matrix $\left(R_f\right)$ that maximizes the optimization criteria defined as,

$$R_f = \arg\max\left(\frac{R^T\ \mathfrak{F}^b\ R}{R^T\ \mathfrak{F}^o\ R}\right) \tag{11}$$

The vector (ψ) for $\left(R_f\right)$ is identified by solving the eigenvalue problem. LS-GDA finds the eigen values (δ) and eigen vectors (ψ) to maximize the optimized solution attained as,

$$\delta = \frac{\psi^T \mathfrak{F}^b \psi}{\psi^T \mathfrak{F}^o \psi} \tag{12}$$

As eigen vectors lie in the span of $\varphi(L^s)$, the eigen values are calculated as,

$$\psi = \sum_{I=1}^{D}\sum_{l=1}^{F_I} \beta_{Il} \log(L^s) \tag{13}$$

Here, the span co-efficient is denoted as β_{Il}, and the LS criterion is indicated as $\log(L^s)$. The solution to eigen value problem is obtained by solving,

$$\delta(\beta) = \frac{\beta^T\ \xi Q\ \xi\beta^T}{\beta^T\ \xi\xi\beta^T} \tag{14}$$

where, β is the vector of weights, Q is the symmetrical matrix, and ξ is the gram matrix. To compute the gram matrix, the dot product betwixt the mapped factors by utilizing the positive definite kernel function is estimated. The gram matrix $(\xi(\varphi(\cdot)))$ composed of dot products is described as,

$$\xi(\varphi(\cdot)) = \xi\left(\varphi(L^s)^T \varphi(L^s)\right) \tag{15}$$

Solving the eigenvalue problem yields (β) that define the projection vectors $(\psi) \in R_f$. For this, the matrix ξ undergoes decomposition, which is then replaced in the Eq. (14) to compute eigenvectors as,

$$\delta = \frac{\chi^T\left(I^T\ QI\right)\chi^T}{\chi^T\left(I^T\ I\right)\chi^T} \tag{16}$$

Here, the normalized eigenvectors' identity matrix is specified as I, and χ is the normalized eigen vectors. The flowchart of the LSGDA is shown in Fig. 2.

Concurrently, the matrix $I^T QI$ endures Eigen decomposition; thus, the reduced non-linear discriminant factors are formulated as,

$$R_f = \left\{R_1, R_2, \cdots\cdots, R_N\right\} \tag{17}$$

Here, the N number of reduced factors is represented as $f = 1, 2, 3, ..., N$. For better optimal TCS, the factors being reduced are fed into the FY-SSOA system.

3.4 Test Case Selection by FY-SSOA

Here, by employing the FY-SSOA, the optimal TCs are selected. The shuffled Shepherd Optimization Algorithm (SSOA), developed to solve optimization problems, is a metaheuristic population-based algorithm. The shepherd's behavior is the concept behind this algorithm. In this, sheep are generated randomly. Then, regarding the previously created herd, they are put into a herd. Till forming all herds via the shuffling process, which ameliorates the herds' survivability, the process is continued. Nevertheless, for guiding the sheep toward the horse, more waiting time is required to compute step size regarding the shepherd's movement. Thus, for shuffling, the Fishers Yates (FY) technique is utilized, which mitigates the waiting and enhances the shuffling process. This FY-centered shuffling of SSOA is termed FY-SSOA. The following are the steps involved in FY-SSOA:

- Initially, the shepherds' initial population also known as the community (here, the population indicates the num-

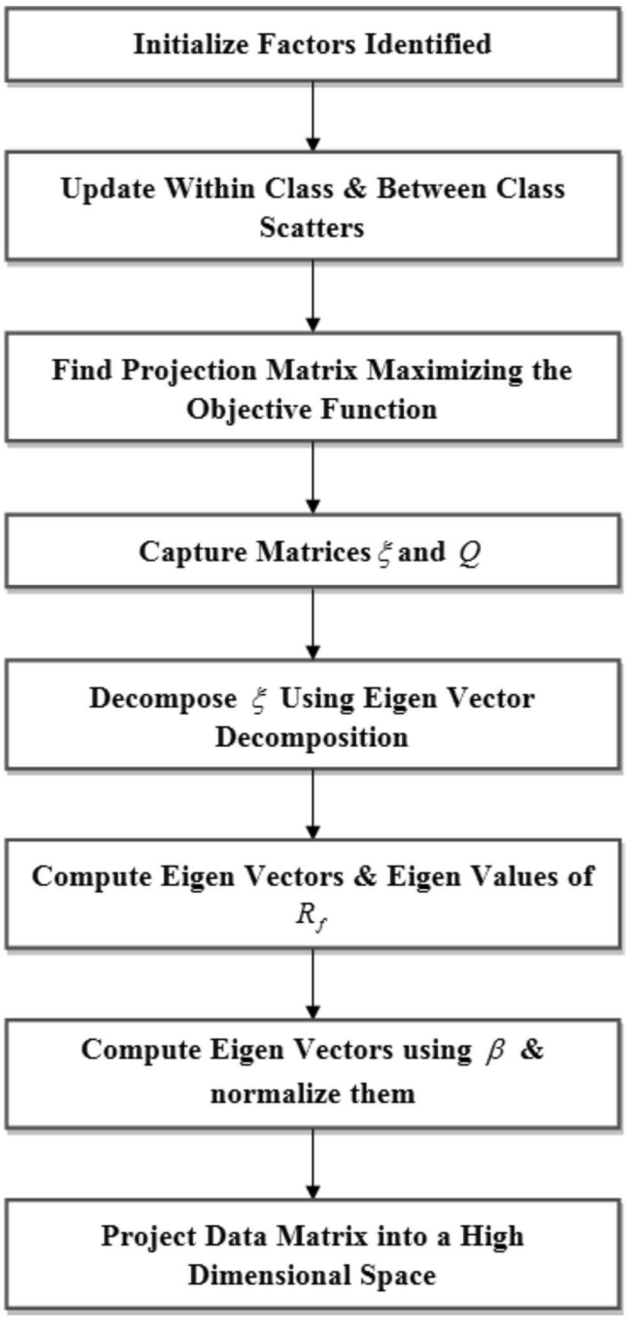

Fig. 2 Flowchart of the LSGDA

ber of test cases generated) is generated randomly. It is expressed as,

$$R_{f,g} = R_{f,g}^{\min} + rand \times \left(R^{\max} - R^{\min}\right) \tag{18}$$

where, the variable's lower bound is specified as $R_f^{\min}$, the variable's upper bound is signified as $R_f^{\max}$, and a random variable betwixt 0 and 1 is notified as *rand*. Also, the $N-$ number of communities is notated as $f = 1, 2, ..., N$, and the $M-$ number of members belonging to every single community is represented as $g = 1, 2, ..., M$.

- The fitness value (ρ) of every single member in the community is computed after generating the community members (sheep and horse), which is measured as,

$$\rho = N * M \tag{19}$$

The fitness criterion is to select test cases that can cover uncovered paths via assessing the fault detection ability based on the reduced factors from Section 3.3. Hence, the fitness function considering the factors selected is employed in this paper. Considering these factors would enable more effective identification of faults earlier in the software process. According to the fitness function, the test cases that contain the required selected factors are selected for fault detection. It can be expressed as,

$$\rho_{N\times M} = \arg\min_{(N,M)} \max\left(R_f\right) \tag{20}$$

- After calculating the fitness value, in accordance with the fitness value, all the members are shuffled along with sorted in descending order by utilizing the FY algorithm.
- The FY algorithm randomizes the community as well as reordered them in decreasing order.
  - Write down every single member of the community population as of 1 *to* ρ.
  - Select a random member *s* betwixt one and the number of unstruck members (ρ).
  - In the community that has not been crossed out, the s^{th} member is crossed out by counting from the lower end. Subsequently, write it at the end of the separate list. Here, the shuffled list of community members is represented.
  - These steps are repeated until all the community members have been struck out.
- After that, by performing the step size estimation process $\left(S_{f,g}\right)$, the movement of every single member of the community is estimated. Under the diversification strategy $\left(S_{f,g}^{diverse}\right)$, the members having the ability to visit new regions in space are enclosed. But, under the intensification strategy $\left(S_{f,g}^{\text{int}ensify}\right)$, the members having the capacity to visit the regions that have already been visited are encompassed. The step size estimation process is expressed as,

$$S_{f,g} = S_{f,g}^{diverse} + S_{f,g}^{\text{int } ensify} \tag{21}$$

Here, $S_{f,g}^{diverse}$ and $S_{f,g}^{\text{int } ensify}$ are formulated as,

$$S_{f,g}^{diverse} = \lambda \times c \times \left(R_{f,d} - R_{f,g}\right) \tag{22}$$

$$S_{f,g}^{\text{int }ensify} = \lambda^* \times c_1 \times \left(R_{f,in} - R_{f,g}\right) \tag{23}$$

where, the random integers are symbolized as c and c_1, the best member (horse) is indicated as $R_{f,d}$, the worst members (sheep) of the community are denoted as $R_{f,in}$, and the control parameters for exploration and exploitation are given as λ and λ^*. The control parameters are represented as,

$$\lambda = \lambda_0 - \lambda_0 \times T \tag{24}$$

$$\lambda^* = \lambda_0^* + \left(\lambda_{MAX}^* - \lambda_0^*\right) \times T \tag{25}$$

Thus, it is evident that as iteration (T) increases, λ^* increases with mitigation in λ value. In this, λ_0^* specifies the reduction in λ value and λ_{MAX}^* symbolizes the maximum value.

- The new position of every single member in the community $\left(R_{f,g}^{new}\right)$ is attained after estimating the movement of every single member in the community. It is computed as,

$$R_{f,g}^{new} = R_{f,g} + S_{f,g} \tag{26}$$

Thus, till gratifying the stop condition, the position updation process is repeated. Or else, return to step 3 and continue the phases. Hence, the test cases selected $\left(S_{tcs}\right)$ via FY-SSOA are expressed as,

$$S_{tcs} = S_1, S_2, ..., S_p \tag{27}$$

where, the number of test cases selected is notified as $tcs = 1, 2, 3, ..., p$. The FY-SSOA's pseudocode is cited below,

Pseudocode for proposed FY-SSOA

Input: Reduced factor $\left(R_f\right)$

Output: Selected test cases $\left(S_{tcs}\right)$

Begin

Initialize Shepherd population, iteration I_t and maximum Iteration M_t

Initialize position $(R_{f,g})$

Determine fitness value $\rho = N * M$

While $I_t \le M_t$

Perform Fisher-Yates Shuffling

Estimate Step size $S_{f,g} = S_{f,g}^{diverse} + S_{f,g}^{\text{int}ensify}$

Determine updated position $R_{f,g}^{new} = R_{f,g} + S_{f,g}$

End while

Obtain $\left(S_{tcs}\right)$

End

The pseudo-code of the FY-SSOA algorithm includes the fundamental steps used for the optimization process. Firstly, the algorithm divides the population into a number of communities from which the best members are selected for the herding behavior. The selection process is done by evaluating the fitness of each sheep and sorting the population using Fisher-Yates Shuffling. Finally, the sub-population generated by recording the best solutions gives the optimal test cases selected for the testing process.

3.5 Ranking with Entropy

The ranking methodology is implemented after completing the TCS process. Generally, to prioritize the test cases being selected, the ranking is employed. Here, by adopting the entropy-centered ranking (ε^p) model, the test cases are prioritized. It is formulated as,

$$\varepsilon^p = -\sum S_{tcs} \cdot \log\left(\frac{1}{S_{tcs}}\right) \tag{28}$$

3.6 Test Case Prioritization via ED-EPCA

Here, to achieve the test cases' complete reordering, the more crucial TCs are selected regarding their priority information. After choosing appropriate test cases, an algorithm should be determined for ordering test cases such that the fault detection rate by executing the test cases in the prioritized sequence is maximized. The problem of prioritizing test cases to attain such a maximum fault detection rate and enabling earlier or less costly detection of errors is an NP-hard problem. The migration behavior of emperor penguins from cold to warmer domains in the colony is the basis for the development of the Emperor Penguin Colony (EPC) optimization algorithm. EP is uniformly distributed as well as experiences spiral-like movement in the colony. Moreover, temperature and distance are the factors on which the huddling behavior of the emperor penguins solely relies. Nevertheless, the convergence speed is affected by the randomness vector enclosed in the movement of the uniformly distributed penguins, thus resulting in a local optimum problem. The EP is Elliptical Distributed (ED) in the dimensional search space to deter the aforementioned issues. The induction of ED in the traditional EPCA is termed ED-EPCA. The steps included in the ED-EPCA algorithm are explicated below:

Stage 1: Primarily, the population of the penguins in the colony (ε^p) is initialized as,

$$\varepsilon^p = \varepsilon^1, \varepsilon^2, \varepsilon^3, ..., \varepsilon^C \tag{29}$$

where, the $C-$ number of penguins in the colony is indicated as $p = 1, 2, 3, ..., C$. Here, the population of the penguins in the colony is the selected test cases in software testing.

Stage 2: After initializing the population, the fitness value of every single penguin $\left(\overline{f(\varepsilon^p)}\right)$ in the population is computed as,

$$\overline{f(\varepsilon^p)} = \overline{f\left(\varepsilon^1, \varepsilon^2, ..., \varepsilon^C\right)} \tag{30}$$

Stage 3: Here, the penguins' movement toward the objective function (warmer region) is determined. In this estimation, the heat radiation transfer ($\hbar^{pen}$) is evaluated as,

$$\hbar^{pen} = \hbar_{tot} O \gamma t^4 \tag{31}$$

Here, the penguin's total surface area is specified as $\hbar_{tot}$, the bird's plumage emissivity is signified as O, the Stefan–Boltzman constant is notified as γ, and the absolute temperature is symbolized as t. Furthermore, the penguin's total surface area $\hbar_{tot}$ is formulated as,

$$\hbar_{tot} = \hbar^u + \hbar^v + \hbar^w + \hbar^x \tag{32}$$

where, the surface area of the trunk is specified as $\hbar^u$, the beak area is signified as $\hbar^v$, the area of the penguin's head is notated as $\hbar^w$, and the flipper area is symbolized as $\hbar^x$.

$$\hbar^u = 2\pi \frac{cd}{b} \sin^{-1} b + 2\pi d^2 \tag{33}$$

$$\hbar^v = \pi RS \tag{34}$$

$$\hbar^w = \pi (D - R)^2 \tag{35}$$

$$\hbar^x = L \times M \tag{36}$$

Here, the semi-length of the body is notified as c, minor axes length is indicated as d, the body length is denoted as b, the cross-sectional area and hypotenuse are represented as R and S, the height of the penguin's head is exhibited as D, and the length and width of the flippers are illustrated as L and M.

Stage 4: After evaluating the heat radiation transfer, the attractiveness of two penguins ($\hbar$) in the colony is measured as,

$$\hbar = \hbar_{tot} O \gamma t^4 e^{-\vartheta \lambda} \tag{37}$$

where, the coefficient of attenuation is proffered as ϑ and the linear distance betwixt the penguins is depicted as λ.

Stage 5: A spiral path was formed by the movement of the penguin toward the warmer one. Thus, the logarithmic spiral formula is expressed as,

$$\mathfrak{R} = \tau e^{\tau'' \phi} \tag{38}$$

Here, the distance from the origin is signified as $\mathfrak{R}$, the arbitrary angle is modeled as ϕ, and τ and τ'' are the constants.

Stage 6: Next, owing to the penguins' attractive nature, they get distracted during their movement. Thus, the destination is not attained; it also stops after a long distance. Consequently, by estimating the distance between the two penguins, the updated position of the penguin is estimated. It is formulated as,

$$\overline{\overline{D_{mn}}} = \frac{e}{f}\sqrt{f^2+1}\left(e^{f\phi_n} - e^{\phi_m}\right) \tag{39}$$

where, the distance betwixt penguins m and n is symbolized as $\overline{\overline{D_{mn}}}$, the constants are denoted as e and f, and the spiral angle of the penguins m and n are notified as ϕ_m and ϕ_n. To attain the distance from m and n', the obtained outcomes are multiplied by the attractiveness value. It is computed as,

$$\overline{\overline{D_{mn'}}} = \hbar\frac{e}{f}\sqrt{f^2+1}\left(e^{f\varphi_{n'}} - e^{\varphi_m}\right) \tag{40}$$

Cartesian and polar co-ordinate relation is utilized for angle ϕ, which is expressed as,

$$\phi = \tan^{-1}\frac{x'''}{y'''} \tag{41}$$

At the position n', the logarithmic spiral's parameters x''' and y''' are expressed as,

$$x''' = ee^{f\frac{1}{f}\ln\left\{(1-\hbar)e^{f\tan^{-1}\frac{y'''_n}{x'''_n}} + \hbar e^{f\tan^{-1}\frac{y'''_n}{x'''_n}}\right\}} \cos\left\{\frac{1}{f}\ln\left\{(1-\hbar)e^{f\tan^{-1}\frac{y'''_n}{x'''_n}} + \hbar e^{f\tan^{-1}\frac{y'''_n}{x'''_n}}\right\}\right\} \tag{42}$$

$$y''' = ee^{f\frac{1}{f}\ln\left\{(1-\hbar)e^{f\tan^{-1}\frac{y'''_n}{x'''_n}} + \hbar e^{f\tan^{-1}\frac{y'''_n}{x'''_n}}\right\}} \sin\left\{\frac{1}{f}\ln\left\{(1-\hbar)e^{f\tan^{-1}\frac{y'''_n}{x'''_n}} + \hbar e^{f\tan^{-1}\frac{y'''_n}{x'''_n}}\right\}\right\} \tag{43}$$

The logarithmic spiral movement turns monotonous owing to the earlier estimation of information about the angle. Therefore, to increase population diversity, the elliptical distribution is adopted. It is expressed as,

$$y''' + \delta\omega \tag{44}$$

where, the mutation factor in path change is represented as δ and the elliptical distribution parameter is specified as ω, which is illustrated as,

$$\omega = \aleph\sqrt{1 - \left(\frac{2e}{f}\right)^2} \tag{45}$$

where, $\aleph$ denotes the elliptic distribution.

Step 7: After achieving either of the required optimizations, the algorithm will be ceased. With varied initial positions, the steps have to run numerous times to get better optimization. Thus, the prioritized TCs, which are notated as TCS_{priori}, are the outcomes of this algorithm. Moreover, they are then wielded for FD purposes.

3.6.1 Fitness Function

In order to adapt ED-EPCA to test case selection, it should be converted to the optimization problem, where the optimized solutions can be obtained by a reasonable fitness function. According to ED-EPCA, candidate solutions in the search space indicate the test cases selected, and the ordered solutions in the search process are obtained with the fitness function.

Hence, for the efficiency and success of optimization, a better fitness function is a significant factor. For test data arrangement, the good fitness function value should be returned for those test data, which nearly meet the fault detection criteria. Fault detection criteria are crucial in software testing to cover 100% of the source code by running the selected test cases in the ordered manner. With the assistance of the fitness function, the ED-EPCA algorithm should arrange the test cases, which have the ability to meet the target testing coverage through a maximum fault detection rate. It can be expressed as,

$$f(\varepsilon^p) = \max\left(\sum_{N,M}\left(dC_c - C_c\right)\right) \tag{46}$$

Here, the test coverage's derivative is signified as dC_c. The pseudo-code of the ED-EPCA is elucidated below:

Pseudocode for proposed ED-EPCA

Input: Ranked test cases $\left(\varepsilon^{p}\right)$

Output: Prioritized test cases TCS_{priori}

Begin

Initialize penguin population $\left(\varepsilon^{p}\right)$

Estimate $\overline{f\left(\varepsilon^{p}\right)}=\overline{f\left(\varepsilon^{1},\varepsilon^{2},...,\varepsilon^{C}\right)}$

For $1\le p\le C$

Evaluate heat radiation transfer $\left(\hbar^{pen}\right)$

Determine $\hbar_{tot}=\hbar^{u}+\hbar^{v}+\hbar^{w}+\hbar^{x}$

Compute attractiveness $\left(\hbar\right)$

Update $\overline{\overline{D_{mn}}}=\frac{e}{f}\sqrt{f^{2}+1}\left(e^{f\phi_{n}}-e^{\phi_{m}}\right)$

Obtain TCS_{priori}

End for

End

The steps used for prioritizing the test cases using LD-EPCA are shown in the above pseudocode. The LD-EPCA algorithm represents the huddling behavior of penguins, where body temperature and heat radiation are the crucial factors. By defining these two factors in the LD-EPCA, the distance between each search agent is updated. Then, the prioritized test cases are obtained by considering the fitness value nearer to fault detection. The flowchart of the LD-EPCA algorithm is shown in Fig. 3,

4 Results and Discussion

Here, to validate the proposed FD model's efficacy, the outcomes attained by the proposed system are contrasted with the baseline techniques. The proposed methodology is executed utilizing JAVA programming.

4.1 Case Study

Can the factors related to the software Fault-based prioritization improve the fault detection rate of test cases is the basic problem used to demonstrate the implementation of proposed FY-SSOA and ED-EPCA methods. The user requirements of the SUT have been taken as the input from which the test cases are generated manually. From the generated test cases, the factors related to software faults are identified and used for test case optimization using the FY-SSOA algorithm. The experiments were conducted under two levels, namely statement level and the functions level. Statement level experiment includes checking for the correctness of statements and the function considers the number of conditions used in the model.

In the implementation of FY-SSOA, all the test cases are selected based on their objective values attained for the factors. The values of all factors for each test case were investigated and the main test cases that have the defined factor values were selected. Then, the selected test cases were prioritized using the ED-EPCA algorithm. In ED-EPCA implementation, the factors that cause significant differences in APFD values are considered, where there were a number of levels with each factor and 100 test cases per program. Then, the selected test cases undergo a mutation testing process to ensure the effectiveness of the generated test cases. The mutation testing is carried out by using the mutation testing tool called MuClipse for Eclipse. The mutation testing involves statement mutation, which replaces the statements with various kinds of statements, value mutation, which alters the values to identify the errors, and decision mutation, which modifies the arithmetic or logical operators to detect the errors. Thus, mutation testing helps to reach an efficient model performance. After mutation testing is done,

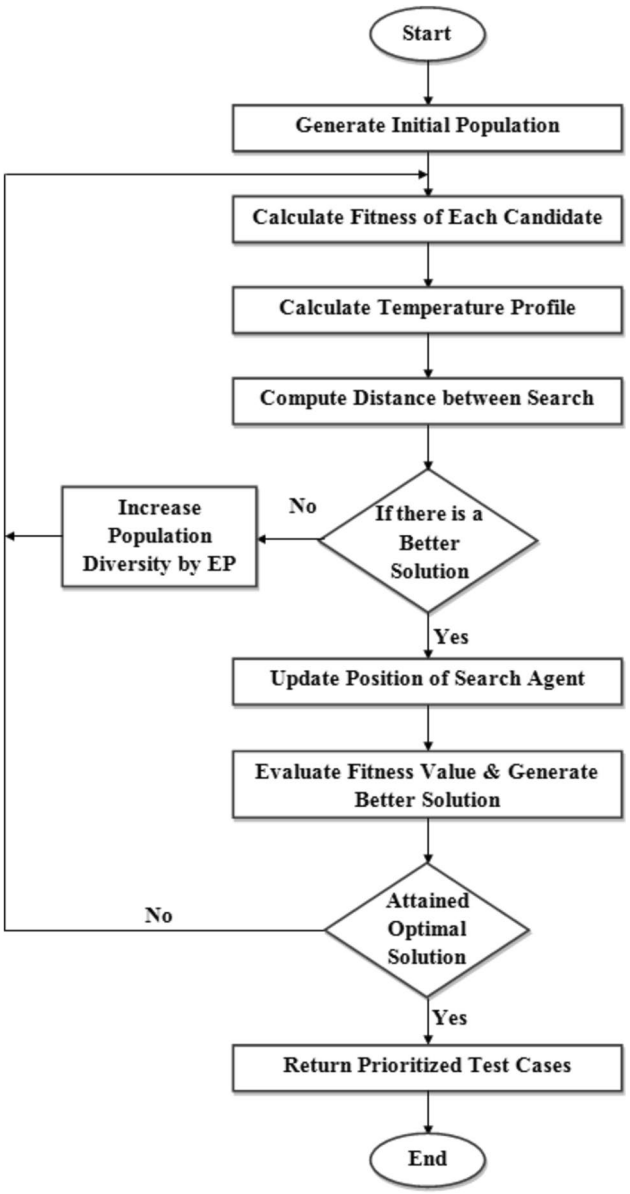

Fig. 3 Flowchart of LD-EPCA

the test cases are subjected to the test prioritization phase. Here, four prioritization techniques along with the proposed ED-EPCA were employed per experiment. The factors within each program generating a maximum of observations, each including an APFD value per experiment were obtained. Under these values, the test cases are prioritized.

4.2 Benchmark Programs and Evaluation Criteria

For this work, a sample healthcare application from the internet has been selected. This healthcare application contains phases, such as Patient Registration, Patient Login, Doctor Registration, Doctor Login, Book Appointment, Consultation, Medical Record upload, Bill payment, Pharmacy details, Discharge summary details form, etc. Each and every form has several operations. Next, the test cases are manually generated based on the model test cases available on the internet for all operations in the forms. Next, the healthcare application is run, and simultaneously, the created test cases are checked with the predicted output. Based on that, the test case result is mentioned as Pass or Fail. This completed test case is considered as the input to our framework to evaluate the system.

Based on the factors, the optimal test cases are selected and prioritized using the modified techniques and compared with existing techniques. Software bugs are categorized into various kinds of bugs according to their distinct nature and influence. Some of the software bugs include functional bugs, logical bugs, workflow bugs, unit-level bugs, system-level integration bugs, out-of-bound bugs, and security bugs that arise during the process of software testing. A functional bug indicates a defect in software applications, which may occur during coding errors, insufficient testing, and other factors related to hardware limitations. A logical bug happens when the code does not provide the expected results. The malfunctions, fault, and bottleneck issue within the working procedure refers to the workflow bug. During testing, if the tiniest element is affected by such errors, then it is considered as a unit-level bug. System-level integration bugs arise when the number of systems collaborate and reach failure or errors. Out-of-bound bugs is defined as the occurred error, which crosses the limit of the specific operation. The flaws or risks in the software and hardware are resulted in security bugs. The evaluation indicators like Average Percentage of Faults Detected (APFD), time, and memory usage are the parameters espoused for measuring all the algorithms' performance (Table 1).

$$Memory\ usage = Total\ memory - Free\ Memory \tag{47}$$

$$APFD = 1 - \frac{sum\ of\ test\ cases}{set\ of\ test\ cases * set\ of\ faults} + \frac{1}{2 * set\ of\ test\ cases} \tag{48}$$

The configuration and implementation parameters used in the ED-EPCA heuristic algorithm are shown in Table 2.

4.3 Performance Measurement of Proposed FY-SSOA

Glowworm Swarm Optimization (GSO), Shuffled Frog Leaping Algorithm (SFLA), Seagull Optimization Algorithm (SOA), and SSOA are the baseline techniques with which the proposed FY-SSOA model is compared regarding memory usage in Fig. 2.

To make a cost-effective test case selection, the proposed method's effectiveness is evaluated based on memory usage.

Table 1 Analysis of Literature Survey

Researcher	Objective	Methodology	Advantages	Limitations
Gao [11]	To contribute to different fault correction rates, fault detection rates, and different fault introduction rates.	A cost calculation method, encompassing the debugger assignment and software release time.	According to the model, Optimal testing resource allocation and the optimal release time can be attained.	Ineffective whilst working in a complex environment.
Xiao et al. [29]	Cost-effective prediction system for the FDP and the FCP.	ANN	Regarding software reliability along with total cost, the optimal release technique was attained.	Higher Mean Square Error (MSE).
Gokilavani and Bharathi [13]	To investigate faults in the software.	Principle Component Analysis (PCA) extraction along with a K-Means clustering-centric ranking model.	The sum of the faults was detected superiorly.	Lacks the information regarding basic needs along with risks related to bugs.
Lin et al. [19]	To determine the FD and Diagnosis (FDD) model's supremacy.	Condition-centric conventional system.	The model deterred unintentional lighting runtime complications.	The dataset utilized was limited.
Cui et al. [8]	Combining spectrum and mutation for enhancing the fault localization accuracy.	Spectrum-centric fault localization (SBFL), mutation-centric fault localization (MBFL), and ranking.	The adopted ranking methodology enhanced the fault localization accuracy.	The programs with multiple faults were not considered.
Jahan et al. [15]	Risk-centric testing by automating the risk assessment process, along with finding high-risk faults early.	Semi-automatic risk-centric test case prioritization technique.	The technique improved test efficiency by spotting defects early overall and even earlier in the high-risk modules.	Potential errors were increased owing to the number of modified requirements, complexity, along with size.
Nagaraju et al. [23]	To formulate the test allocation problem that maximizes fault discovery.	Discrete Cox hazards methodology.	The optimal test activity allocation problem was addressed.	Practical usage was not successful.
Nithya and Chitra [24]	To eliminate unwanted and redundant test data apart from maximizing fault detection.	Gradient-based approaches.	Unwanted and redundant data were removed and more faults were detected.	The convergence rate was affected by the usage of random permutations.
Choudhary et al. [7]	Optimizing the software's launch time by minimizing the total cost and considering fault detection and correction separately.	Testing effort-based software reliability growth system.	The total incurred cost was mitigated by the reduction in the launch time.	Increased fault correction complexity.
Mahdieh et al. [21]	To improve coverage-centric TCP techniques by concerning the fault-proneness distribution over code units.	Coverage-based TCP techniques.	Applicable to different coverage-centric TCP techniques.	Inaccurate prediction of faults due to the utilization of the lower number of bugs.
Bagherzadeh et al. [5]	For continuously and automatically learning a test case prioritization strategy.	Reinforcement Learning (RL).	Higher ranking accuracy of the testing process was obtained.	The model was time-consuming.
Chi et al. [6]	To improve bug detection capability by removing redundant test cases.	Additional Greedy method-centric Call sequence (AGC).	The TC minimization mode reduced the test suite with the elimination of redundant test cases.	The testing process ceased prematurely at a certain arbitrary point.
Ali et a. [1, 2]	To enhance fault detection ability by test case prioritization and selection.	Change Test cases and Failed Frequency (CTFF) based prioritization methodologies.	The model effectively detected more faults.	The outcomes were not validated statistically.

Table 1 (continued)

Researcher	Objective	Methodology	Advantages	Limitations
Ali et al. [1, 2]	To develop a test case selection and prioritization technique for increasing the fault detection rates.	Pattern-centered verification model.	Augmented the FDR together with random priority.	Reliability along with ambiguity issues occurred.
Huang et al. [14]	To increase fault detection rate by the combination of code coverage and combination coverage.	Regression TCP (RTCP).	The model obtained comparable testing efficacy.	Affected by the time overhead problem.
Arasteh et al. [3]	To minimize the time and cost of the mutation test.	Artificial Bee Colony algorithm.	Reduced generated mutants and the mutation testing's cost.	The model attained sub-optimal results owing to the improper exploitation capability of the algorithm.
Arasteh et al. [4]	To automatically generate test cases with maximum branch coverage in a limited amount of time.	Imperialist Competitive Algorithms (ICA).	The algorithm outperformed the other algorithms.	The memory usage during test case generation was not considered.
Khari et al. [16]	To improve the efficiency of automated software testing using test suite generation and test suite optimization.	Automated testing tool.	A set of minimal test cases with maximum path coverage is provided.	Accomplished less path coverage.
Khari et al. [17]	To evaluate six metaheuristic algorithms for optimizing the path coverage along with branch coverage.	Six Heuristic Algorithms.	Demonstrated the algorithms' effectiveness for path coverage-centric optimization.	The number of test cases generated was limited.

Table 2 Configuration and implementation parameters of ED-EPCA

Parameters	Values
b	0.34 m
c	0.16 m
R	0.02 m
S	0.11 m
L	0.28 m
M	0.065 m

The evaluation of memory usage indicates the level of memory demand required for the entire process of test case selection. The memory usage is evaluated by subtracting the available memory after the test case selection process from the total memory of the system. Here, to select 500 TCs, the proposed model utilized a memory space of 6689554 kb. Conversely, the conventional GSO occupied 8245475 kb, which is greater than the proposed one. Similarly, for selecting various numbers of TCs, the memory space occupied by certain other methodologies like SOA, SFLA, and SSOA is also higher than in the proposed system. Thus, it is clear that the TCs are selected more apparently by the proposed system than the other methodologies. The comparative analysis of the proposed FY-SSOA is tabulated in the below table:

From Table 3, it is clear that the proposed FY-SSOA technique achieves better fitness value than the prevailing methodologies. Thus, for 100 iterations, the proposed method attains a fitness value of 6048, whereas the prevailing methodologies like GSO, SOA, SFLA, and SSOA attained 2054, 3087, 4024, and 5087 fitness values, which are considerably lower. Similarly, for certain other iterations, the fitness value differs. Thereby, the proposed model outperforms the other prevailing methodologies. Figure 3 illustrates the performance analysis based on the selection time (Fig. 4).

As per Fig. 5, to select 200 test cases, the proposed FY-SSOA takes 6475 ms, whereas the traditional SFLA model takes 8054 ms. Similarly, to select 300 test cases, the proposed one and the prevailing SFLA take 8245 ms and 10457 ms, respectively. Likewise, the selection time differs (higher) for other methodologies also. Therefore, it is clear

Table 3 Comparative measure of the proposed FY-SSOA based on fitness vs iteration

Iteration	Fitness				
	GSO	SOA	SFLA	SSOA	Proposed FY-SSOA
100	2054	3087	4024	5087	6048
200	2147	3130	4158	5138	6178
300	2236	3246	4258	5248	6238
400	2348	3358	4387	5348	6395
500	2487	3478	4486	5418	5432

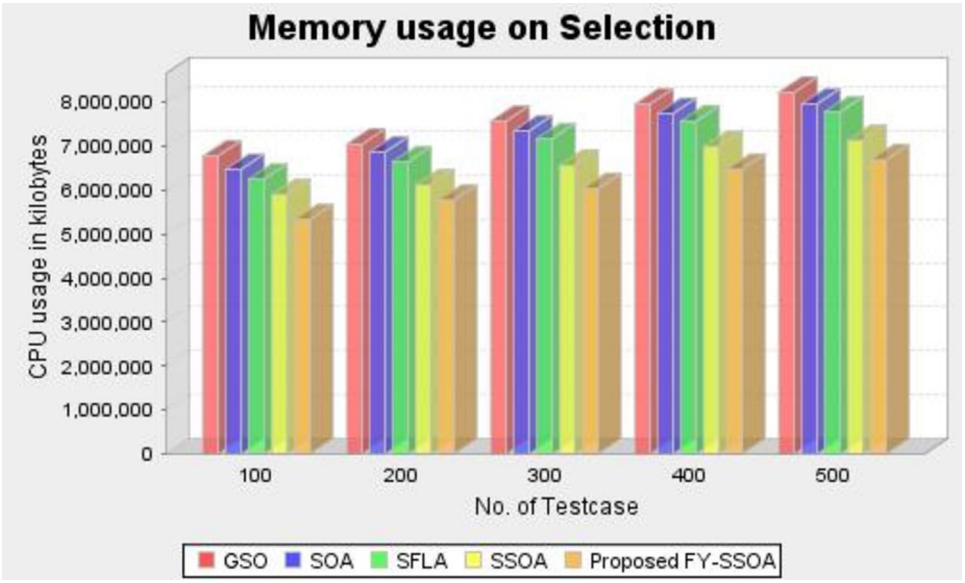

Fig. 4 Performance measure based on memory usage

that when compared with the baseline techniques, the proposed one achieved good performance.

4.4 Performance Evaluation of Proposed ED-EPCA

Cat Swarm Optimization (CSO), Emperor Penguins Colony Algorithm (EPCA), and Penguin Search Optimization Algorithm (PeSOA) are the prevailing techniques with which the proposed model is contrasted based on memory usage in Table 2. The proposed system implemented the CSO, EPCA, and PeSOA methods in the working platform of Java using the Java programming language.

Regarding memory usage, the proposed ED-EPCA model's performance is demonstrated in Table 4. Depending on the number of test cases, the memory usage differs. For prioritizing, the memory utilized by the proposed model is 5421775 kb for 100 test cases. But, for the same number of TCs, the prevailing methodologies occupied a larger memory of 6578452 kb (CSO), 6345782 kb (LOA), 6134886 kb (PeSOA), and 5822387 kb (EPCA), respectively. Similarly, for 500 test cases, a minimum memory of 7077865 kb was utilized by the proposed model. However, a higher memory space of 80458723 kb was attained by the traditional CSO model. Correspondingly, the other prevailing methodologies also utilize more memory than the proposed system.

Figure 6 clearly shows the supremacy of the proposed ED-EPCA model in FD. Accordingly, for 10% of TCs, the APFD attained by the proposed model was 67%. Conversely, for the same percentage of TCs, the prevailing PeSOA obtained merely 57%, which is less than the proposed one.

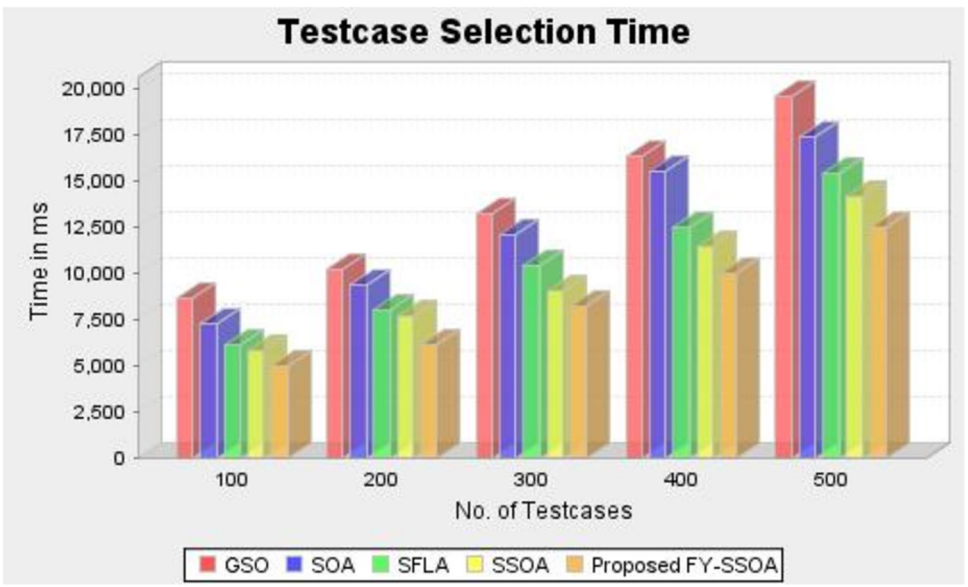

Fig. 5 Performance analysis based on selection time

Table 4 Performance comparison of the proposed ED-EPCA based on memory usage

No. of test cases	CSO	LOA	PeSOA	EPCA	Proposed ED-EPCA
100	6578452	6345782	6134886	5822387	5421775
200	6955214	6745888	6455783	6044683	5732321
300	7584653	7326558	7133674	6520500	6033761
400	7745896	7566324	7355673	7065100	6422786
500	80458723	7865447	7755284	7540340	7077865

Alternatively, the proposed system detected 78% of faults for 30% of TCs, whereas the prevailing PeSOA attained 66%. The percentage of faults detected by the proposed model increases (99.23%) as the number of TCs increases (70%). Thus, it is clear that when contrasted with the existing methodologies, the proposed system detected more faults with higher percentages of APFD.

Figure 7 graphically represents the proposed ED-EPCA model's performance regarding fitness vs iteration. The performance will be better with a higher fitness value.

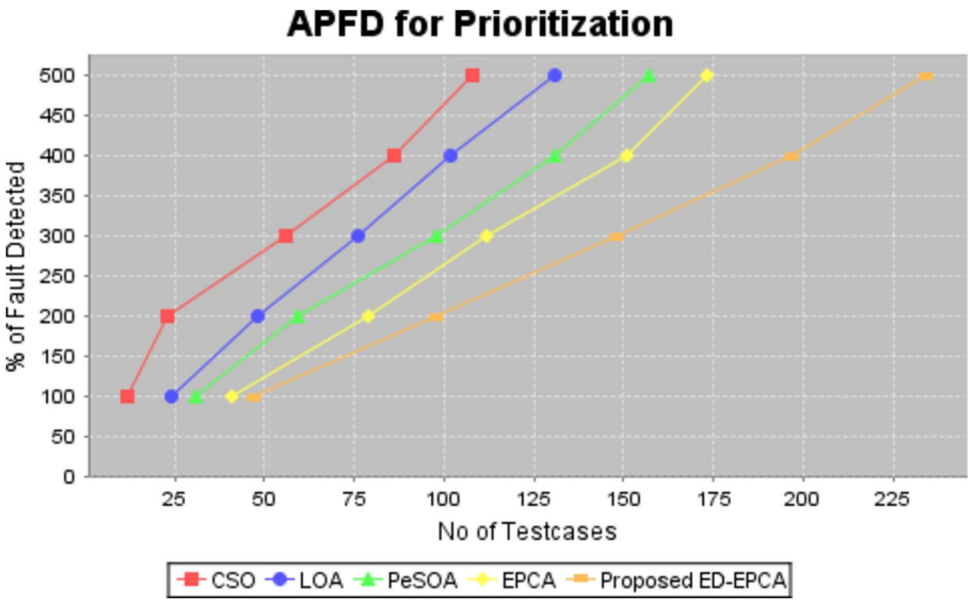

Fig. 6 Performance investigation based on APFD

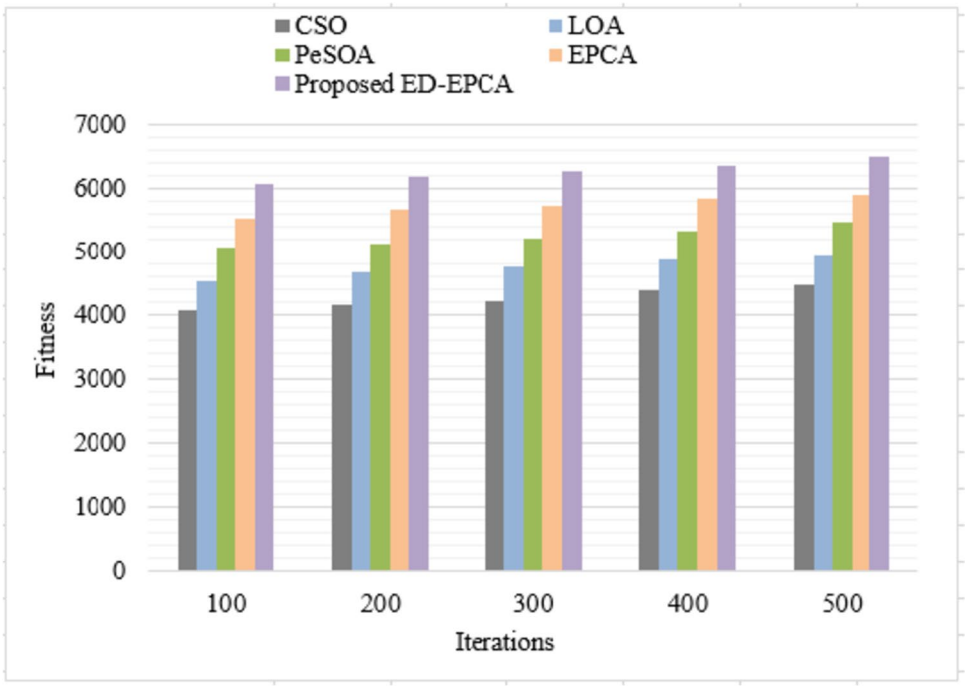

Fig. 7 Superiority measure based on fitness vs. iteration

Table 5 Comparative measurement of the proposed ED-EPCA based on the Prioritization time

Number of test cases	CSO	LOA	PeSOA	EPCA	Proposed ED-EPCA
100	9846	9124	8745	7124	5847
200	19475	17632	15423	12457	9635
300	28473	26548	20154	17845	13457
400	37485	35412	31548	27658	23564
500	46841	41235	37652	34657	28647

Consequently, for 200 iterations, the proposed model attained the fitness value of 6187, whereas the traditional CSO, LOA, PeSOA, and EPCA methodologies obtained 4157, 4687, 5145, and 5684, which are lower than the proposed one. Similarly, for varying iterations like 100, 300, 400, and 500, the proposed model attained higher fitness values than the prevailing methodologies.

Regarding prioritization time, the comparative evaluation of the proposed approach is illustrated in Table 5. Here, to prioritize 100, 200, 300, 400, and 500 tasks, the proposed ED-EPCA takes a prioritization time of 5847 ms, 9635 ms, 13457 ms, 23564 ms, and 28647 ms, respectively. On the other hand, for 100, 200, 300, 400, and 500 tasks, the prevailing EPCA model takes the prioritization time of 7124 ms, 12457 ms, 17845 ms, 27658 ms, and 34657 ms, respectively, which are greater than the proposed model. Therefore, the proposed system shows superior performance than the other traditional algorithms.

Table 6 shows the comparative analysis of the proposed fault detection technique with the existing methods developed by Gokilavani and Bharathi [13], Jahan e al. [15], Mahdieh et al. [21], Bagherzadeh et al. [5], and Ali et al. [1, 2] in Section 2. The analysis was conducted based on APFD. Table 6 indicates that the proposed approach is more efficient compared to the existing techniques, with 99.23% of fault detection capability. Although the system introduced by [15] attained 98% of APFD, it lacks when considering the parameters, such as requirements, complexity, along with size. Hence, by making a cost-effective test case prioritization, selecting the most significant test cases, and incorporating more strength factors, the proposed method advances the fault detection process compared to the existing methods.

Table 6 Comparative Analysis

Methods	APFD (%)
Proposed method	99.23
Gokilavani and Bharathi [13]	95.26
Jahan et al. [15]	98
Mahdieh et al. [21]	60.5
Bagherzadeh et al. [5]	77.2
Ali et al. [1, 2]	93

5 Conclusion

By employing ED-EPCA-based TCP, an effectual FD system has been proposed here via FY-SSOA-based TCS. For an efficient FD in the software test cases, numerous operations have been encompassed in this model. Next, to endorse the proposed model's efficiency, the experiential analysis is conducted, where the performance along with a comparative evaluation of the proposed framework is performed. Various uncertainties were handled by the developed model, thus rendering highly promising outcomes. For the assessment, an openly accessible dataset is utilized. In this work, the proposed model attained 99.23% APFD. In the proposed model, for 500 TCs, the selection process is completed within 12471 ms. In addition, it occupied a memory space of 6689554 kb. Moreover, for prioritizing 500 tests, the proposed technique, which occupies 7077865 kb of memory space, takes 28647 ms. However, utilizing an optimization model to select as well as prioritize test cases leads to higher computation time. The proposed model can be contributed as a checklist guideline for the identification of faults in software testing. The technique renders a way for scheduling and running test cases, which possess the highest priority, earlier for rendering earlier feedback to software testing engineers. As the model was developed to support projects that are developed using the Java programming language, it has not been applied to any other programs. Hence, the model can be extended to be applicable to the programs developed using other technologies. Thus, with some modified deep learning methodologies, the work would be extended in the future to ameliorate the FD with the historical test case dataset. In addition, the work would subsume the modified optimization process aimed at optimal factor selection.

Acknowledgements We thank the anonymous referees for their useful suggestions.

Author's Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by [*1]J Paul Rajasingh, [2]P.Senthil Kumar, [3]S.Srinivasan. The first draft of the manuscript was written by[1]J Paul Rajasingh and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding This work has no funding resource.

Data Availability Statement Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

Declarations

Ethical Approval This article does not contain any studies with human participants or animals performed by any of the authors.

Consent of Publication Not applicable.

Competing Interests The authors declare that they have no competing interests.

References

1. Ali S, Hafeez Y, Hussain S, Yang S (2020) Enhanced regression testing technique for agile software development and continuous integration strategies. Software Qual J 28(2):397–423. https://doi.org/10.1007/s11219-019-09463-4
2. Ali S, Hafeez Y, Jhanjhi NZ, Humayun M, Imran M, Nayyar A, Singh S, Ra I (2020) HTowards Pattern-Based Change Verification Framework for Cloud-Enabled Healthcare Component-Based. IEEE Access 8:148007–148020. https://doi.org/10.1109/ACCESS.2020.3014671
3. Arasteh B, Imanzadeh P, Arasteh K, Gharehchopogh FS, Zarei B (2022) A source-code aware method for software mutation testing using artificial bee colony algorithm. J Electron Test 38(3):289–302
4. Arasteh B, Hosseini SMJ (2022) Traxtor: an automatic software test suit generation method inspired by imperialist competitive optimization algorithms. J Electron Test 38(2):205–215
5. Bagherzadeh M, Kahani N, Briand L (2021) Reinforcement Learning for Test Case Prioritization. IEEE Trans Softw Eng 5589(c):1–21. https://doi.org/10.1109/TSE.2021.3070549
6. Chi J, Qu Y, Zheng Q, Yang Z, Jin W, Cui D, Liu T (2020) Relation-based test case prioritization for regression testing. J Syst Softw 163. https://doi.org/10.1016/j.jss.2020.110539
7. Choudhary C, Kapur PK, Khatri SK, Muthukumar R, Shrivastava AK (2020) Effort based release time of software for detection and correction processes using MAUT. International Journal of System Assurance Engineering and Management 11:367–378. https://doi.org/10.1007/s13198-020-00955-2
8. Cui Z, Jia M, Chen X, Zheng L, Liu X (2020) Improving software fault localization by combining spectrum and mutation. IEEE Access 8:172296–172307. https://doi.org/10.1109/ACCESS.2020.3025460
9. Dadkhah M, Araban S, Paydar S (2020) A systematic literature review on semantic web enabled software testing. J Syst Softw 162:110485. https://doi.org/10.1016/j.jss.2019.110485
10. Danglot B, Monperrus M, Rudametkin W, Baudry B (2020) An approach and benchmark to detect behavioral changes of commits in continuous integration. Empir Softw Eng 25(4):2379–2415. https://doi.org/10.1007/s10664-019-09794-7
11. Gao K (2021) Simulated Software Testing Process and Its Optimization Considering Heterogeneous Debuggers and Release Time. IEEE Access 9:38649–38659. https://doi.org/10.1109/ACCESS.2021.3064296
12. Garousi V, Bauer S, Felderer M (2020) NLP-assisted software testing: A systematic mapping of the literature. Inf Softw Technol 126:106321. https://doi.org/10.1016/j.infsof.2020.106321
13. Gokilavani N, Bharathi B (2021) Test case prioritization to examine software for fault detection using PCA extraction and K-means clustering with ranking. Soft Comput 25(7):5163–5172. https://doi.org/10.1007/s00500-020-05517-z
14. Huang R, Zhang Q, Towey D, Sun W, Chen J (2020) Regression test case prioritization by code combinations coverage. J Syst Softw 169:110712. https://doi.org/10.1016/j.jss.2020.110712
15. Jahan H, Feng Z, Mahmud SMH (2020) Risk-Based Test Case Prioritization by Correlating System Methods and Their Associated Risks. Arab J Sci Eng 45(8):6125–6138. https://doi.org/10.1007/s13369-020-04472-z
16. Khari M, Kumar P, Burgos D, Crespo RG (2018) Optimized test suites for automated testing using different optimization techniques. Soft Comput 22:8341–8352
17. Khari M, Sinha A, Verdu E, Crespo RG (2020) Performance analysis of six meta-heuristic algorithms over automated test suite generation for path coverage-based optimization. Soft Comput 24(12):9143–9160
18. Lima JAP, Vergilio SR (2022) A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments. IEEE Trans Software Eng 48(2):453–465. https://doi.org/10.1109/TSE.2020.2992428
19. Lin G, Kramer H, Granderson J (2020) Building fault detection and diagnostics: Achieved savings, and methods to evaluate algorithm performance. Build Environ 168:106505. https://doi.org/10.1016/j.buildenv.2019.106505
20. Ma P, Cheng H, Zhang J, Xuan J (2020) Can this fault be detected: A study on fault detection via automated test generation. J Syst Softw 170:110769. https://doi.org/10.1016/j.jss.2020.110769
21. Mahdieh M, Mirian-Hosseinabadi SH, Etemadi K, Nosrati A, Jalali S (2020) Incorporating fault-proneness estimations into coverage-based test case prioritization methods. Inf Softw Technol 121(January):106269. https://doi.org/10.1016/j.infsof.2020.106269
22. Mukherjee R, Patnaik KS (2021) A survey on different approaches for software test case prioritization. J King Saud Univ Comput Inf Sci 33(9):1041–1054. https://doi.org/10.1016/j.jksuci.2018.09.005
23. Nagaraju V, Jayasinghe C, Fiondella L (2020) Optimal test activity allocation for covariate software reliability and security models. J Syst Softw 168:110643. https://doi.org/10.1016/j.jss.2020.110643
24. Nithya TM, Chitra S (2020) Soft computing-based semi-automated test case selection using gradient-based techniques. Soft Comput 24(17):12981–12987. https://doi.org/10.1007/s00500-020-04719-9
25. Prado LJ, A & Vergilio S. R. (2020) Test Case Prioritization in Continuous Integration environments: A systematic mapping study. Inf Softw Technol 121:106268. https://doi.org/10.1016/j.infsof.2020.106268
26. Raju S, Uma GV (2012) Factors oriented test case prioritization technique in regression testing using genetic algorithm. Eur J Sci Res 74(3):389–402
27. Santos I, Melo SM, Lopes De Souza PS, Souza SRS (2020) Towards a unified catalog of attributes to guide industry in software testing technique selection. Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020 pp. 398–407. https://doi.org/10.1109/ICSTW50294.2020.00071
28. Shrivastava AK, Kumar V, Kapur PK, Singh O (2020) Software release and testing stop time decision with change point. Int J Syst Assur Eng Manag 11:196–207. https://doi.org/10.1007/s13198-020-00988-7
29. Xiao H, Cao M, Peng R (2020) Artificial neural network based software fault detection and correction prediction models considering testing effort. Appl Soft Comput J 94:106491. https://doi.org/10.1016/j.asoc.2020.106491
30. Yucalar F, Ozcift A, Borandag E, Kilinc D (2020) Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability. Eng Sci Technol 23(4):938–950. https://doi.org/10.1016/j.jestch.2019.10.005

J. Paul Rajasingh received his BE in Computer Science and Engineering from Thiagarajar College of Engineering, Madurai and ME in Software Engineering from College of Engineering, Guindy campus, Anna University. He is pursuing his PhD in Information and Communication Engineering in Anna University. His current research interests include Software Engineering and Big Data analytics.

P. Senthil Kumar holds his PhD in Computer Networks and is currently a professor in the Department of Information Technology at P. B. College of Engineering, Irungattukottai, Chennai-602117. He has 20 years of experience in teaching and research. His research interests include Networks, Software Engineering, Internet of Things and Cloud Computing.

S. Srinivasan holds his PhD in Information and Communication Engineering and is currently working as a professor in the Department of Computer Science and Engineering at R.M.D. Engineering College, Kavaraipettai, Tiruvallur District, India. He has 20 years of experience in teaching and research. His research interests include Mobile Computing, Software Engineering and Image Processing.