International Journal of Scientific Research in Computer Science and Engineering
Vol.**13**, Issue.**3**, pp.**30-39**, June **2025**
E-ISSN: 2320-7639
Available online at: www.isroset.org

Research Article

QAG-TCP Tool for Effective Test Prioritization of Object-Oriented Programs: Design and Implementation

Hassan Abubakar[1*], Fatima Zambuk[2]

[1]Dept. of Computer Science, Faculty of Physical and Computing Sciences, Usmanu Danfodio University, Sokoto, Nigeria
[2]Faculty of Science, Department of Mathematical Science, Abubakar Tafawa Balewa University, Bauchi, Nigeria

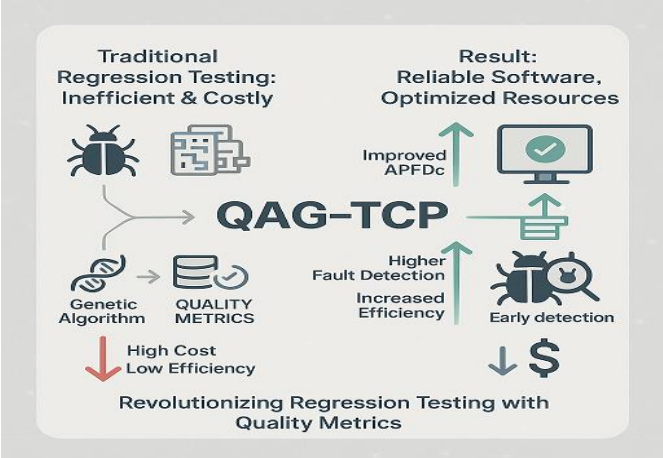
**Corresponding Author:* ✉

Received: 24/Apr/2025; **Accepted**: 26/May/2025; **Published**: 30/Jun/2025. | **DOI**: https://doi.org/10.26438/ijsrcse.v13i3.702

Abstract— Regression testing, a crucial software development phase, can be time-consuming due to the need to re-execute all test cases after code changes. It has been established that Complexity measurement is an essential component for ensuring Software Quality is maintained which in turn determines the success of fault defection in Object- Oriented Program Codes. Lack of complexity measurement mechanism makes some of the existing object-oriented based test case prioritization techniques often not effectively identifying faults early in the testing process. This inefficiency leads to increased testing costs, decrease in Percentage of fault detection, Overall Test Effort in detecting faults, and Average percentage of fault detection rate per cost. In this paper we present an improved technique that handles complexity measurement by incorporating Software Quality measures, the technique will also consider vary fault severity/cost using real faults for Object Oriented programs. This will significantly improve the overall effectiveness in fault detection which will enable developers to identify faults more accurately and efficiently in Object-Oriented Program. The proposed technique is implemented as an automated tool that integrates with existing testing frameworks. Some early results indicates significant improvement

Keywords— Regression testing, Quality-aware, cost cognizant, prioritization, test cases, software quality

Graphical Abstract



1. Introduction

The software development lifecycle (SDLC) is a structured process for creating high-quality software applications. It involves several phases, each with its own set of activities designed to ensure the software meets its intended purpose and functions as expected [1]. Testing plays a vital role throughout the SDLC, acting as a safeguard to identify and rectify errors before the software is deployed to real-world users.

One crucial testing category is functional testing, which focuses on verifying if the software behaves according to its documented requirements and specifications[2]. This involves designing test cases that simulate real-world usage scenarios and ensure the software delivers the intended functionality. Testers meticulously examine core features, data processing, user interactions, and error handling mechanisms to identify any deviations from the expected behavior. A functional testing form the foundation for ensuring the software operates as designed and caters to user needs.

As software development progresses, changes and enhancements are inevitably introduced. Regression testing becomes paramount at this stage[3], [4]. It involves re-running a subset of previously designed functional tests to ensure that new changes haven't unintentionally introduced

regressions, or bugs, into existing functionalities. Regression testing safeguards the overall quality and stability of the software by verifying that modifications haven't caused unintended consequences.

However, with a vast number of functional tests at their disposal, testers often face a critical decision: which tests to run first during regression testing, especially when time and resources are limited. This is where the concept of test case prioritization comes into play[5]. It's a strategic approach that involves ranking test cases based on specific criteria, aiming to maximize the testing process's effectiveness and efficiency. Test case prioritization techniques consider various factors when assigning priorities. Some techniques focus on the likelihood of a test case uncovering a bug (fault detection rate). Others might prioritize tests that cover critical functionalities or those that are more cost-effective to run (considering execution time or resource requirements). By strategically prioritizing test cases, testers can focus their efforts on high-risk areas first, increasing the probability of catching regressions early on in the development cycle.

The benefits of effective test case prioritization are numerous. Maximize fault detection, optimize resource allocation, reduce testing time and improve software quality. Test case prioritization empowers testers to strategically select which functional test cases to run first during regression testing, maximizing fault detection, optimizing resource allocation, and ultimately leading to the development of high-quality software.

In as much as the importance and strategies of the existing test case prioritization is recognized, significant gaps still exist especially in Object-Oriented Programs paradigm (OOP) within the dynamics landscape of the Fourth Industrial Revolution. Many current techniques lack mechanism for incorporating software complexity measures, an essential component for maintaining software quality in OOP. Furthermore, several techniques are not explicitly designed for OOP paradigms[4],frequently depends on artificial faults simulation instead of real-world fault data [6] and often consider fault severity and detection cost uniform [7],[8]. These collective limitations affect the overalls optimal performance of the existing techniques in terms of percentage of fault detection, test effort efficiency and average percentage of fault detection per cost (APFDc). These translate to less effective prioritization, higher testing costs, and prolonged debugging cycle for complex OOP applications.

Driven by these identified limitations, this research aims to develop a robust test case prioritization technique that addresses the aforementioned limitations. To be precise, the main objective of this research is to develop a new technique that effectively handles complexity measurement by incorporating Software Quality measures, accurately accounts for varying fault severity and cost using real fault data, and is specifically meant for Object-Oriented Programs. By achieving this, this research seeks to brings significant improvement overall effectiveness in fault detection, thereby enables developers to trace faults more accurately and efficiently in diverse OOP environment

1.1 Objective of the Study

In order to greatly improve the efficacy and efficiency of regression testing for object-oriented applications, the main goal of this research is to create and empirically validate the Quality-aware GA-based cost-cognizant test case prioritization (QAG-TCP) technique. In particular, this study aims to demonstrate QAG-TCP's ability to detect faults earlier and more thoroughly, tackling the crucial problem of preserving software quality and dependability in the face of constant changes occurring during software development cycles.

1.2 Organization

This article is organized into the following sections: Section 1 introduces the background of test case prioritization, the problem addressed, and the study's objectives. Section 2 contains a review of related work on test case prioritization and relevant software quality metrics. Section 3 elaborates on the theoretical foundations and key calculations, including the criticality, quality index, and QFitness functions. Section 4 details the architecture and essential steps involved in the QAG-TCP tool's design and operational requirements. Section 5 explains the detailed QAG-TCP methodology and implementation, outlining its core algorithms. Section 6 describes the empirical results, discussion of QAG-TCP's performance, and an illustrative example of its application. Section 7 provides recommendations and practical insights derived from the study's findings. Finally, Section 8 concludes the research work and outlines future directions for enhancement and application.

2. Related Work

Test case prioritization (TCP) is fundamental to effective regression testing and software reliability. Recent academic efforts have introduced innovative strategies, blending artificial intelligence, advanced code analysis, and evolutionary algorithms, all aimed at boosting the efficiency and effectiveness of fault detection in complex software. A systematic literature review by [9] offers a comprehensive overview of TCP methods, classifying existing methodologies by their principles, objectives, and evaluation metrics. Their work confirms TCP's status as a highly relevant and current topic, drawing significant interest from the software engineering community.

Manikkannan and Babu [10] introduce a novel TCP technique that ingeniously employs an embedded autoencoder model. This AI-driven strategy focuses intently on learning highly effective, condensed representations of test cases, a process that in turn critically facilitates their optimal prioritization. The overarching aim here is a substantial enhancement in regression testing efficiency, leading directly to markedly higher fault detection rates. In a parallel vein, Zhu and Liu [11] present a TCP algorithm specifically conceived around the principle of improved code coverage. Their methodology strategically prioritizes test cases by maximizing the coverage

achieved over newly implemented or modified code segments, with empirical evaluations consistently demonstrating its strong capability to both boost fault detection and streamline the entire regression testing process.
Beyond the realms of AI-driven representation learning and direct code coverage, other significant contributions tackle distinct facets of TCP. Ahmed et al. [12] put forth a value-based cost-cognizant TCP approach, which meticulously accounts for both the severity of identified faults and the associated cost of executing individual test cases throughout the prioritization process. This holistic consideration is shown to lead to demonstrably improved regression testing effectiveness. In the fascinating field of metaheuristics, Raamesh et al. [13] developed HBR2O, a novel hybrid algorithm that artfully combines elements from both the Battle Royale Optimization and Remora Optimization algorithms. HBR2O offers an exceptionally efficient and highly effective pathway for test case selection and prioritization, having consistently showcased superior performance when benchmarked against existing state-of-the-art algorithms. Its strength lies in its ability to pinpoint high-quality test cases while simultaneously optimizing resource consumption, particularly in terms of processing time and memory.

The unique and often intricate challenges presented by object-oriented software, where complex interdependencies can significantly complicate the precise re-execution of test cases, have also spurred the development of highly specialized TCP solutions. Yadav and Dutta [14], for example, propose a technique grounded in the utilization of a dependency graph. This visual representation meticulously charts the interconnections between various software components, enabling a notably cost-effective strategy for prioritizing regression tests within object-oriented systems. Their mutGA technique, which strategically incorporates mutation operations on test cases, notably achieved superior APFD (Average Percentage of Faults Detected) values when rigorously benchmarked against more traditional methods like retest-all. Similarly, Bello et al. [15] contributed the ECRTP technique, an evolutionary TCP specifically designed for object-oriented programs that profoundly leverages fault dependencies. By carefully assigning a value to test cases based on the severity of their dependent faults, ECRTP aims to ensure the earlier detection of critical issues during regression testing. Experimental results from their studies robustly validate ECRTP's improved performance in terms of both test effort efficiency and overall fault detection effectiveness. Importantly, the authors also thoughtfully outline future research avenues for ECRTP, including the integration of the very latest regression testing information and the consideration of additional object-oriented metrics like coupling and cohesion, all intended to further enhance its capability in ensuring truly robust software quality.

In pursuit of more efficient and effective testing paradigms especially on Object-Oriented Softwares, this research titled "Quality-Aware Genetic Algorithm Based Cost Cognizant Test Case Prioritization for Object-Oriented Programs" offers a particularly compelling synthesis of multiple critical aspects. By ingeniously integrating genetic algorithms for sophisticated optimization, maintaining a clear and practical focus on cost-cognizance, specifically tailoring its prioritization capabilities for the inherent complexities of object-oriented programs, and consistently maintaining a quality-aware perspective throughout the entire process, this type of approach directly confronts the multifaceted and often intricate challenges that define modern software testing. It aims not only to address existing limitations but also to deliver highly effective and exceptionally efficient test solutions that contribute significantly to the overall reliability and enduring robustness of contemporary software systems.

3. Theory or Calculation

This section delineates the fundamental mathematical formulations and theoretical constructs underpinning the Quality-aware GA-based cost-cognizant test case prioritization (QAG-TCP) technique. It elaborates on the critical metrics and the fitness function that drive the prioritization process, integrating concepts of code quality, test case importance, and execution cost.

I. Test Case Criticality

Test case criticality quantifies the significance of a test case in detecting faults, considering the statements it covers and their associated execution cost. It is a foundational element in assessing a test case's value. The criticality for a given test case is formulated as:

$$\boldsymbol{Criticality} = \frac{\sum(\boldsymbol{Criticality\_i})}{\boldsymbol{Cost}} \quad (1)$$

where:

Criticality is the overall criticality of the test case

Criticality_i is the criticality of statement i in the test case

Cost is the cost of running the test case

II. Quality Index (QI)

The Quality Index (QI) provides a quantitative measure of the inherent structural quality of the program's code, particularly focusing on object-oriented design principles. This index integrates established object-oriented metrics to provide a composite score that guides the prioritization towards more fragile or complex code areas. The Quality Index is calculated based on Coupling Between Objects (CBO) and Lack of Cohesion of Methods (LCOM) metrics, aiming to provide a score normalized within a specific range. Two equivalent expressions for the Quality Index are employed:

$$\text{Quality} = (\text{CBO} < \text{LCOM}) + (\text{Math.abs}(\text{LCOM} - 1) < 0.1) - (\text{CBO} > 3 \;\&\&\; \text{LCOM} > 3) + 1 \quad (2)$$

Or

$$\text{Quality} = \min(3, \max(1, \text{CBO} - \text{LCOM} + |\text{LCOM} - 1|)) \quad (3)$$

where:

CBO is the coupling between objects metric
LCOM is the lack of cohesion of methods metric
Quality is the overall quality score

These expressions collectively assess the structural quality, with a higher Quality Index indicating a potentially more complex or critical code module that warrants earlier testing attention.

III. QFitness Function

The QFitness function is the core of QAG-TCP's prioritization logic, serving as the fitness criterion for the Genetic Algorithm. It uniquely combines test case order, fault detection capability (test case award value), and the derived Quality Index of the program's code to provide a comprehensive prioritization score. The QFitness for a sequence of test cases (chromosome) is defined as:

$$Qfitness = \Sigma\,(order * tfi) * Qi \quad (3)$$

where:

fitness is the overall fitness value of the chromosome (test case sequence)

order is the position of the test case in the sequence (starting from 1)

tfi is the test case award value for the corresponding fault

Qi is the quality index

By multiplying these three components, the QFitness function provides a robust metric that prioritizes test cases that appear earlier in the sequence, are more effective at fault detection, and target areas of the code base identified as having lower quality or higher complexity. This multi-faceted fitness evaluation guides the genetic algorithm to evolve test suite prioritizations that are both effective in finding faults and efficient in their application, considering the inherent quality of the software under test.

4. Experimental Method

This section details the design, implementation, and experimental setup of the Quality-aware GA-based cost-cognizant test case prioritization (QAG-TCP) technique. The QAG-TCP technique is implemented as an automated tool that integrates with existing testing frameworks for Object-Oriented programs. We used the tool as a means of putting the entire process of regression test case prioritization by QAG-TCP into practice alongside providing the supporting tools which will be used in the next phase of the research which is performance experiments.

Requirements of QAG-TCP Tool

The QAG-TCP tool required the following:

1. The tool should be able to generate coverage information for each test case selected. The information is used in evaluating the Average Percentage of the rate of Fault Detection of the prioritization approach.
2. Users ought to have the capacity to generate and save the coverage in an arrangement that can be effortlessly recovered for future use, for example, .txt design.
3. Users ought to be able to generate and save test cases as Junit test cases/classes.
4. The tools use the syntactically correct Object-Oriented program as source files for mutant generation and analysis since µJava does not detect errors in the source code. The users should be able to use syntactically correct OOP source files for mutant generation and analysis.
5. Users should be able to know all the mutants, as the mutants are used for the detection of all affected statements from the ESDG.
6. Users should be able to copy the prioritized test cases for future use in mutation analysis and evaluation of APFDc.
7. Users should be able to enter the affected statements as input to the tool, select the source file, and generate the prioritized test cases.

The QAG-TCP Technique's Structure

The term "software system architecture" describes the high-level abstraction of a software system, which is made up of several significant computing components and connectors that explain how these components interact. This research project architecture is conceptually divided into three layers: the application layer, the database, and the presentation layer as utilized by [16] for a similar technique. The QAG-TCP technique implements the QFitness, CKJM-IFN, and QAG-TCP algorithms shown in Figure 2, 3 and 4 to perform the regression testing on the supplied source program.

Conceptual Design

The QAG-TCP tool was planned, developed, and implemented using the Java programming language within the Eclipse IDE (Version: 2023-06 (4.28.0)). This object-oriented implementation is designed to receive affected statements (derived independently for each program/application) and the computed Quality Index of the program. Based on these inputs, QAG-TCP executes its core prioritization logic to generate an ordered sequence of test cases. Figure 1 illustrates the conceptual design, depicting the interaction between input components, the core processing layer, and the output display.

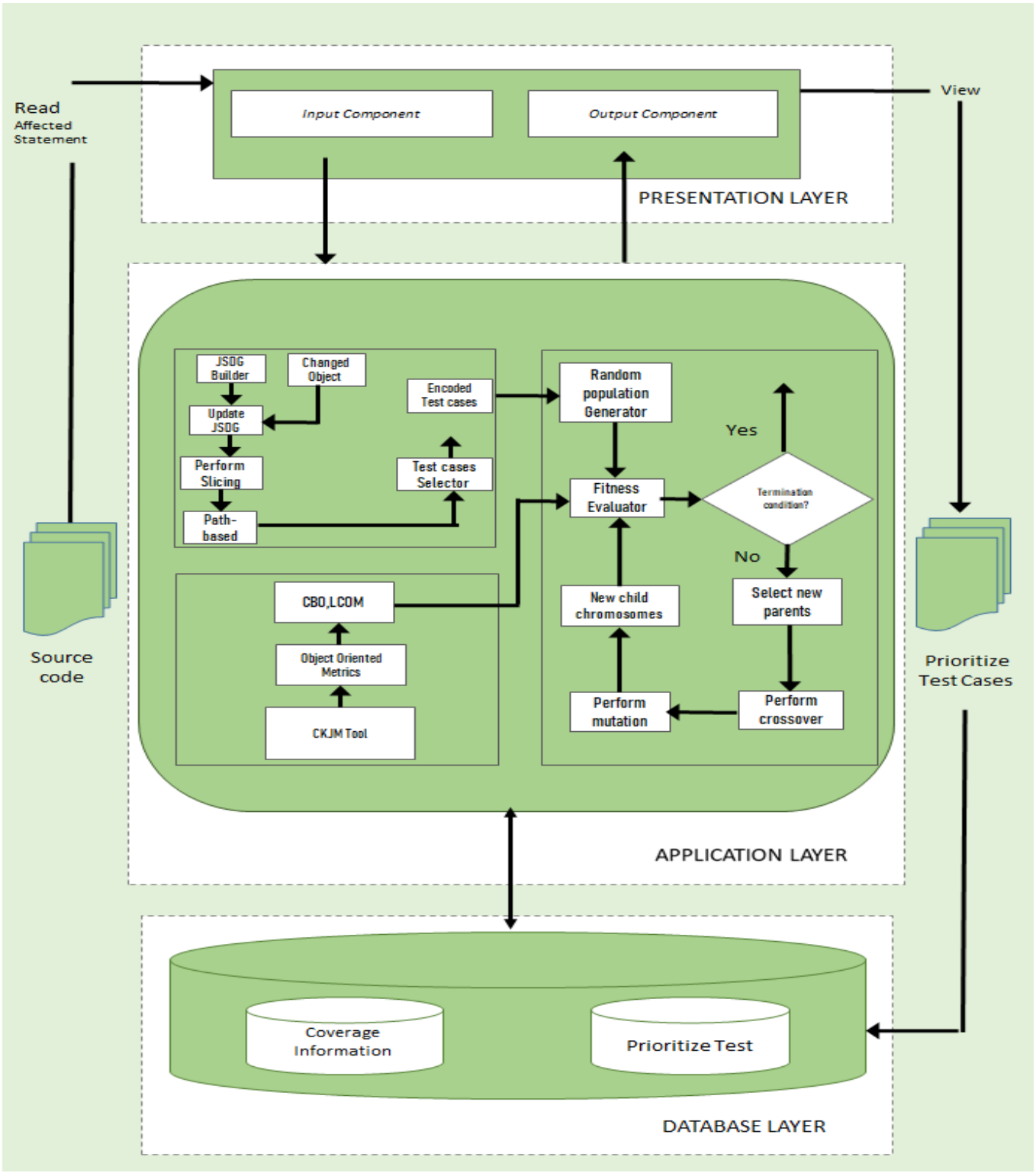


Figure 1: Conceptional Design

Input Component

Syntactically correct source files from the original program were used by the tool. The affected statements are accepted as input into the tool by an input component in the presentation layer. Using each test case's CoverageInfo, the affected statements/nodes are used to determine which test cases are affected.

Output Component

Following the information flow, the user views the output in the output pane at the presentation layer. The output is shown with the test cases prioritized after a certain number of reiterations.

The QAG-TCP Core

QAG-TCP technique implements the CKJM-IFN (Algorithm 1), computeQFitness (Algorithm 2), and QAG-TCP (Algorithm 3) at the application layer of the architecture that performs regression test case prioritization based on the process described by QAG-TCP. These algorithms comprise seven major components, which are a Slicing performer, test case selector, test case encoder, random population generator, fitness evaluator, ordered crossover performer, and swap mutation performer as shown in Figure 1. The main contribution of this research in regression testing is described in detail and how it interacts with other components as well as the other tools used to make the QAG-TCP perform regression test case prioritization.

Quality Indicator Implementation

This study adds a new phase to the entire regression testing process, which replaces the previous one as the third phase: the Quality Indicator phase. This stage was a composite of many components working together to perform different functions.

When a Java program is run, its original program is passed to the Javac command, which separates the program's classes into individual class files. Then, by analyzing the bytecode of compiled Java files, the CKJM program accesses the class files to compute the Object-Oriented metrics. For the specified classes, the application computes the six Object-Oriented metrics (WMC, NOC, CBO, RFC, LCOM, Ca, and NPM). After that, a text file called OCKJM.txt is used to store the output.

Algorithm I: CKJM-IFN Algorithm

Source: Adapted from Abubakar et al. (2023, Algorithm I)

Inputs:

- javaSourceFiles: A collection of Java source files (.java).

Output:

- A text file (CK_Metrics_Output.txt) containing the calculated CK metrics for each class.

Procedure: GenerateCKMetrics

1. **Start**
2. Initialize compiledClasses as an empty collection.
3. Initialize ckMetricsCollection as an empty collection.
4. Define metricsOutputFile as "CK_Metrics_Output.txt".
   // Compile source code to get class files
5. For each sourceFile in javaSourceFiles:
6. `classFile` ← `CompileJava(`sourceFile`)`
7. Add `classFile` to `compiledClasses`.
   // Calculate CK metrics for each compiled class
8. For each class in compiledClasses:
9. `metrics` ← `ExecuteCKJMTool(`class`)`
10. Add `metrics` to `ckMetricsCollection`.
    // Store the collected metrics
11. Open metricsOutputFile for writing.
12. For each metricSet in ckMetricsCollection:
13. Write `metricSet` to `metricsOutputFile`.
14. Close metricsOutputFile.
15. **End**

Algorithm 1: CKJM-IFN Algorithm

Algorithm II: computeQFitness Algorithm

Source: Adapted from Abubakar et al. (2023, Algorithm II)

Terminology:

- **Chromosome:** A candidate solution, representing an entire test suite.
- **Gene:** An individual component of a chromosome, representing a single test case.

Inputs:

- testSuiteChromosome: The test suite to be evaluated.
- metricsFile: The path to the file containing CK metrics (CK_Metrics_Output.txt).

Output:

- fitnessScore: A numerical value representing the overall quality of the testSuiteChromosome.

Procedure: CalculateFitnessScore

1. **Start**

   // Load required data
2. ckMetrics ← Read data from metricsFile.
3. testCases ← Extract individual genes (test cases) from testSuiteChromosome.
   // Evaluate strength based on fault coverage
4. Initialize aggregatedStrength to 0.
5. For each testCase in testCases:
6. `coverageData` ← `AnalyzeFaultCoverage(`testCase`)`
7. `strength` ← `CalculateTestCaseStrength(`coverageData`)`
8. `aggregatedStrength` ← `aggregatedStrength` + `strength`.
   // Evaluate software quality from CK metrics
9. softwareQualityIndex ← DeriveQualityIndex(ckMetrics)

   // Combine metrics into a final fitness score
10. fitnessScore ← CombineScores(aggregatedStrength,softwareQualityIndex)
11. **Return** fitnessScore
12. **End**

Algorithm 2: QFitness Algorithm

Algorithm III: QAG-TCP Algorithm

Source: Adapted from Abubakar et al., 2023, Algorithm III

Inputs:

- P: The Java source code of the program under test.

- T: The complete, unordered test suite.

Output:

- T_prioritized: The test suite T with its test cases ordered for optimal fault detection.

Procedure: PrioritizeTestSuiteWithGA

1. **Start**
   // Phase 1: Initialization and Test Selection
2. model ← BuildSystemDependenceGraph(P).
3. updatedModel ← UpdateModelWithChanges(model).
4. affectedCode ← IdentifyAffectedStatements(updatedModel).
5. coverageInfo ← MapTestCasesToCoverage(T).
6. relevantTestSuite ← SelectTestsCoveringAffectedCode(T, affectedCode, coverageInfo).

   // Phase 2: Genetic Algorithm Optimization
7. encodedPopulation ← CreateInitialPopulation(relevantTestSuite).
8. For each chromosome in encodedPopulation:
9. Calculate its fitness using the **`CalculateFitnessScore`** procedure.
   // Main GA Loop
10. **While** termination condition is not met:

    // Selection
11. `parent1`, `parent2` ← `SelectFittestChromosomes(encodedPopulation)`.

    // Reproduction
12. `offspring` ← `ApplyCrossover(parent1, parent2)`.
13. `mutatedOffspring` ← `ApplyMutation(offspring)`.

    // Fitness Evaluation and Replacement
14. Calculate fitness of `mutatedOffspring` using **`CalculateFitnessScore`**.
15. `encodedPopulation` ← `UpdatePopulation(encodedPopulation, mutatedOffspring)`.
16. **End While**
    // Phase 3: Finalization
17. bestChromosome ← FindBestChromosome(encodedPopulation).
18. T_prioritized ← DecodeChromosomeToTestSuite(bestChromosome).
19. **Return** T_prioritized
20. **End**

Algorithms 3: QAG-TCP Algorithm

Execution of the QAG-TCP Tool

The QAG-TCP tool is presented in Figure 2 as the initial user interface. A dummy dataset is used to illustrate all the various operations that constitute the whole QAG-TCP technique.

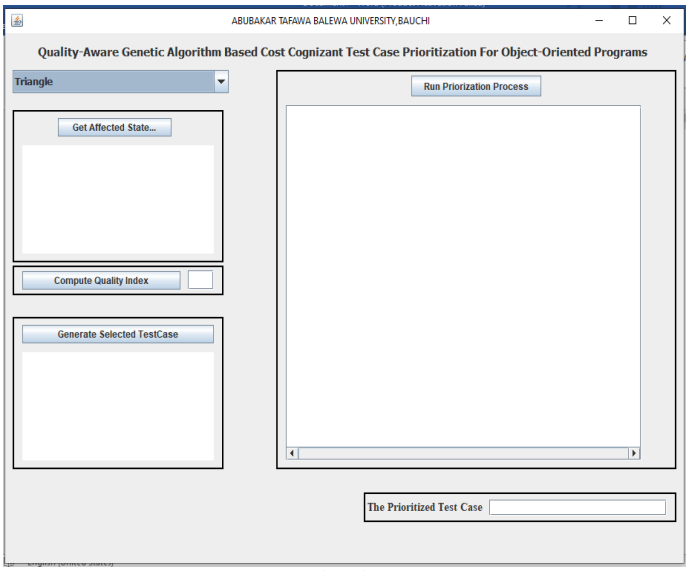


Figure 2: Main Interface for QAG-TCP Tool

The QAG-TCP main interface contains all the necessary commands and text areas needed for the user to work with. Each command and its corresponding result display area are grouped in a single panel. The first panel includes a dropdown menu for selecting a program object. The remaining steps for the execution of the QAG-TCP tool are described below:

1. The process of prioritizing test cases commences when the user inputs the affected statements by clicking the Get Affected Statement Action Button in the first panel. The modified model generates the affected statements by employing the µJava tool's mutants. The affected nodes and statements are stored in a text file affectedStatement.txt and later accessed in the tool for the remaining operations. The affected Statements are equally displayed in a Text Area as shown in Figure 2
2. The next step is to click on the Compute Quality Index in the second panel of the main interface of the tool, this will involve picking the source code of the program, producing the OOP metrics of the program, and performing some logical operation to produce and index that will have exponential effect on the logical procedure that produces the prioritized test case. The Quality Index will be stored and displayed in the Text Field as shown in Figure 3 below
3. The third panel contains a command that generates selected test cases when the action button is clicked. The text area below the button displays the selected test cases along with the affected statements they cover.
4. The fourth panel contains a command to retrieve prioritized test cases and a text area displaying their fitness score. Figure 3 depicts a screenshot of the main interface prototype.

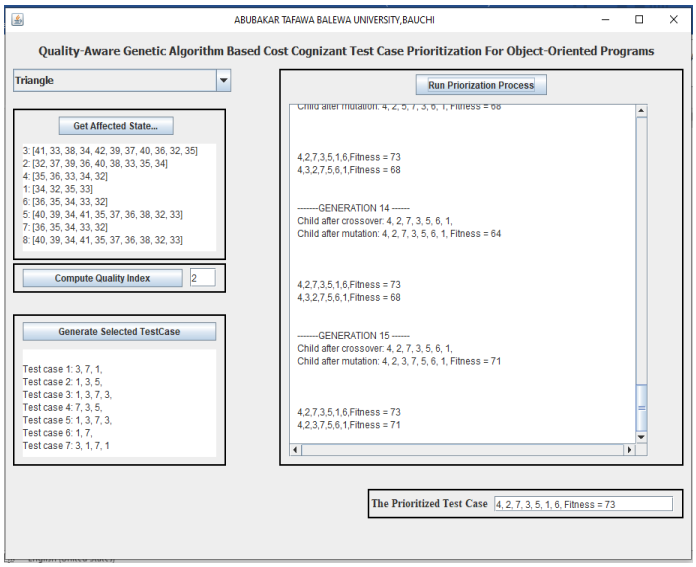


Figure 3: Prioritization output Displayed

5. Results and Discussion

Results

This section presents the empirical results obtained from the experimental evaluation of the Quality-aware GA-based cost-cognizant test case prioritization (QAG-TCP) technique. The analysis focuses on QAG-TCP's performance across three key metrics: fault detection effectiveness (EFFpa), testing effort efficiency (TEEpa), and the cost-cognizant fault detection rate (APFDc). A comprehensive comparative analysis of QAG-TCP with other existing techniques will be presented in a separate, dedicated publication.

Fault Detection Effectiveness (EFFpa): QAG-TCP demonstrated high fault detection effectiveness across the object-oriented programs used in the evaluation. As illustrated by its average effectiveness score of 75.11%, its prioritization strategy consistently showed strong capabilities in identifying faults. Statistical analysis further confirmed a significant performance level in fault detection achieved by QAG-TCP.

Testing Effort Efficiency (TEEpa): In terms of testing effort efficiency, QAG-TCP exhibited strong performance. While its average execution time was 2.34 seconds, the technique achieved high efficiency in locating faults relative to the effort required, as depicted by its overall efficiency scores (Figure 5). Statistical analysis indicated a highly significant efficiency level for QAG-TCP, underscoring its optimized resource utilization.

APFDc Performance: QAG-TCP generally yielded beneficial trends in APFDc scores, with its average APFDc performance of 94.36%. This indicates its strength in prioritizing test cases in a cost-cognizant manner, aiming to detect faults earlier in the testing process while considering associated costs.

Illustrative Example: QAG-TCP Prioritization Process

An illustrative example will be provided in this section showing all the computational process in each stage of the technique

Statements in the Program

The statements in the source code were picked and numbered sequentially, the result was used to generate a dynamic matrix. The choice of a dynamic matrix is because the number of statements is assumed to be dynamic since each program has its number of codes and can change during updates/modifications. The matrix is shown below

[1, 2, 3, 4, 5, 6]
[11, 12, 13, 14, 15, 16]
[21, 22, 23, 24, 25, 26]
[31, 32, 33, 34, 35, 36]
[41, 42, 43, 44, 45, 46]
[51, 52, 53, 54, 55, 56]
[61, 62, 63, 64, 65, 66]
[71, 72, 73, 74, 75]

Test Case Selection

Test cases that execute at least one affected statement are chosen by a test case selector based on coverage data and affected statements. Next, a numerical integer representing a subset of the test cases is encoded and used as the input for the prioritizing component.

3 = [10, 8, 12, 11, 9]
2 = [14, 12, 9, 11, 13, 8, 15, 10]
4 = [17, 8, 18, 12, 13, 14, 16, 10, 11, 15, 9]
1 = [8, 9, 10, 11]
6 = [13, 11, 10, 9, 8, 12, 15, 14]
5 = [12, 14, 13, 8, 10, 11, 9, 15]
7 = [13, 11, 10, 9, 8, 12, 15, 14]
8 = [8, 6, 4, 5, 10, 7, 9, 11]

Criticality Calculations

Formula 1 in Section 3 was used to derive the following table

Table 1: Criticality Derivation Table

TestCase	No_of_ Statements	Assigned Cost	TestCase Criticality
1	4	13	4
2	8	12	14
3	5	11	7
4	11	18	20
5	8	13	13
6	8	17	12
7	8	11	16
			82

Therefore, the Criticality Sum is 82

Quality Index Calculation

Using the formula 2 or alternatively formula 3 in section 3 the following was derived

Table 2: Quality Index

CBO	LCOM	Quality Index
1.0319	2.5021	2

Qfitness Function Calculation

Tables 1 and 2 will now be used to compute QFitness Value. Assuming we take Test Case 3 below.
As shown above in table 1 , the criticality of test case 3 is 7, its order is 3 from Table 3 and tfi is 12 from Table 3. Using the formula 4 in section 3 we calculate the QFitness of each Test Case

$$T3_{(Qfitness)} = Order * tfi * Qi$$

$$T3_{(Qfitness)} = 3 * 7 * 2$$

$$T3_{(Qfitness)} = 42$$

The above calculation is done for all the test cases and the results are shown in the table below

Table 3: QFitness Results

TestCase	Order	TestCase Criticality	Quality Index	Qfitness
6	1	12	2	24
5	2	13	2	52
3	3	7	2	42
2	4	14	2	112
7	5	16	2	160
4	6	20	2	240
1	7	4	2	56
				686

Test Case prioritization

The higher the QFitness value the higher the priority of the test case, hence from the Table 3 the prioritized test cases are as shown below

Table 4: Final Order(prioritized test cases)

t4	t7	t2	t1	t5	t3	t6
4	7	2	1	5	3	6

Table 4 shows t_4, t_7, t_2, t_1, t_5, t_3, t_6 is the order by which faults need to be trap efficiently. The process will be repeated while performing exchanging parent with a child until the QFitness value remain the same before and after mutation. That final order is the prioritized test cases.

Discussion

The empirical evaluation clearly highlights the robust performance of the proposed QAG-TCP technique, particularly its notable achievements in fault detection effectiveness and testing effort efficiency. These strengths are directly attributable to QAG-TCP's innovative methodology. QAG-TCP's enhanced fault detection is rooted in its data-driven framework. This framework incorporates real-world fault data and leverages object-oriented code quality metrics. By precisely targeting potentially problematic code modules identified through these metrics, QAG-TCP effectively and proactively uncovers faults. This structured approach contributes significantly to its high effectiveness. Similarly, QAG-TCP's superior testing effort efficiency showcases the method's capability to optimize resource utilization. The inherent adaptive nature of its genetic algorithm allows it to intelligently explore the test case prioritization space, maximizing fault revelation while optimizing the effort expended. This balance makes QAG-TCP a highly practical solution for regression testing. While the APFDc results showed positive trends for QAG-TCP, which will later be proved through demonstrating its statistical significance to fully demonstrate its statistical advantage.

6. Conclusion and Future Scope

QAG-TCP presents a promising approach to object-oriented regression testing prioritization. Its incorporation of real-world fault data, cost considerations, and complexity measurement yields a more comprehensive and effective prioritization strategy. The empirical evaluation confirmed QAG-TCP's significant improvements in fault detection effectiveness, achieving an average score of 75.11%. The study's findings also consistently demonstrated QAG-TCP's high test effort efficiency, with an average execution time of 2.34 seconds. This highlights its ability to detect faults earlier and, overall, detect more faults while optimizing effort. While QAG-TCP exhibited beneficial trends in its APFDc scores 94.36%, the full implications of this aspect will be further explored in subsequent work. This approach distinguishes itself from traditional techniques by utilizing real-world fault data and complexity measurement, establishing a more realistic and effective prioritization process. Further research will explore its application to diverse software domains and investigate additional quality attributes for inclusion in the fitness function to enhance its capabilities.

Author's Statement

Disclosure: The authors declare that the research presented in this paper, titled "Quality-Aware Genetic Algorithm Based Cost Cognizant Test Case Prioritization for Object-Oriented Programs," was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. The authors declare that no specific funding was received for this research from any public, commercial, or not-for-profit organizations. This research was self-funded by the authors. The authors declare no competing interests. All authors have read and agreed to the published version of the manuscript.

Acknowledgements

The authors extend sincere gratitude to the anonymous reviewers whose insightful comments and constructive feedback significantly enhanced the quality and clarity of this manuscript. Their valuable suggestions were instrumental in shaping the final version of this work. Furthermore, the authors wishes to express deep appreciation to Professor A.Y Gital for their invaluable guidance and unwavering support throughout this research

Funding Source
None.

Authors' Contribution
The co-author(Hassan Abubakar) was responsible for the conceptualization, methodology, software development, validation, formal analysis, investigation, resource acquisition, initial draft writing, and visualization of this study. All these contributions were carried out under the direct guidance and supervision of the senior author who is also the second author.

Conflict of Interest
All authors declare that they have no financial, commercial, legal, or professional relationships with other organizations or individuals that could potentially influence the objectivity or integrity of this research. No other author has received consultancies, employment, or support from advocacy groups, fees, honoraria, patents, royalties, or stock/share ownership that could be perceived as creating a conflict of interest.

Data Availability
The Source codes used as input datasets in the current study are publicly accessible for in *https://sir.csc.ncsu.edu/portal/index.php* and are expected to be maintained permanently within the repository in accordance with their archival policies. The custom code developed for this study will be made available in the *https://github.com/HAGummi/QAG_TCP*. The repository ensures the long-term availability of the code through its persistent identifier and archival practices.

References

[1] R. S. Pressman and B. R. Maxim, "Software Engineering: A Practitioner's Approach," *McGraw-{Hill}. {Safe} {Work} {Australia}, {EMERGENCY} {PLANS} {FACT} {SHEET}. http://bit.ly/1nhk52x {Accesed} on {Jan}*, Vol.**19, 2017.**

[2] S. Saroja and S. Haseena, "Functional and Non-Functional Requirements in Agile Software Development," *Agil. Softw. Dev. Trends, Challenges Appl.*, pp.**71–86, 2023.**

[3] F. F. Xia, "GIS Software Product Development Challenges in the Era of Cloud Computing," in *New Thinking in GIScience*, Springer, pp.**129–142, 2022.**

[4] H. Abubakar, F. U. Zambuk, U. M. Ahmed, and A. Y. Gital, "A Review on the New Trend in Regression Test Case Prioritization," *ATBU J. Sci. Technol. Educ.*, Vol.**11**, No.**1**, pp.**426–436, 2023.**

[5] M. Heusser and M. Larsen, *Software Testing Strategies: A testing guide for the 2020s*. Packt Publishing Ltd, **2023**.

[6] D. Paterson, "Improvements to Test Case Prioritisation considering Efficiency and Effectiveness on Real Faults," no. March, **2019**.

[7] A. Bello, "EVOLUTIONARY COST-COGNIZANT REGRESSION TEST CASE PRIORITIZATION FOR OBJECT-ORIENTED PROGRAMS," **2019**.

[8] A. Bello, A. B. Md. Sultan, and S. Shehu, "Multi-Criteria Evolutionary Regression Test Prioritization for Dynamic Object-Oriented Programs," *Int. J. Adv. Electron. Comput. Sci.*, Vol.**6**, No.**1**, pp.**14–18, 2019.**

[9] I. P. Fernandes and L. E. G. Martins, "Test case prioritization methods: A systematic literature review," *J. Softw. Eng. Res. Dev.*, Vol.**13**, No.**2**, pp.**13–51, 2025.**

[10] D. Manikkannan and S. Babu, "Test Case Prioritization via Embedded Autoencoder Model for Software Quality Assurance," *IETE J. Res.*, Vol.**70**, No.**4**, pp.**3845–3855, 2024.**

[11] Y. Zhu and F. Liu, "Test Case Prioritization Algorithm Based on Improved Code Coverage.," *IAENG Int. J. Comput. Sci.*, Vol.**50**, No.**2, 2023.**

[12] F. S. Ahmed, A. Majeed, T. A. Khan, and S. N. Bhatti, "Value-based cost-cognizant test case prioritization for regression testing," *PLoS One*, Vol.**17**, No.**5**, pp.**e0264972, 2022.**

[13] L. Raamesh, S. Radhika, and S. Jothi, "A cost-effective test case selection and prioritization using hybrid battle royale-based remora optimization," *Neural Comput. Appl.*, Vol.**34**, No.**24**, pp.**22435–22447, 2022.**

[14] S. Yadav, D.K., Dutta, "Regression test case selection and prioritization for object oriented software," *Microsyst Technol*, Vol.**26**, pp.**1463–1477, 2020.**

[15] A. Bello, A. Sultan, A. A. Abdul Ghani, and H. Zulzalil, "Evolutionary Cost Cognizant Regression Test Prioritization for Object-Oriented Programs Based on Fault Dependency," *Int. J. Eng. Technol.*, Vol.**7**, No.**4.1**, pp.**28**–**32**, **2018**.

[16] R. M. Parizi, "Automatic randomized test generation technique for aspectoriented software (Doctoral dissertation, Universiti Putra Malaysia)," **2012**.

AUTHORS PROFILE

Dr. Hassan Abubakar received his PhD in Computer Science from Abubakar Tafawa Balewa University, Bauchi State, Nigeria. He earned his Master's Degree in Computing: Information Engineering with Network Management from Robert Gordon University, Aberdeen, UK. He also holds a B.Sc. in Computer Science from Usmanu Danfodiyo University, Sokoto, Nigeria. He is currently a Lecturer at the Department of Computer Science, Usmanu Danfodiyo University, Sokoto, Nigeria. His research interests include software engineering, programming, web design, and database management. He has 21 years of teaching experience and 8 years of research experience.

Dr. Fatima Umar Zambuk received her PhD in Computer Science from Abubakar Tafawa Balewa University (ATBU), Bauchi State, Nigeria. She is currently a lecturer at the Department of Computer Science, Abubakar Tafawa Balewa University, Bauchi, Nigeria. Her research interests include cloud computing, task scheduling algorithms, optimization techniques, and machine learning, with a particular focus on energy efficiency and resource management in virtualized environments.

