Original Research Paper

Greedy Algorithm Implementation for Test Case Prioritization in the Regression Testing Phase

Akira Wahyu Putra and Nilo Legowo

Department of Information System Management, BINUS Graduate Program Master of Information System Management, Bina Nusantara University, Jakarta, Indonesia

Article history
Received: 10-09-2024
Revised: 11-11-2024
Accepted: 18-11-2024

Corresponding Author:
Akira Wahyu Putra
Department of Information System Management, BINUS Graduate Program Master of Information System Management, Bina Nusantara University, Jakarta, Indonesia
Email: akira.putra@binus.ac.id

Abstract: This study examines the implementation of Test Case Prioritization (TCP) using the Greedy Algorithm (GA) to enhance regression testing efficiency within a financial technology company's software development cycle. With testing durations increasing significantly, this study aims to address inefficiencies by applying the Greedy Algorithm to optimize test suite size and fault detection. The research methodology involves applying the Greedy Algorithm during the Regression Testing phase, comparing the prioritized suite with the original suite using metrics such as Average Percentage Fault Detection (APFD) and Test Suite Size Reduction (TSSR). Results show that the Greedy Algorithm achieved substantial improvements in both test suite size and fault detection effectiveness across different projects. For Project A, the test suite was reduced from 51-22 test cases, achieving a TSSR of 56.8%, with an APFD increase of 206.21%, rising from 0.0853-0.2613. Project B demonstrated even greater optimization, reducing the test suite from 36-8 test cases, resulting in a TSSR of 77.8% and an APFD improvement of 83.92%, rising from 0.3194-0.5875. These outcomes underscore the algorithm's effectiveness in eliminating redundant test cases, accelerating testing, and enhancing fault detection thereby supporting the company's goal of faster release cycles without compromising quality.

Keywords: Regression Testing, Greedy Algorithm, Test Case Prioritization, APFD, Software Development

Introduction

Information Systems (IS) and the Software Development Life Cycle (SDLC) are closely linked, working together to deliver effective software solutions. Within the SDLC, Software Testing (ST), especially Regression Testing (RT), plays a crucial role in ensuring software quality. However, as software evolves, challenges like longer testing times arise, affecting efficiency and early bug detection.

Testing guarantees that software updates work properly and that a product satisfies quality criteria. Software has to be current in the highly competitive marketplace of today. Although using intuition to test relies on personal ability, comprehensive testing by a dedicated team is better. Improper testing causes problems to go unnoticed, which raises expenses and requires more labor to resolve. Problems are more expensive to find after software is deployed than they are in the early phases of planning.

In this research, the company, which operates in the financial technology industry, currently executes automated testing using Web driver IO for the backend is implemented. However, with each update, the duration of testing increases, impacting bug detection and development time. To address this, Test Case Prioritization (TCP), particularly using the Greedy Algorithm (GA), is suggested to optimize testing processes.

Regression testing during development in the company takes 2 days per week, with one day for testing and one for bug fixing. This amounts to 8 h of testing per week, aligned with the company's working hours. However, because additional test cases are added with every system update, as illustrated in Fig. (1), the time required for automated testing has almost doubled to 6 h by the fifth release. Testing efficiency and early bug detection are challenged by this extended testing period. Therefore, it's essential to optimize the testing procedure, particularly when the quantity of test cases increases in subsequent releases.

Science Publications

© 2025 Akira Wahyu Putra and Nilo Legowo. This open-access article is distributed under a Creative Commons Attribution (CC-BY) 4.0 license.

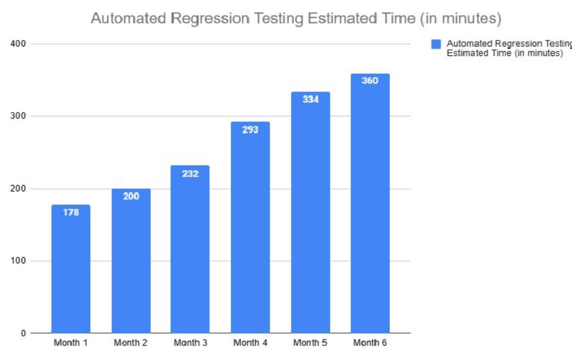

Fig. 1: Automated regression testing Time

Figure (2) illustrates the estimated time for manual regression testing derived from the time estimation for a single test case in minutes. In the current condition of the company, manual regression testing is still conducted due to scenarios that cannot be addressed through automated testing. The average time required to complete manual regression testing, based on the last 5 releases, exceeds 6 h.

As the system requirements evolved due to system improvement and the increasing number of automated test scripts, significant effort and time were required to maintain and update the code to align with the changes in system requirements. These factors can slow down development and hinder early bug detection during regression testing. This could jeopardize the company's goal of increasing team productivity by 15% and adding an optional additional release each week, bringing the total to two releases per week and reducing the regression testing time to only just 1 day and QA Engineers need to maintain automation script due to changes in requirements. With one release already consuming nearly two days of work hours, there's a pressing need for research to ensure efficient execution of automated testing in line with system requirements.

To tackle this issue, numerous techniques for TCP have been introduced in existing literature. Among the various Test Case Prioritization (TCP) techniques, the Greedy Algorithm (GA) has acquired significant consideration since its proposal in 1999, mainly due to its commonly acknowledged effectiveness.

The Greedy Algorithm (GA) has been recognized for its effectiveness in TCP and this study aims to explore its application to enhance regression testing efficiency and effectiveness. Based on the provided problem statement, here are two problem formulations:

1. RQ1: How can regression testing be effectively conducted through the implementation of Test Case Prioritization?
2. RQ2: What methodologies will be used to compare the original test suite with the prioritized suite optimized by the Greedy Algorithm?

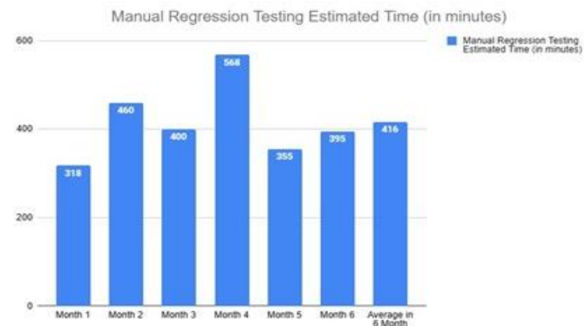

Fig. 2: Automated regression testing time

This study aims to achieve two primary objectives in enhancing the company's regression testing process. The first objective is to implement the Greedy Algorithm for test case prioritization in ongoing projects, specifically Projects within the company. The research seeks to streamline the testing process, reduce redundancy, enhance time efficiency, and improve early fault detection. The second objective involves a comparative evaluation between the original and optimized test suites. The research will assess the effectiveness of the optimized test suite, prioritized using the Greedy Algorithm, by comparing it to the original suite in terms of time efficiency and overall software quality. This comparison will provide valuable insights into improving the regression testing process at the company.

Theoretical Background

Software Development Life Cycle

As a project roadmap, providing a flexible framework to meet the software development's goals is the main feature of the Software Development Life Cycle (SDLC). It includes stages like defining requirements, designing, developing, and testing the software. In implementing software development practices, effort and resources play an essential role, especially in the regression testing phase (Hettiarachchi *et al*., 2016)

Regression Testing

Modifying and maintaining software often involves a crucial activity known as regression testing. This maintenance process, while incurring costs, is defined as essential within the Software Development Life Cycle (SDLC). Regression testing is conducted to revalidate and provide confidence in the modified software, ensuring that alterations have not negatively impacted the software's behavior (Yoo and Harman, 2012).

Regression testing involves rerunning previously executed tests to ensure that recent software changes have not introduced new issues. This is a critical process in the software development lifecycle, but it often consumes significant testing resources. As software evolves, test

suites tend to grow in size, making it impractical to execute every test case. This challenge is amplified by shorter release cycles, which heighten the importance of efficient regression testing (Greca *et al*., 2023).

Ansari *et al*. (2016) outlined several regression testing techniques in their study. (Ansari *et al*., 2016) The various methods for regression testing are detailed in Fig. (3).

During regression testing, the addition of any code to the application necessitates extensive retesting, resulting in significant time consumption (Qiu *et al*., 2015). Existing test cases may fail due to changes in the system's behavior, which may reflect the modifications rather than indicate issues. To ensure relevance, all test cases should be reviewed before being reused to test new versions of the system (Di Nardo *et al*., 2015).

The first step in the regression testing phase is to identify the modified source code or source features. During maintenance, developers modify related source code to add new features or to fix faults. This can be done by manually reviewing the code. Once the changed source code or source features have been modified, the tester needs to identify the test cases that cover those changes. The regression testing phase can be illustrated in Fig. (4) (Singh *et al*., 2016).

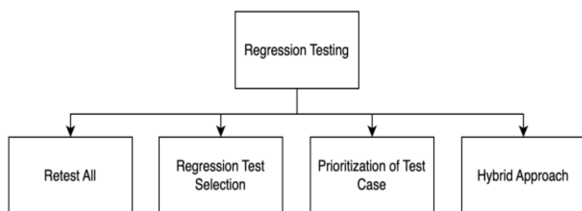

Fig. 3: Regression testing techniques (Ansari *et al*., 2016)

Regression testing can be done manually or through automation. Automation is faster and reduces the risk of human error. Implementing automation testing is particularly beneficial for efficient regression testing with automation. It cuts down on test execution time, making the process more efficient (Sutapa *et al*., 2020).

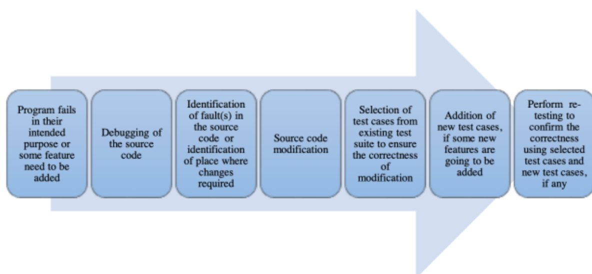

Fig. 4: Regression testing phase (Singh *et al*., 2016)

Test Case Prioritization

Not all test cases covering changed source code or features need retesting. Testers should focus on cases likely to uncover new bugs related to feature requirements. Regression testing introduced Test Case Prioritization to strike a balance between test objectives and real-world limitations. This involves strategically scheduling test case execution (Lou *et al*., 2019). After selecting test cases, they're added to the regression testing suite. Retesting is the final phase, where chosen cases are executed to verify software functionality after changes (Singh *et al*., 2016).

Test-suite prioritization techniques are implemented to reduce the costs linked to storing and reusing test cases in software maintenance. This is achieved by eliminating duplicate test cases from test suites. The primary goal is to optimize the testing process, making it more efficient and cost-effective. By prioritizing test cases based on specific criteria, these techniques contribute to a streamlined approach, ensuring that testing efforts are focused on critical aspects of software functionality and changes during maintenance activities (Singh *et al*., 2016). Prioritizing test cases involves arranging test cases in an optimal sequence to enhance critical coverage properties, such as detecting faults early in the testing process (Yoo and Harman, 2012).

Apart from the fact that effective implementation of Test Case Prioritization (TCP) positively impacts testing duration and saves resources, this implementation could also lead to gaining stakeholder assurance. In this context, different methods and techniques have been utilized to attain the most optimal test suite specifically for Regression Testing (Qasim *et al*., 2021). The summary of Test Case Prioritization is illustrated in Fig. (5).

Requirement-based methods are ranked as the fourth most popular strategy out of the numerous ways shown in Fig. (5). Based on its requirements, a system is built. As a result, using requirements information may improve the identification of key test cases in addition to what can be accomplished using code-related data alone. In the requirement-based approach, test cases are prioritized and generated by requirement gathering.

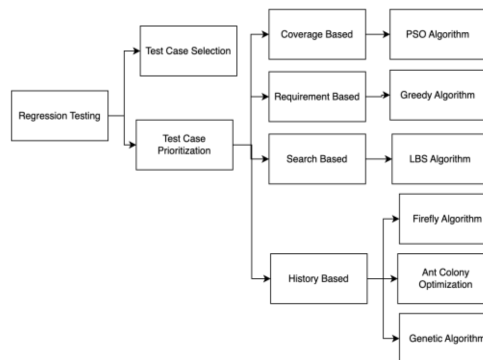

Fig. 5: Test case prioritization techniques (Qasim *et al*., 2021)

In comparing thirty studies on regression testing approaches, Qiu *et al.* (2014) found that, of the total studies reviewed, the majority (56.7%, or 17 out of 30) used test case prioritizing techniques to optimize the order in which test cases were executed with the goal of increasing fault detection rates. Ten studies also used the same method to reduce the total number of test cases required to comprehensively cover all modified areas. Only two research, nevertheless, used test suite minimization strategies (Qiu *et al.*, 2015).

Greedy Algorithm (GA)

Early efforts to improve test case efficiency in regression testing primarily focused on reducing redundant test cases, which contributed to increased testing costs. A method was proposed to minimize the overall test suite size by selecting only the most essential test cases— those that are critical to covering all specified requirements. This approach reduces redundancy while ensuring that all necessary functionality is tested. Essential test cases are those that cannot be removed without compromising the test suite's ability to meet all requirements. Identifying redundant test cases involves a pairwise comparison with essential cases, followed by the application of a Greedy Algorithm to optimize the selection process. This method aims to streamline regression testing by balancing coverage needs with resource efficiency (Jehan and Wotawa, 2023).

The Greedy Algorithm is used in test case prioritization because of a number of advantages. First of all, because of its effectiveness and simplicity, it's a great option for test case prioritizing, especially when dealing with large and complex test suites. The iterative selection methodology helps to eliminate pointless test cases and minimize the total size of the test suite in order to optimize testing resources. If the testers' main goal is to decrease the time required to perform the test suite reduction technique, the Greedy Algorithm is the ideal choice (Lin *et al.*, 2017).

Table 1: Example of requirement coverage data from a test suite's test cases. (Gladston *et al.*, 2016)

	REQ1	REQ2	REQ3	REQ4	REQ5
TS1	✓			✓	
TS2	✓	✓	✓		
TS3		✓	✓		
TS4	✓	✓			
TS5	✓		✓	✓	
TS6	✓				✓

Where: TS: Test case that is defined in a test suite; REQ: Requirements that related to the test suite and system’s feature

The goal of a Greedy Algorithm search is to reduce the estimated cost of reaching a particular objective. While it is a straightforward approach, it becomes attractive in situations where it delivers high-quality results because it is generally cost-effective in terms of both implementation and execution time (Li *et al.*, 2007).

Gladston (2016) defines the greedy algorithm utilized and the steps are as follows as illustrated in Table (1) (Gladston *et al.*, 2016):

1. Select essential test cases.
2. Start by choosing all essential test cases, which typically represent the core functionality or critical aspects of the system under test.
3. Remove redundant test cases.
4. Next, redundant test cases that provide overlapping coverage or contribute minimally to overall test coverage should be eliminated.
5. Address Uncovered Requirements After removing redundant cases, check for any remaining uncovered requirements. If any exist, select additional test cases that fulfill most of these requirements to ensure comprehensive coverage.

Previous Research

In a study conducted by Jehan and Wotawa (2023), the greedy test suite minimization techniques coverage was explored, with a comparison between the greedy algorithm and the delayed greedy algorithm. The findings reveal that the Greedy Algorithm achieved an 87.4% reduction in test suite size, whereas the Delayed Greedy Algorithm reduced it by 74.2%. Additionally, the Greedy Algorithm demonstrated a faster test suite minimization compared to the delayed greedy algorithm (Jehan and Wotawa, 2023).

Khatibsyarbini *et al.* (2018) conducted research on implementing test case prioritization techniques in software testing. The study aimed to compare several prioritization methods, evaluating their effectiveness using Average Percentage Fault Detection (APFD) and execution time. The results indicated that the Greedy Algorithm produced APFD values that were very similar, though slightly lower, compared to both Particle Swarm Optimization (PSO) and Genetic Algorithm (GA). Additionally, the Greedy Algorithm demonstrated significant efficiency improvements, reducing execution time by nearly half compared to PSO and approximately one-ninth compared to GA. These findings suggest that the Greedy Algorithm offers a competitive approach in terms of both fault detection effectiveness and execution efficiency (Khatibsyarbini *et al.*, 2018).

Yamuç *et al.* (2017) conducted research to reduce test suites comprising test cases and test requirements. Tests were conducted on the dataset to compare the Greedy Algorithm with the Genetic Algorithm (GA). The

Greedy Algorithm yielded a minimum execution time of 193 sec, while the GA completed the tests in 153 sec. (Yamuç *et al*., 2017).

Alian *et al*. (2016) collect and examine papers focused on regression testing techniques, specifically those related to test suite reduction. The assessed techniques for reducing test cases are categorized into Greedy Algorithm, hybrid algorithm, requirement-based, coverage-based, clustering, genetic algorithm, fuzzy logic, and slicing approaches. Techniques based on the Greedy Algorithm offer noteworthy reductions in the number of test cases (Alian *et al*., 2016).

In their 2016 research, Singh *et al*. (2016) conducted a comparative analysis of various test suite minimization techniques. Among the techniques considered, the Greedy Algorithm was included for evaluation. The findings revealed that the Greedy Algorithm, as one of the evaluated techniques, achieved a notable TSSR ranging between 41.67 and 50%. This outcome underscores the efficacy of the Greedy Algorithm in reducing the size of test suites, showcasing its potential as a valuable approach in test suite minimization strategies, as demonstrated in Singh and Shree's research.

In Srivastava's (2008) research, the application of the Greedy Algorithm in reducing effort and prioritizing test cases during regression testing was explored. The study involved an analysis that distinguished between prioritized and non-prioritized test cases. By utilizing the Average Percentage of Faults Detected (APFD) method, Srivastava evaluated the effectiveness of the two test case categories. During the regression testing phase, the result of prioritized test cases was more efficient, surpassing the non-prioritized test suite with an impressive 81% effectiveness rate. This research highlights the significance of the Greedy Algorithm in optimizing the execution of test cases, particularly in the context of regression testing (Srivastava, 2008).

In summary, the collective evidence from the research studies strongly supports the effectiveness of the Greedy Algorithm in test case prioritization and suite minimization. The Greedy Algorithm consistently demonstrated significant reductions in test suite size.

Test Suite Size Reduction (TSSR)

Test Suite Size Reduction (TSSR) serves as a quantitative statistic that indicates how much of the total size of the test suite can be reduced using a certain approach. This rate percentage offers a quantifiable indication of the test suite's optimization and streamlining efforts, in addition to reflecting the effectiveness of the selected methodology. In the context of software testing and quality assurance procedures, TSSR provides useful insights into the effectiveness and influence of the used method on improving the manageability and efficacy of the test suite prioritization process by quantifying the decrease (Wong *et al*., 1995):

$$TSSR = \frac{TS_{orig} - TS_{red}}{TS_{orig}} \times 100\% \quad (1)$$

where:
TS_{orig} = Total number of the original test suite
TS_{red} = Total number of the reduced test suite

AFPD (Average Percentage Fault Detection)

APFD is an approach for measuring the percentage ratio of bugs detected in each test execution suite and is an evaluation method for the fault detection rate (Elbaum *et al*., 2004) This measure functions as a weighted average indicator of detected bugs problems and is used as a benchmark for assessing how well test cases are prioritized. A value of APFD close to 1 indicates a good fault detection performance (Maspupah *et al*., 2023). AFPD provides insights into the fault detection capabilities within the refined test suite (Srivastava, 2008). This assessment offers numerical insights into how well the implemented greedy algorithm performs, illuminating its capacity to improve fault detection rates during the regression testing phase:

$$APFD = 1 - \left(TF_1 + TF_2 + TF_3 + \ldots + TF_m\right) / (nm) + \frac{1}{2n} \quad (2)$$

where:
TF = Fault from selected executed test suite
m = Counts of faults found
n = Total number of test cases

Materials and Methods

This section outlines the materials and methodologies employed in implementing the Greedy Algorithm for test case prioritization during the regression testing phase at PT XYZ.

Data Collection

The research was conducted on PT XYZ’s Core Transaction Payment System, a financial technology platform that processes large volumes of financial transactions daily. This system consists of multiple modules, including payment initiation, confirmation, and reconciliation, each requiring thorough testing due to its critical nature. The study examined two distinct projects within the system:

1. Project A: Focused on the partial payment feature. This project had 15 existing test cases that were reused for regression testing, making it ideal for evaluating the optimization of pre-existing suites
2. Project B: A newly developed feature with no prior test cases. This allowed for the creation and prioritization of a test suite from scratch

Testing Environment

The automation testing framework WebdriverIO was utilized to execute the test cases. WebdriverIO, a Node.js-based tool, facilitated automated testing for browser and API interactions. The framework was integrated into PT XYZ's Continuous Integration/Continuous Deployment (CI/CD) pipeline to streamline execution. The programming language JavaScript was employed for scripting. The testing infrastructure was hosted on local development environments, with capabilities to simulate real-world transactional scenarios.

Test Suite Metrics

Two primary metrics were used to evaluate the effectiveness of the prioritization approach:

1. Test Suite Size Reduction (TSSR): This metric assessed the efficiency of the optimization by quantifying the reduction in the number of test cases
2. Average Percentage Fault Detection (APFD): This metric measured the fault-detection effectiveness of the prioritized test suite. Higher APFD values indicated faster and more efficient detection of software faults

Methods

Figure (6) shows the research framework. Development is started when there is an issue within the system, or there is a need for improvement of certain features in the system. The programmer will analyze which code needs to be improved after the product team breaks down the requirements of the project.

Upon completion of the system development phase, regression testing will be conducted to ensure that recent modifications do not negatively impact existing functionalities. This process requires the careful selection of test cases most likely to uncover new defects. The selected test cases should be closely aligned with the requirements of the modified features to verify that these features function correctly in light of recent changes, while also safeguarding the integrity of the overall system functionality. This approach aims to enhance the detection of defects introduced by modifications and supports the continued reliability of both new and existing functionalities.

This research will focus on Steps 5 and 7 of the regression testing process as outlined by Singh *et al.* (2016) and will apply the Greedy Algorithm methodology as utilized by Gladston *et al.* (2016). Regression testing will be initiated when system bugs are identified, prompting developers to debug and locate faults or areas of the source code requiring modification. Once these steps are completed, the research will proceed, beginning with the collection of relevant test cases. These test cases will be grouped according to similar requirements linked to specific features, creating an organized framework to enhance the relevance and coverage of the testing process for each feature.

In the critical phase of this research, the Greedy Algorithm is implemented to generate a prioritized list of test cases. Each Test scenario (T) is mapped to a corresponding Requirement (R), ensuring that only the chosen test cases are selected for regression testing. This implementation involves two test suite scenarios: One that is implemented with the greedy algorithm for test case prioritization and another that remains as the original test suite. This approach allows a comparative analysis, highlighting the efficiency and effectiveness of the greedy algorithm in optimizing test case selection.

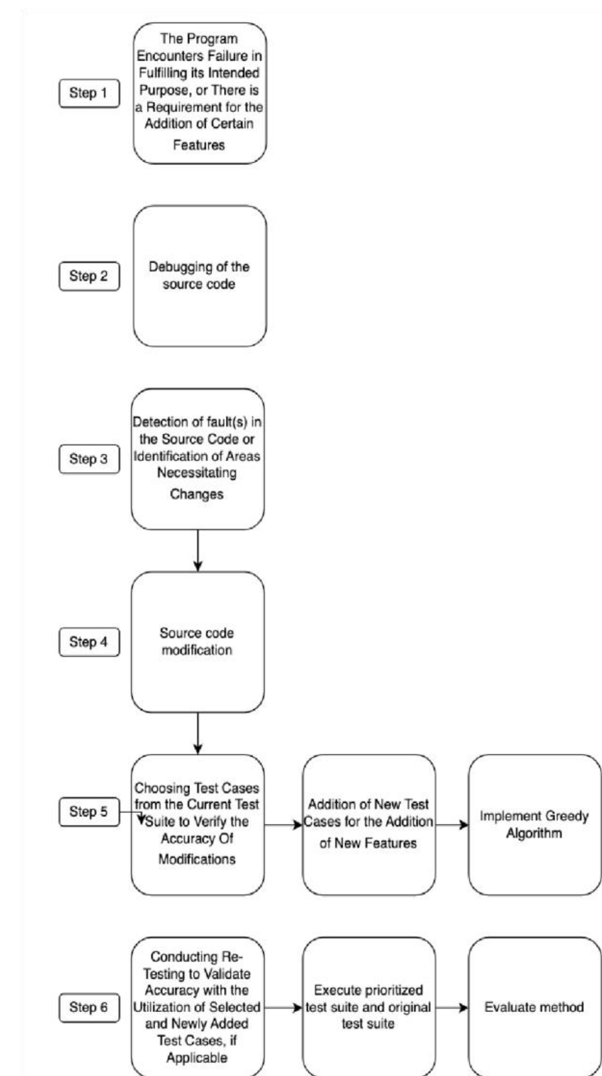

Fig. 6: Research framework

After prioritizing test cases and adding new test cases to address recent requirements and features, the process moves to the re-testing phase. In this phase, testers execute the prioritized test cases to ensure that the software operates correctly following the modifications. This step serves two critical functions: It verifies the stability of both updated and existing functionalities and identifies any defects that may have been introduced during development. Re-testing is vital for confirming that the modifications meet their intended objectives and that no unintended issues have disrupted the system, thereby maintaining overall reliability and functionality.

The research proceeds by executing both the prioritized and initial (unprioritized) test suites, following the prioritization phase. This stage includes a comparative analysis to highlight the differences between the prioritized test suite, arranged through the Greedy Algorithm, and the unprioritized test suite. The tests are conducted locally on a laptop using the Visual Studio Code terminal, with both expected and actual outcomes documented in cases of test failure.

The methodology for evaluating prioritized test cases encompasses two main metrics. First, the Test Suite Size Reduction (TSSR) is calculated to quantify the reduction in test cases achieved by prioritization with the Greedy Algorithm. TSSR compares the count of prioritized test cases against the original suite, yielding a percentage that reflects the reduction rate, thereby demonstrating the efficiency benefits of the proposed approach. This measure assesses the extent of test suite minimization while preserving coverage quality.

Following this, the original and prioritized test suites undergo separate testing to measure bug detection rates in each. Upon completion, the fault detection effectiveness is assessed through the Average Percentage Fault Detection (APFD) metric, which calculates the rate and extent of bug discovery across test executions. The APFD results for both the original and prioritized suites are compared to evaluate the efficiency of fault detection achieved through prioritization. This comparison underscores the impact of test case prioritization on regression testing efficiency, thereby affirming the effectiveness of the Greedy Algorithm in optimizing test execution and fault detection.

Results

In this research, the development team at the company is responsible for improving the current system by implementing two new features: Project A and Project B. Following the finalization of backlog grooming and sprint planning, the development phase begins. During this phase, the lead engineer assigns user stories to engineers, enabling them to start their tasks without delay.

During this phase, engineers analyze the code changes, identify impacted code segments, and make necessary modifications. Engineers also conduct self-testing to ensure that the developed code aligns with the specified requirements. Upon completing all tasks, engineers initiate a pull request for the code to be promptly tested by QA Engineers.

During the development phase, testing is initially conducted manually, alongside the creation of automation scripts primarily focused on API testing. While efforts to develop automation testing for UI and UI flow cases are underway, they are currently limited to manual tests. Once the development phase is completed, testing progresses to regression testing and User Acceptance Testing (UAT). Automation tests play a crucial role in reducing manual testing efforts during the regression testing phase in the staging environment. By automating repetitive test cases and ensuring comprehensive coverage, automation testing enhances efficiency, accelerates the testing process, and facilitates the timely identification of any regressions or discrepancies.

Currently, the software tester team has generated a repository of at least more than 1200 automated test cases. These test cases span across a diverse range of features and requirements that relate to the company's operations. Developed by individual teams, each test case specifically targets specific feature scopes, ensuring full test coverage across the system. Moreover, the QA team has engineered an automation test framework that is made for flexibility. This framework contains component files offering consumable functions, database components, and reusable data, assisting the whole test automation creation and testing process. By adopting this strategic approach, the company not only promotes efficiency in test case development but also facilitates cleaner and more maintainable code, ultimately enhancing the robustness of its testing procedures.

Test Case Creation

Test cases that are relevant to the ongoing development are consistently created by the QA Team and kept in a well-organized repository. Unfortunately, the majority of test cases that are currently in use are no longer relevant. This is due to the inherent flexibility of software development, whereby newly specified features and requirements overrule and substitute previously established requirements and test cases. As a result of later improvements and revisions, a large number of the test cases that were initially designed to verify particular functionalities have been classified as obsolete.

Despite the majority of test cases becoming obsolete, there remains a subset that retains relevance and validity in the context of the ongoing development efforts. These are the select few test cases that have managed to be relevant after a few iterations and enhancements, still applicable as reliable indicators of system behavior and performance. Its relevance within the software

development phase reflects its significance and underscores the importance of evaluating and reusing existing test cases where feasible.

QA engineers will develop new test cases in response to the product team's requirements in order to meet the software's changing needs. The purpose of these newly created test cases is to verify the most recent features and functionalities, assuring that the program fulfills the standards set out in the requirements. Using the requirements that were acquired during the backlog grooming process, the writer will create test cases.

It's important to note that only test cases that have not yet covered predetermined requirements will be created. Additionally, test cases will be crafted following a specific template as shown in Fig. (7), which includes:

1. Description: A brief explanation of the test case being created
2. Precondition: The stages or conditions that must be met before executing the test case
3. Scenario: The steps involved in the test case execution phase
4. Expected result: Determination of the anticipated outcome as a validation measure. If the test case deviates from the expected result, it will be marked as failed. Conversely, if the test case aligns with the expected result, it will be deemed as passed.

To further enhance test organization, test cases are categorized by type, distinguishing between UI Test Cases and API Test Cases. During execution, this distinction guides the testing approach, with UI Test Cases focusing on elements such as layout, buttons, and textual content, while API Test Cases emphasize backend validation. API testing ensures logical consistency, data integrity, and proper user access restrictions, making it critical for verifying the reliability of core functionalities.

UI testing, meanwhile, is limited to aspects like layout consistency, wording, and other visual elements. While UI testing is valuable for ensuring an intuitive user experience, the primary emphasis is on API functionality due to the current testing infrastructure, which is centered around API automation developed by the tester team.

Test Case Sample

The total number of API test cases developed by the QA team for the features of Project A and Project B is 51 and 36 test cases, respectively, only 15 existing test cases for Project A are being re-used as the rest of the test cases became obsolete. As for Project B, there are no existing test cases because it's a new feature to be developed by the development team. Therefore, testing for the Project B feature will be based on newly created test cases. After the test cases were created and reviewed with the product manager. After carefully weighing the needs and functionality of every feature, these numbers were determined.

Currently, the QA team primarily conducts automation testing from an API perspective, utilizing Web Driver IO as the test automation framework. Although the automated testing is executed locally, the execution report is integrated into the company's overall Test Report. This report not only captures the execution results but also documents the steps taken, the actual outcomes, and the final results of each test execution, as illustrated in Fig. (7). This process ensures transparency and traceability in the testing phase, contributing to the overall quality assurance efforts for the projects.

The automated test framework currently focuses on API automation testing, as the team primarily develops API-related features for payment functionalities. Since UI testing can be effectively conducted manually, the testing efforts are concentrated on validating the backend processes through automation. This strategic focus ensures that the critical aspects of the payment system are thoroughly tested, enhancing the overall reliability and performance of the developed features.

Requirement Mapping

Table (2) illustrates the requirement mapping for the Project A and B features for the development team, encompassing a total of 43 and 18 distinct requirements, respectively. These requirements serve as crucial guidelines for the forthcoming implementation of the greedy algorithm. Each requirement represents specific functionalities, constraints, and expectations, providing a comprehensive user journey for the development team. Project A focuses on enhancing the repayment method features for a designated loan, followed by the progression of the payment process until the status is successfully updated to "paid" status. Project B focuses on the loan management system. By following these requirements, the development team attempts to ensure the seamless integration of all features into their system, highlighting the must-have features for the developed software. By breaking down these requirements, the Development Team pinpoints the most important test cases that cover everything the system needs to do. This ensures the team gets all bases covered and that the current software behaves as expected in all situations. The chosen test cases will play a pivotal role in validating the approach's efficacy in meeting these requirements, including enhancing the overall user experience and system performance.

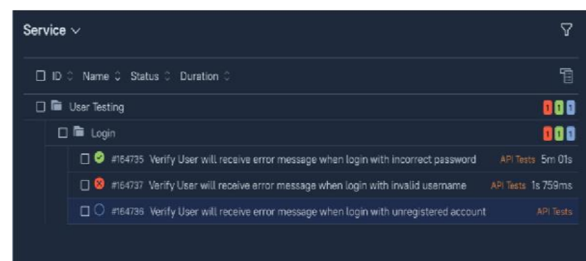

Fig. 7: Test execution launch example

Table 2: Total requirements of the system in each project enhancements

Project	Total requirement (s)
Project A	41
Project B	18

Greedy Algorithm Implementation

Once all requirements have been clearly defined, the next step is to implement the Greedy Algorithm to determine which requirements are addressed by the selected test cases. This method facilitates the creation of a prioritized test suite by identifying the test cases that should be executed first and those that can be excluded.

Table (3) shows the implementation of the Greedy Algorithm for Project A Feature. The selection process ensures comprehensive coverage by including test cases that cover the most requirements first, followed by those that cover essential requirements. The 22 test cases were chosen by methodically selecting test cases that optimize requirement coverage while guaranteeing that all essential requirements are tested. Initially, Test Case 1A and Test Case 24A were selected because they cover the highest number of requirements, 12 and 10 respectively. The algorithm identifies and selects essential test cases, which cover unique requirements not addressed by any other test cases.

The discarded test cases in the implementation of the greedy algorithm are discarded because other test cases already covered all their requirements. If the requirements are already covered by other test cases, the Greedy Algorithm will consider those related test cases as redundant. If the number of covered requirements and the specific requirements covered are identical between test cases, one test case is chosen at random.

Table (4) shows the implementation of the greedy algorithm for the Project B feature. Project B test cases are able to be minimized into 8 test cases from the initial 36 test cases. Test case 29 was chosen because it covers the highest coverage with 8 requirements. The algorithm identifies and selects essential test cases, which cover unique requirements not addressed by any other test cases, which are Test Case TC20B, TC26B, TC28B, TC32B, and TC36B.

The other 3 test cases, which are Test cases TC27B, TC29B, and TC10B are also included to ensure no requirements are left untested and to reinforce coverage.

Once the writer identifies and prioritizes the essential test cases, the writer organizes them into a comprehensive test suite. This suite is carefully put together to match the main goals of the testing phase. It serves as a comparison as well between the prioritized test suite with the original ones.

Table 3: Greedy algorithm implementation on Project A feature

Test case	Total coverage	Representing essential requirements?
TC 1A	12	Yes
TC 2A	2	Yes
TC 4A	2	Yes
TC 5A	2	Yes
TC 6A	2	Yes
TC 8A	4	Yes
TC 9A	1	Yes
TC 10A	1	Yes
TC 11A	1	Yes
TC 19A	2	Yes
TC 24A	10	Yes
TC 26A	2	Yes
TC 27A	2	Yes
TC 45A	3	Yes
TC 46A	2	Yes
TC 47A	2	Yes
TC 48A	2	Yes
TC 49A	2	Yes
TC 51A	2	Yes
TC 18	4	Yes
TC 3A	2	No
TC 28A	2	No

Table 4: Greedy algorithm implementation on Project B feature

Test case	Total coverage	Representing essential requirements?
TC 20B	2	Yes
TC 26B	1	Yes
TC 28B	1	Yes
TC 32B	2	Yes
TC 36B	1	Yes
TC 27B	2	No
TC 29B	8	No
TC 10B	2	No

Test Suite Execution

The testing execution will be integrated into the Allure Test Launch test report and if there is any error or the case does not meet the expected result, the expected and actual results will be attached to the test report.

Table (5) illustrates the execution result of both the test suite for Project A. In this execution, the original test suite contained a total of 51 test cases, whereas the prioritized test suite contained 22 test cases due to Greedy Algorithm Implementation. Both of the executions resulted in 4 identical failed test cases during the test run. While the execution time shows a clear difference, the prioritized test suite, which has less number of test cases, is able to finish the execution in 1 h and 1 min. On the other hand, the original test suite finished the test execution in 2 h and 17 min.

Table 5: Test execution result on Project A feature

Description	Project A original	Project A greedy algorithm implemented
Number of test cases	51	22
Execution time	2 h 17 min	1 h 1 min
Failed test case	4 Failed test cases	4 Failed test cases

Table 6: Test execution result on Project B feature

Description	Project B Original	Project B greedy algorithm implemented
Number of test cases	36	8
Execution time	56 min 23 sec	15 min 32 sec
Failed test case	5 Failed test cases	5 Failed test cases

The results of both test suites' execution for Project B are shown in Table (6). Due to the implementation of the greedy algorithm, the prioritized test suite in these executions had 8 test cases out of the 36 total test cases in the original test suite. During the test run, both executions produced five identically failed test cases. The prioritized test suite, which contains fewer test cases, is able to complete the execution in 15 min and 32 sec, despite the execution time showing a noticeable difference. However, the original test suite took 56 min and 23 sec to complete the test execution.

Test Suite Size Reduction Rate (TSSR) Project A

From the implementation of the Greedy Algorithm in Project A and Project B, the resulting prioritized test suites for each project have been determined and are presented respectively. The number of test cases in each test suite is 22 for Project A and 8 for Project B. In detail, the table includes the following components:

1. Total initial test cases: Indicates the initial number of test cases before reduction, which is 51 test cases
2. Total test cases after reduction: Indicates the number of test cases after the reduction process using the greedy algorithm, which is 22 test cases.
3. Number of reduced test cases: Shows the number of test cases that were reduced, which is 29 test cases (51-22 = 29)

Percentage of test case reduction: Uses the TSSR formula to calculate the percentage reduction in test cases. TSSR is calculated as.

Test Suite Size Reduction Rate (TSSR) Project A

$$TSSR = \frac{51-22}{51} \times 100\% = 56.8\% \quad (3)$$

Project A, the initial 51 test cases were reduced to 22, yielding a TSSR of 56.8%. The implementation of the Greedy Algorithm in Project A has successfully reduced the test suite size significantly, which can help in reducing the time and resources required for testing while maintaining adequate test coverage.

Average Percentage Fault Detection (APFD) original Project a Test Suite

$$APFD = 1-(45+46+47+51)/(51\times 4)+\frac{1}{2\times 51} = 0.0853 \quad (4)$$

Average Percentage Fault Detection (APFD) prioritized Project a Test Suite.

$$APFD = 1-(14+15+16+22)/(22\times 4)+\frac{1}{2\times 22} = 0.2613 \quad (5)$$

There are significant differences in Project A's test suite when comparing the Average Percentage of Failures Detected (APFD) results between the Greedy algorithm's test suite and the original suite. Establishing an APFD value of 0.2613, the Greedy Algorithm method prioritizes critical test cases in order to maximize fault detection efficiency. In a shorter amount of time, the approach ensured comprehensive testing efficiently. At 0.0853, the APFD value of the original test suite was lower, despite its goal of thorough coverage tests. Larger execution durations and less effective fault detection were caused by its potential redundancy and wider coverage.

Test Suite Size Reduction Rate (TSSR) Project B

In detail, the table includes the following components:

1. Total initial test cases: Indicates the initial number of test cases before reduction, which is 36 test cases
2. Total test cases after reduction: Indicates the number of test cases after the reduction process using the greedy algorithm, which is 8 test cases.
3. Number of reduced test cases: Shows the number of test cases that were reduced, which is 28 test cases (36-8 = 28)

$$TSSR = \frac{36-8}{36} \times 100\% = 77.8\% \quad (6)$$

This high reduction rate indicates that many of the original test cases were deemed redundant. The prioritization and reduction of test cases can significantly impact the execution time, as fewer test cases mean less time needed for testing. Overall, the results show that the Greedy Algorithm effectively reduced the test suite size, leading to a more efficient testing process for Project B.

Average Percentage Fault Detection (APFD) Original Project B Test Suite

$$APFD = 1 - (10 + 20 + 26 + 28 + 36) / (36 \times 5) + \frac{1}{2 \times 36} = 0{,}3194 \quad (7)$$

Average Percentage Fault Detection (APFD) Prioritized Project B Test Suite

$$APFD = 1 - (1 + 2 + 3 + 5 + 8) / (8 \times 5) + \frac{1}{2 \times 8} = 0{,}5875 \quad (8)$$

Comparing the Average Percentage of Failures Detected (APFD) values between Project B's original test suite and the suite implemented by the Greedy algorithm provides valuable insights into their respective testing strategies. The prioritized test suite for Project B has a higher APFD of 0.5875 compared to the original test suite's APFD of 0.3194.

Discussion

The implementation of the Greedy Algorithm across Projects A and B resulted in significant reductions in the test suite sizes, as evidenced by the Test Suite Size Reduction Rate (TSSR) metrics. For Project A, the initial 51 test cases were reduced to 22, yielding a TSSR of 56.8%. This high reduction rate indicates that many of the original test cases were deemed redundant. The prioritization and reduction of test cases can significantly impact the execution time, as fewer test cases mean less time needed for testing. This efficiency gain helps save resources and accelerate the testing process while still ensuring that essential functionalities are covered.

Project B saw a reduction from 36 test cases to 8, resulting in a TSSR of 77.8%, indicating the identification and elimination of many redundant test cases. The prioritization and reduction of test cases can significantly impact the execution time, as fewer test cases mean less time needed for testing. This efficiency gain helps save resources and accelerate the testing process while still ensuring that essential functionalities are covered.

The application of the Greedy algorithm to Project A's testing strategy led to a more efficient approach by prioritizing critical test cases essential for verifying core functionalities. This optimization resulted in an APFD value of 0.2613, which is notably higher than the 0.0853 achieved by the original test suite. This enhanced APFD indicates that the Greedy algorithm significantly improved fault detection effectiveness, allowing for quicker identification of critical issues within a reduced number of test cases.

The implementation of the Greedy algorithm reduced the number of test cases from 51-22, contributing to a more streamlined and time-efficient testing process. This reduction not only improved fault detection but also significantly shortened the execution time, highlighting the Greedy algorithm's effectiveness in optimizing both test coverage and efficiency.

In contrast, the original test suite for Project A aimed to cover a broader range of scenarios and functionalities. Despite its extensive coverage, the original suite's APFD value of 0.0853 reflects lower effectiveness in detecting faults. The original suite's longer execution time, approximately 2 h and 17 min, compared to the prioritized suite's 1 h and 1 min, suggests that the broader coverage may have introduced inefficiencies and redundancy.

Comparing the Average Percentage of Failures Detected (APFD) values between Project B's original test suite and the suite implemented by the Greedy algorithm provides valuable insights into their respective testing strategies. The prioritized test suite for Project B has a higher APFD of 0.5875 compared to the original test suite's APFD of 0.3194. This indicates that the prioritization strategy has significantly improved the effectiveness of detecting faults early in the test process. The higher APFD value suggests that the prioritized test suite is better at identifying faults, and enhancing overall testing efficiency.

The implementation of the Greedy algorithm in Project B's testing strategy demonstrated a focused approach to maximizing fault detection efficiency. By prioritizing critical test cases that were essential for verifying core functionalities, the Greedy algorithm achieved an APFD value of 0.5875. This approach ensured that important requirements were thoroughly tested within a shorter execution time. The algorithm's ability to swiftly identify failures indicated a streamlined testing process that efficiently detected critical issues early in the development cycle. This efficiency in fault detection underscores the Greedy algorithm's effectiveness in optimizing test coverage without compromising the thoroughness of critical tests.

The longer execution time and potential redundancy in test case coverage may have contributed to this lower APFD value. The execution of both Project B test suites shows notable differences, original test suite finishing in 56 min and 23 sec, while the prioritized test suite finishes in 15 min and 32 sec.

The results of implementing the greedy algorithm can be summarized as follows. The application of the Greedy Algorithm led to a significant reduction in the test suite sizes, improved fault detection effectiveness, and a notable decrease in execution time for both Project A and

Project B. The detailed calculations for Average Percentage Fault Detection (APFD) and execution time reduction are as follows:

$$APFD\,increas = \frac{APFD_{prioritized} - APFD_{original}}{APFD_{original}} \times 100\% \quad (9)$$

Average Percentage Fault Detection (APFD) Increase in Project A

$$Project\,A = \frac{0.2613 - 0.0853}{0.0853} \times 100\% = 206.21\% \tag{10}$$

For *Project A*, the *APFD* value increased by 206.21%, from 0.0853-0.2613, which indicates a substantial enhancement in detecting faults early in the testing process. This improvement suggests that the greedy algorithm successfully prioritized the test cases that were more likely to uncover defects, thus enhancing the efficiency of the testing process.

Average Percentage Fault Detection (APFD) Increase in Project B

$$Project\,B = \frac{0.5875 - 0.3194}{0.3194} \times 100\% = 83.92\% \tag{11}$$

Project B APFD increased by 83.92%, from 0.3194-0.5875, showing a similar improvement in fault detection effectiveness. This increase demonstrates the algorithm's ability to prioritize test cases that uncover faults more effectively, thus contributing to higher software quality.

Time Reduction Percentage

The reduction in execution time reflects the efficiency gains achieved through test case prioritization will calculated as follows:

$$Time\,Reduction = \frac{Time_{original} - Time_{prio}}{Time_{original}} \times 100\% \tag{12}$$

Time Reduction Percentage Project A

$$Time\,Reduction = \frac{137 - 61}{137} \times 100\% = 55.47\% \tag{13}$$

where, in *Project A*:

- Original execution time = 137 min
- Prioritized execution time = 61 min

In Project A, the execution time decreased by 55.47%, from 137-61 min, highlighting a significant improvement in testing efficiency. This reduction was achieved by prioritizing test cases, which led to a smaller, more efficient suite that could be executed more quickly.

Time Reduction Percentage Project B

$$Time\,Reduction = \frac{56.38 - 15.53}{56.38} \times 100\% = 72.44\% \tag{14}$$

where, in *Project A*:

- Original execution time = 56.38 min
- Prioritized execution time = 15.53 min

For Project B, the execution time was reduced by 72.44%, from 56.38-15.53 min, demonstrating even greater efficiency gains. This timesaving directly contributes to faster test cycles, aligning with the goal of reducing testing durations and accelerating the software release process.

The overall Average Percentage Fault Detection (APFD) value indicates that the prioritization strategy has significantly enhanced the effectiveness of early fault detection in the testing process. A higher APFD value reflects that the prioritized test suite is more adept at identifying faults, thereby improving overall testing efficiency. The accuracy of fault detection is crucial for ensuring software quality and optimizing the use of testing resources. By implementing the Greedy Algorithm for prioritization, teams can effectively select test cases that are most likely to reveal defects early in the testing cycle. This approach not only increases the chances of identifying critical faults but also minimizes the time and effort required for executing tests that may have less impact. While there is an inherent risk in testing processes due to the possibility of missed scenarios, PT XYZ has addressed this by hiring a third-party team to manually test edge cases. The internal team focuses solely on testing based on the project requirements and acceptance criteria.

The research that was conducted reinforces previous studies by demonstrating that the Greedy Algorithm effectively prioritizes test suites, eliminates redundant test cases, and enhances the efficiency of the testing process through the elimination of test cases based on redundant requirement coverage. This study substantiates the findings of Srivastava (2008); Harris (2015); Singh *et al*., (2016); Jehan and Wotawa (2023), which showed that implementing the Greedy Algorithm and evaluating it using Test Suite Size Reduction (TSSR) and Average Percentage of Faults Detected (APFD) results in superior outcomes compared to the original test suite. A novel aspect of this research is its application during the regression testing phase and the use of automated testing with Web driver IO using the current projects. This research not only confirms the efficiency of the Greedy Algorithm but also showcases its effectiveness in modern automated testing environments, further optimizing test case execution and improving overall testing efficiency.

Previous research examining the application of Greedy algorithms in regression test optimization has shown significant potential in reducing test suite size and improving fault detection efficiency. For example, a study by Singh *et al*. (2016), showed that the Greedy algorithm was able to reduce Test Suite Size Reduction (TSSR) by

50% compared to traditional optimization techniques. Similarly, a study by Srivastava (2008), highlighted that Average Percentage Fault Detection (APFD) improved by 11% when using this algorithm.

While these results are promising, there are some unaddressed gaps in the previous research. Firstly, previous research did not mention the domain feature of the test cases sample, whereas this research is using the Core Transaction Payment System feature at the company, using specified test cases that related to the transaction feature. Secondly, most of the previous studies did not measure the impact of applying the Greedy algorithm on regression test execution time at scale, especially in a constantly changing environment such as that experienced by the company.

This research aims to fill the gap by investigating the adaptation of the Greedy algorithm in the context of regression testing involving dynamic data and system configuration. One of the main focuses is to significantly reduce the test execution time compared to the currently used testing approach, so as to be able to fulfill operational needs more quickly and efficiently.

Conclusion

The implementation of the Greedy Algorithm is a technique for achieving effective Test Case Prioritization. The Greedy Algorithm works by selecting test cases that cover the most critical requirements or are most likely to detect faults, optimizing the test suite to reduce redundancy and enhance efficiency. As demonstrated in Projects A and B, the Greedy Algorithm significantly reduced the test suite sizes while maintaining or even improving fault detection effectiveness, as evidenced by the increased Average Percentage of Faults Detected (APFD) values.

By reducing the number of test cases and prioritizing the most critical ones, the execution time for regression testing can be significantly shortened, making regression testing more efficient, and reducing the time and resources needed while still ensuring comprehensive coverage of critical functionalities. This allows for faster feedback and quicker identification of potential issues, enabling more efficient use of resources.

Overall, the application of the Greedy Algorithm has shown varying degrees of success in optimizing test suite sizes and improving fault detection effectiveness across different projects. In Project A, the test suite was reduced from 51-22 test cases, achieving a TSSR of 56.8% (a reduction of 43 test cases or 84.3%). For Project B, the Greedy Algorithm delivered even more impressive results, reducing the test suite from 36-8 test cases, resulting in a TSSR of 77.8% (a reduction of 28 test cases or 77.8%). Projects A and B showed notable reductions and improvements in efficiency, with APFD values rising by 206.2% (from 0.0853-0.2613) in Project A and by 83.9% (from 0.3194-0.5875) in Project B. This optimization reduced redundant test cases, sped up testing, and supported the company's goal of achieving faster release cycles.

Acknowledgment

We would like to thank Bina Nusantara University for providing the resources and support necessary for this research. We also acknowledge the valuable contributions of our colleagues and collaborators, whose expertise and assistance were greatly appreciated.

Funding Information

This research was entirely self-funded and conducted without external financial support.

Author's Contributions

Akira Wahyu Putra: Conducting the whole research phase. Gather data needed to conduct the research.

Nilo Legowo: Guiding the whole research phase. Giving feedback and constructive opinions related to the research.

Ethics

The authors confirm that this manuscript has not been published elsewhere and that no ethical issues are involved.

Reference

Alian, M., Suleiman, D., & Shaout, A. (2016). Test Case Reduction Techniques - Survey. *International Journal of Advanced Computer Science and Applications*, *7*(5). https://doi.org/10.14569/ijacsa.2016.070537

Ansari, A., Khan, A., Khan, A., & Mukadam, K. (2016). Optimized Regression Test Using Test Case Prioritization. *Procedia Computer Science*, *79*, 152–160. https://doi.org/10.1016/j.procs.2016.03.020

Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2015). Coverage-based regression test case selection, minimization, and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability*, *25*(4), 371–396. https://doi.org/10.1002/stvr.1572

Elbaum, S., Rothermel, G., Kanduri, S., & Malishevsky, A. G. (2004). Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Journal*, *12*(3), 185–210. https://doi.org/10.1023/b:sqjo.0000034708.84524.22

Gladston, A., Nehemiah, H. K., Narayanasamy, P., & Kannan, A. (2016). Test Suite Reduction Using HGS Based Heuristic Approach. *Computing and Informatics*, *34*(5), 1113–1132.

Greca, R., Miranda, B., & Bertolino, A. (2023). State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review. *ACM Computing Surveys*, *55*(13s), 1–36. https://doi.org/10.1145/3579851

Hettiarachchi, C., Do, H., & Choi, B. (2016). Risk-based test case prioritization using a fuzzy expert system. *Information and Software Technology*, *69*, 1–15. https://doi.org/10.1016/j.infsof.2015.08.008

Jehan, S., & Wotawa, F. (2023). An Empirical Study of Greedy Test Suite Minimization Techniques Using Mutation Coverage. *IEEE Access*, *11*, 65427–65442. https://doi.org/10.1109/access.2023.3289073

Khatibsyarbini, M., Isa, M. A., Jawawi, D. N. A., & Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, *93*, 74–93. https://doi.org/10.1016/j.infsof.2017.08.014

Li, Z., Harman, M., & Hierons, R. M. (2007). Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, *33*(4), 225–237. https://doi.org/10.1109/tse.2007.38

Lin, C.-T., Tang, K.-W., Wang, J.-S., & Kapfhammer, G. M. (2017). Empirically evaluating Greedy-based test suite reduction methods at different levels of test suite complexity. *Science of Computer Programming*, *150*, 1–25. https://doi.org/10.1016/j.scico.2017.05.004

Lou, Y., Chen, J., Zhang, L., & Hao, D. (2019). Chapter One - A Survey on Regression Test-Case Prioritization. In A. M. Memon (Ed.), *Advances in Computers* (Vol. 113, pp. 1–46). Elsevier. https://doi.org/10.1016/bs.adcom.2018.10.001

Maspupah, A., Muharram, M. K., & Daeli, S. G. (2023). Analisis Efektifitas Algoritma FAST Menggunakan Metrik Average Percentage Fault Detection dan Waktu Eksekusi Pada Test Case Prioritization. *Journal of Information System Research (JOSH)*, *4*(2), 451–457. https://doi.org/10.47065/josh.v4i2.2822

Qasim, M., Bibi, A., Hussain, S. J., Jhanjhi, N. Z., Humayun, M., & Sama, Najm Us. (2021). Test case prioritization techniques in software regression testing: An overview. *International Journal of Advanced and Applied Sciences*, *8*(5), 107–121. https://doi.org/10.21833/ijaas.2021.05.012

Qiu, D., Li, B., Ji, S., & Leung, H. (2015). Regression Testing of Web Service: A Systematic Mapping Study. *ACM Computing Surveys*, *47*(2), 1–46. https://doi.org/10.1145/2631685

Singh, S., & Shree, R. (2016). An Analysis of Test Suite Minimization Techniques. *International Journal of Engineering Sciences & Research Technology International Journal of Engineering Sciences & Research Technology*, *5*(11), 252–260. https://doi.org/10.5281/zenodo.165632

Srivastava, P. R. (2008). Test case prioritization. *Journal of Theoretical and Applied Information Technology*, *5*, 178–181.

Sutapa, F. A. K. P. G., Kusumawardani, S. S., & Permanasari, A. E. (2020). A Review of Automated Testing Approach for Software Regression Testing. *IOP Conference Series: Materials Science and Engineering*, *846*, 012042. https://doi.org/10.1088/1757-899x/846/1/012042

Wong, W. E., Horgan, J. R., London, S., & Mathur, A. P. (1995). Effect of test set minimization on fault detection effectiveness. *ICSE '95: Proceedings of the 17th International Conference on Software Engineering*, 41–50. https://doi.org/10.1145/225014.225018

Yamuç, A., Cingiz, M. O., Biricik, G., & Kalipsiz, O. (2017). Solving test suite reduction problem using greedy and genetic algorithms. *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 1–5. https://doi.org/10.1109/ECAI.2017.8166445

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, *22*(2), 67–120. https://doi.org/10.1002/stvr.430