***Optimization and Prioritization of Test Cases through Hungarian Algorithm***

*Abstract:* Test cases play a significant role in developing high-quality software products. When test cases are generated over the modules of the software product then it is necessary to generate optimized test cases that are feasible for the modules and save the time of testing. The present paper is an attempt to generate test cases that may be valid or invalid and further valid test cases are prioritized using Hungarian method which produces the optimized time of testing and can be applied to any testing strategy. Generally, testing is used to test the software products through generation of optimized test cases which are prioritized through the proposed technique. Computed results are depicted in tabular and graphical forms.

*Keywords:*Testing Strategy, Test Cases, Hungarian Method, Prioritization, Optimization.

# 1. Introduction

Software testing assures the reliability and quality of software systems, which makes it a crucial step in the development process. However, as software programs become more complex, it becomes more difficult and highly resource-intensive to test every potential scenario. Thus, there is a dire demand for efficient methods for sorting through and improving the test cases. In response to this need, this research study offers a novel approach known as the "Hungarian algorithm" for optimization and prioritization of the test cases. The Hungarian algorithm provides a number of significant advantages when utilized for the prioritization and optimization of test cases. The first benefit is that it effectively reduces the time and cost associated with software testing, which is highly beneficial in the hurried and cost-conscious software development environment of today. Additionally, this approach significantly increases testing efficiency by wisely directing resources to high-impact test cases. Using mathematical operations like square\_number, cube\_number, absolute\_value, factorial, and fibonacci\_number, the said concept is successfully applied. The findings of the present study conclusively show that the cost and time of software testing have greatly dropped, which has enhanced the quality of the software products as a result. Along with software testers and developers, it also benefits to the end users who have access to higher-quality software solutions. From the literature, it is revealed that the said approach is never been applied for the testing strategies, but several studies have been conducted in the past to develop various techniques for test case prioritization. Let us describe some of the important references. In the year 2015, Rhmann et al. [1] introduced a novel approach for test case prioritization using UML activity diagrams that makes use of a genetic algorithm. By taking into account the complexity and risk of the UML activity diagram, the approach generates a prioritized test suite that covers all edges and nodes. The results showed that prioritized test cases performed better in fault discovery than test cases prioritized using traditional methods. In the year 2016, Rhmann et al. [2] demonstrated an approach for optimizing test cases that was applied to select and rank test cases based on fault exposure, requirement coverage, risk, statement coverage, and execution time. By making the most of the testing time available, software testing efficiency was increased. Wasiur Rhmann and V. S. [3] utilized extenics theory to accelerate the generation of test cases from UML sequence diagrams. They designed a test case technique, formalized the sequences, and used XML to transform them into message flow graphs. The airplane departure scenario used in the study served as an example of how effectively the technique produced practical test cases. Rhmann and Saxena [4] modified the firefly method to produce prioritized and effective test cases from UML activity diagrams. The method considers the complexity, risk, and coverage of the UML activity diagram when creating test pathways. Compared to test cases generated using standard methods, test cases created using the suggested procedure were able to identify more problems.

In the year 2019, Alkawaz and Silvarajoo[5] surveyed test case prioritization and optimization techniques in software regression testing and discussed code coverage-based, risk-based, and requirement-based prioritization approaches, highlighting the advantages, limitations, and the existing research in the field. The survey provides valuable insights into the current state of test case prioritization and optimization for software regression testing. Lu et al. [6] introduced an Ant Colony System (ACS) with sorting-based local search for coverage-based test case prioritization. The optimization algorithm combines the behavior of the algorithm with sorting-based local search to enhance the prioritization in an effective manner. The authors applied the approach to coverage-based test case prioritization and evaluated its performance against other techniques. The results emphasize the effectiveness of the ACS algorithm in improving test case prioritization for software reliability. Huang et al. [7] presented an abstract test case prioritization technique utilizing repeated small-strength level-combination coverage. The paper introduces a novel coverage metric and puts forth a prioritization strategy based on this metric. The authors conducted experiments to assess the effectiveness of the approach and compared it with other state-of-the-art techniques. The results highlight the efficiency of the proposed prioritization method in reducing the number of test cases while maintaining adequate coverage. Bajaj and Sangwan [8] conducted a literature review on genetic algorithm-based test case prioritization techniques. The paper provides an overview of research in this field, discussing concepts, challenges, and trends. The authors analyzed and summarized the findings from various studies, including objectives, methodologies, and performance evaluations. The review highlights the potential of genetic algorithms in optimizing test case prioritization and identifies future research areas.

In the year 2020, Taneja et al. [9] proposed a novel technique for minimizing the test cases in object-oriented testing. The approach is based on the concept of class dependency and utilizes a greedy algorithm to select the most crucial test cases. The authors evaluated the approach on real-world object-oriented software systems, demonstrating its significant capability to reduce the test suite's size without compromising fault detection ability. Bajaj and Sangwan [10] discussed the utilization of nature-inspired approaches for minimizing test suites in regression testing and focused on three specific approaches namely genetic algorithms, particle swarm optimization, and ant colony optimization. Evaluating the approaches on real-world software systems, the algorithms show effectiveness in reducing the test suite's size while maintaining fault detection ability. Mohapatra et al. [11] proposed an intelligent local search algorithm for test case minimization. The approach relied on the concept of test case similarity and employed a local search algorithm to identify a set of similar yet effective test cases for fault detection. By evaluating the approach on real-world software systems, authors demonstrated its significant reduction in test suite size without compromising fault detection ability. Zhou et al. [12] compared the effectiveness of random test case prioritization with other approaches, including coverage-based prioritization, fault-based prioritization, and risk-based prioritization. Through evaluations of real-world software systems, authors found that random test case prioritization was less effective as compared to the other approaches. Dhareula and Ganpati [13] introduced a flower pollination algorithm for test case prioritization in regression testing. The approach is inspired by flower pollination, utilizes a population-based algorithm to select efficient and effective test cases for fault detection. By evaluating the approach on real-world software systems, authors demonstrated its ability to significantly reduce test suite execution time without compromising fault detection ability. Wang et al. [14] proposed a regression test case prioritization method based on the fixed-size candidate set ART algorithm. The approach, grounded in adaptive random testing, employs a fixed-size candidate set of test cases to reduce test suite execution time. Through evaluations on real-world software systems, authors showed the approach's capability to significantly decrease execution time without compromising fault detection ability. Chi et al. [15] put forward a relation-based test case prioritization approach for regression testing. The approach focuses on test case relations and employs a greedy algorithm to select critical test cases and demonstrates its ability to substantially reduce test suite execution time while maintaining fault detection ability.

In the year 2021, Bajaj and Sangwan [16] proposed a novel approach for test case prioritization using discrete cuckoo search algorithms. The authors argue that traditional methods for test case prioritization are ineffective for large software systems due to computational cost and lack of scalability. The proposed approach utilizes a discrete cuckoo search algorithm to search for an optimal test case prioritization solution. The algorithm is evaluated on a set of real-world software systems, and the results demonstrate a significant improvement in the efficiency of test case prioritization. Boyar et al. [17] presented a novel approach for test case prioritization in software regression testing. Traditional methods for test case prioritization do not effectively consider the changes made to the software since the last test. The proposed approach introduces a new algorithm that prioritizes test cases based on the software modifications. The algorithm is evaluated on real-world software systems, and the results show a substantial enhancement in the effectiveness of regression testing. Khatibsyarbini et al. [18] conducted a literature review on the utilization of machine learning in test case prioritization. The authors argue that machine learning can enhance the effectiveness of test case prioritization by incorporating factors such as defect history, software structure, and user behavior. The authors surveyed various machine learning techniques employed in test case prioritization and discussed the advantages and disadvantages associated with each technique. Huang et al. [19] proposed a learn-to-rank approach for model-based regression test case prioritization. The authors argued that traditional methods for test case prioritization are inadequate for model-based regression testing as they fail to consider the relationships between test cases. The proposed approach employed a learn-to-rank method to prioritize test cases based on the predicted ability to detect defects. The effectiveness of the method is evaluated on real-world software systems, and the results exhibit a significant improvement in model-based regression testing. Hasnain et al. [20] conducted a systematic literature review on functional requirement-based test case prioritization in regression testing. The authors identified 52 relevant papers and analyzed each one to identify different approaches to functional requirement-based test case prioritization. The authors also discussed the challenges associated with functional requirement-based test case prioritization and proposed future research directions in this field. Laaber et al. [21] applied test case prioritization to software microbenchmarks. The authors argued that test case prioritization can enhance the efficiency of software micro benchmarking and proposed a novel approach for test case prioritization specifically tailored for software microbenchmarks. The effectiveness of the approach is evaluated on real-world software systems, and the results demonstrated a significant improvement in the efficiency of software micro benchmarking.

In the year 2022, Raamesh et al. [22] introduced a novel approach to regression testing that focuses on test case minimization and prioritization and utilizes an SBLA-based AdaBoost convolutional neural network to learn the significance of test cases. The SBLA-based AdaBoost convolutional neural network combines the strengths of Support Vector Machines (SVMs), AdaBoost, and convolutional neural networks. SVMs are employed to learn the importance of test cases, the AdaBoost algorithm is utilized to merge the predictions of the SVMs, and the convolutional neural network learns the spatial relationships between the test cases. Vescan et al. [23] proposed the ANT algorithm with faults severity, a new test case prioritization algorithm. The ANT algorithm employs a genetic algorithm to identify the optimal test case prioritization. Fault severity is used as a guiding factor in the search process of the ANT algorithm. Fault severity serves as a measure of the likelihood of detecting a fault through a particular test case. Dahiya et al. [24] conducted a performance comparison between the TFC-SVM approach and the random approach for regression test case prioritization. The TFC-SVM approach utilizes an SVM to learn the significance of test cases. On the other hand, the random approach randomly orders the test cases. The comparison results indicate that the TFC-SVM approach outperforms the random approach in terms of fault detection rate. Demir and Amrahov [25] proposed the Dominating Set-based Test Prioritization (DSTP) algorithm as a new test prioritization technique. The DSTP algorithm employs a dominating set to prioritize test cases. A dominating set represents a set of test cases that covers all code paths within the software system. The DSTP algorithm utilizes a genetic algorithm to identify the optimal dominating set. Bajaj et al. [26] introduced the Improved Novel Bat Algorithm (INBA) as a new algorithm for test case prioritization and minimization. The INBA algorithm is based on the bat algorithm, which is a metaheuristic algorithm inspired by the echolocation behavior of bats. It aims to find the optimal test case prioritization and minimization. Iqbal and Al-Azzoni [27] proposed a novel test case prioritization approach specifically designed for model transformations. The approach employs a genetic algorithm to identify the optimal test case prioritization. The genetic algorithm focuses on finding the test cases that are most likely to detect faults in the model transformations.

In the year 2023, Khaleel and Anan [28] conducted a review on the utilization of artificial intelligence (AI) techniques for test case prioritization. The review paper thoroughly explores the various AI techniques employed in test case prioritization, delving into the advantages and disadvantages associated with each technique. Additionally, the paper addresses the challenges that arise when implementing AI in test case prioritization. Furthermore, the paper provides insights into the future prospects of AI in the field of test case prioritization.

# 2. Methodology

# In the proposed work, an optimization technique is used for optimization and prioritization of the test cases which play a vital role in the efficient development of the software products. In this regard, Hungarian technique is used which is a trustworthy optimization method with an outstanding reputation for handling assignment issues successfully, especially when optimal solutions are required. The following steps are considered for the optimization and prioritization of the test cases:

## *A. Step1: Data Collection and Preprocessing*

1. Test Case Selection: A group of test cases are selected that accurately represent the functionality of the system under test. The test cases should have expectations for the outcomes that are apparent;
2. Function Selection: Simultaneously, a list of relevant functions are selected, including the square, cube, absolute, factorial, and Fibonacci numbers, etc. that will be employed in the optimization process.

## *B. Step2. Initialization*

1. Initializing Variables: In this step, two key variables are required to be initialized:
   * test\_results: which represents the set of valid test cases;
   * functions: a list of the selected functions.

## *C. Step3. Construction of Cost Matrices*

1. Execution and Timing: Every test case is executed using every selected function, tracking the execution times. Three cost matrices are produced in this phase, each of which shows the execution timings for a different function;
2. Matrix Population: The test results and functions are repeatedly run through the cost matrices to capture the execution time for every feasible combination of test case and function.

## *D. Step4: Average of Matrices*

Matrix Averaging: A function called average\_of\_matrices() is used to calculate the average matrix by adding the three cost matrices element by element. Each test case and function pair's average execution times are stored in the average matrix.

## *E. Step5. Computation via Hungarian Algorithm*

1. Conversion to Numpy Arrays: Before using the Hungarian Algorithm, the cost matrices are converted into NumPy arrays to make the algorithm compatible;
2. Hungarian Algorithm: The Hungarian algorithm is used by specialists in combinatorial optimization to determine the optimal solution to the assignment problem. This is accomplished by finding the assignment that, in this case, reduces the overall cost or execution time. The algorithm's efficiency and optimality make it a helpful tool for optimizing software testing. It determines the most efficient approach to pairing test cases with functions to minimize testing time.

## *F. Step6.* ***Optimized and Prioritized Test Cases***

1. **Minimizing Total Testing Time**: Employing the Hungarian Algorithm, we focus on the effective pairing of test cases with functions to provide optimized test cases that reduce the overall testing time.
2. **Sorting for Time Efficiency**: The optimized test cases are prioritized to help shorten the total testing period by being arranged in decreasing order of execution times. This further speeds up the testing process.
3. **Efficient Prioritization**: By giving time-consuming test cases precedence, the subsequent prioritizing process successfully reduces the overall testing duration, improving the program's overall quality and testing efficiency.

## *G. Step7. Total Testing Time*

* Calculate Total Testing Time: The average matrix's values are computed using the Hungarian Algorithm's provided indexes, and added to derive the overall testing time. This statistic serves as a crucial performance indicator by measuring the projected testing time after optimization.

The Hungarian algorithm is used in this methodology to prioritize and improve test cases in an organized manner. The Hungarian method plays a crucial role in achieving effective and efficient software testing by reducing testing time. It efficiently pairs test cases with functions.

# 3. Results and Discussion

The main objective of the present work is to solve an optimization problem with the Hungarian approach in order to reduce the total time that test cases require to run. To prioritize and optimize the test cases, a range of functions is employed over square\_number (), cube\_number (), absolute\_value (), factorial (), and fibonacci\_number (), to measure the execution times of the test cases. First, random techniques are used to generate a series of test cases. Next, the selection of the only valid test cases is done, and then keep a list of test cases. For each test case, the duration of execution time is recorded very carefully for each function which executes in order to produce a 5x5 cost matrix. The rows in this matrix represent test cases, and the columns represent functions. The Hungarian algorithm is used very effectively to optimize the assignment of test cases to functions and minimize the total execution time. This efficient algorithm provides an optimal way to pair test cases with functions, making test case prioritizing easier and testing process efficiency higher. Further, the steps are given below:

*A. Step1. Test Case Execution and Cost Matrices*: Executing the test cases using different functions and measuring the execution times. Three cost matrices are generated: cost\_matrix11, cost\_matrix12, and cost\_matrix13.

* cost\_matrix11: This matrix represents the execution times of the test cases using the functions. Each row corresponds to a test case, and each column corresponds to a function. The values in the matrix represent the execution time in scientific notation up to two decimal digits.

Table 1. Representation of Cost\_matrix11

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test\_Case\_ID | square\_number() | cube\_number() | absolute\_value() | factorial() | fibonacci\_number() |
| TC\_ID\_1 | 1.91e-06 | 1.67e-06 | 1.43e-06 | 2.62e-06 | 1.91e-06 |
| TC\_ID\_2 | 9.54e-07 | 7.15e-07 | 4.77e-07 | 9.54e-07 | 1.19e-06 |
| TC\_ID\_3 | 4.77e-07 | 7.15e-07 | 2.38e-07 | 4.77e-07 | 4.77e-07 |
| TC\_ID\_4 | 4.77e-07 | 7.15e-07 | 4.77e-07 | 1.19e-06 | 1.19e-06 |
| TC\_ID\_5 | 4.77e-07 | 4.77e-07 | 2.38e-07 | 7.15e-07 | 7.15e-07 |

* cost\_matrix12 and cost\_matrix13 are represented in a similar fashion and given below:

Table 2. Representation of Cost\_matrix12

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test\_Case\_ID | square\_number() | cube\_number() | absolute\_value() | factorial() | fibonacci\_number() |
| TC\_ID\_1 | 1.91e-06 | 9.54e-07 | 7.15e-07 | 1.91e-06 | 1.43e-06 |
| TC\_ID\_2 | 4.77e-07 | 4.77e-07 | 4.77e-07 | 7.15e-07 | 1.43e-06 |
| TC\_ID\_3 | 7.15e-07 | 4.77e-07 | 4.77e-07 | 7.15e-07 | 9.54e-07 |
| TC\_ID\_4 | 7.15e-07 | 4.77e-07 | 2.38e-07 | 9.54e-07 | 9.54e-07 |
| TC\_ID\_5 | 7.15e-07 | 4.77e-07 | 2.38e-07 | 7.15e-07 | 2.38e-07 |

Table 3. Representation of Cost\_matrix13

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test\_Case\_ID | square\_number() | cube\_number() | absolute\_value() | Factorial() | fibonacci\_number() |
| TC\_ID\_1 | 1.43e-06 | 7.15e-07 | 1.67e-06 | 1.91e-06 | 1.67e-06 |
| TC\_ID\_2 | 4.77e-07 | 4.77e-07 | 4.77e-07 | 7.15e-07 | 1.19e-06 |
| TC\_ID\_3 | 4.77e-07 | 7.15e-07 | 7.15e-07 | 7.15e-07 | 4.77e-07 |
| TC\_ID\_4 | 4.77e-07 | 4.77e-07 | 7.15e-07 | 1.19e-06 | 1.19e-06 |
| TC\_ID\_5 | 9.54e-07 | 7.15e-07 | 4.77e-07 | 4.77e-07 | 4.77e-07 |

*B. Step2. Average Matrix Calculation*: The average matrix, average1, is computed by taking the element-wise average of the three cost matrices. This matrix represents the average execution time for each test case and function combination.

Table 4. Representation of Average of the Cost Matrix1 (5x5)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test\_Case\_ID | square\_number() | cube\_number() | absolute\_value() | factorial() | fibonacci\_number() |
| TC\_ID\_1 | 1.75e-06 | 1.11e-06 | 1.27e-06 | 2.15e-06 | 1.67e-06 |
| TC\_ID\_2 | 6.36e-07 | 5.56e-07 | 4.77e-07 | 7.95e-07 | 1.27e-06 |
| TC\_ID\_3 | 5.56e-07 | 6.36e-07 | 4.77e-07 | 6.36e-07 | 6.36e-07 |
| TC\_ID\_4 | 5.56e-07 | 5.56e-07 | 4.77e-07 | 1.11e-06 | 1.11e-06 |
| TC\_ID\_5 | 7.15e-07 | 5.56e-07 | 3.18e-07 | 6.36e-07 | 4.77e-07 |

*C. Step3. Hungarian Algorithm*: Next, the Hungarian algorithm is utilized to find the optimal assignment of test cases to functions, minimizing the total execution time. For this purpose, the Munkres class is used from the munkres module.

*D. Step4. Prioritized Test Cases*: The priority needs to be further adjusted after the Hungarian algorithm yields the optimal test cases. Execution time is used as the main criterion and rank the perfect test cases in descending order. This multi-step technique generates a prioritizing sequence that places the most critical and time-consuming test cases higher in order to improve testing efficiency. Effective prioritization improves software testing by reducing total execution time and guaranteeing the achievement of important testing goals.

From the table 4, the optimum solution via Hungarian method is given below:

Test\_Case\_1🡪 cube\_number() represents that the test\_case\_1 is assigned to the cube\_number(), similar interpretation is used for other optimum values like Test\_ Case\_2🡪absolute\_value(), Test\_Case\_3 🡪factorial(), Test\_Case\_4 🡪square\_number(), Test\_Case\_5 🡪 fibonacci\_number().

E. Step5. Total Time Calculation: The total time is computed by summing the execution times from the average1 matrix based on the assigned indexes. The total time represents the minimum total execution time achieved by the Hungarian algorithm.

**Total Time:** 1.59e-06 seconds

Additionally, the matrix expansion method is used to expand the matrix and get the overall time. First, a cost matrix is created that was 10x5 order. The cost matrix (10x5) is averaged after three iterations and represented below:

Table 5. Representation of Average of the Cost Matrix2 (10x5):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test\_Case\_ID | square\_number() | cube\_number() | absolute\_value() | factorial() | fibonacci\_number() |
| TC\_ID\_1 | 2.07e-06 | 1.19e-06 | 1.27e-06 | 2.15e-06 | 1.11e-06 |
| TC\_ID\_2 | 5.56e-07 | 4.77e-07 | 3.97e-07 | 7.15e-07 | 1.03e-06 |
| TC\_ID\_3 | 4.77e-07 | 4.77e-07 | 2.38e-07 | 7.95e-07 | 4.77e-07 |
| TC\_ID\_4 | 7.15e-07 | 5.56e-07 | 3.18e-07 | 1.03e-06 | 1.03e-06 |
| TC\_ID\_5 | 5.56e-07 | 4.77e-07 | 2.38e-07 | 6.36e-07 | 4.77e-07 |
| TC\_ID\_6 | inf | inf | inf | inf | inf |
| TC\_ID\_7 | inf | inf | inf | inf | inf |
| TC\_ID\_8 | inf | inf | inf | inf | inf |
| TC\_ID\_9 | inf | inf | inf | inf | inf |
| TC\_ID\_10 | inf | inf | inf | inf | inf |

\* inf: dummy value

The total time for the 10x5 matrix expansion is **1.59e-06** seconds.

Similarly, a cost matrix (5x10) is computed by expanding the matrix size. After iterating three times, the average of the cost matrix (5x10) is represented below:

Table 6. Representation of Average of the Cost Matrix3 (5x10)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test\_Case\_ID | square\_number() | cube\_number() | absolute\_value() | factorial() | fibonacci\_number() | dummy\_1 | dummy\_2 | dummy\_3 | dummy\_4 | dummy\_5 |
| TC\_ID\_1 | 1.59e-06 | 7.15e-07 | 5.56e-07 | 7.15e-07 | 4.77e-07 | inf | inf | inf | Inf | inf |
| TC\_ID \_2 | 1.03e-06 | 4.77e-07 | 7.15e-07 | 4.77e-07 | 6.36e-07 | inf | inf | inf | Inf | inf |
| TC\_ID \_3 | 1.11e-06 | 3.18e-07 | 3.18e-07 | 2.38e-07 | 3.18e-07 | inf | inf | inf | Inf | inf |
| TC\_ID \_4 | 1.83e-06 | 1.03e-06 | 6.36e-07 | 9.54e-07 | 7.95e-07 | inf | inf | inf | Inf | inf |
| TC\_ID \_5 | 1.43e-06 | 1.27e-06 | 3.97e-07 | 1.03e-06 | 3.97e-07 | inf | inf | inf | Inf | inf |

\* inf: dummy value

The total time for the 5x10 matrix expansion is **1.59e-06** seconds.

The computed results show that all matrix expansions of dimensions 5x5, 10x5, and 5x10 result in an execution time of 1.59e-06 in total. These findings offer unmistakable proof of the proposed method's effectiveness in improving test case prioritizing. The system's performance has significantly improved when the Hungarian algorithm is used for test case allocation. The research's matrix expansion method also demonstrates a high degree of scalability and flexibility. It's ideally suited for applications with more complicated testing requirements and diversified function sets because it can easily handle larger collections of test cases and functions. Fig. 1 compares the visible execution timings of test cases before and after optimization. The graph is created using the widely used Python data visualization tool Matplotlib. This visual comparison aims to evaluate the efficiency gains obtained during the optimization process. The bars labeled 'Before Optimization' and 'After Optimization' display the mean times at which each test case was executed. This graphical analysis offers valuable information regarding the optimization's consequences in terms of time savings and overall software testing efficiency.

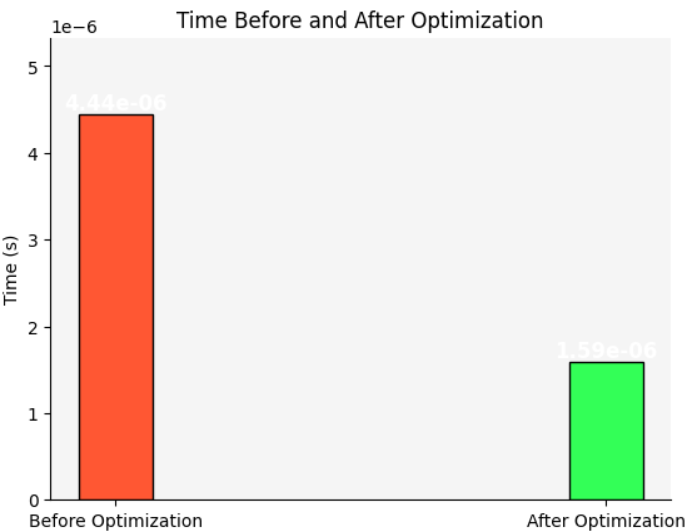


Figure 1. Representation of Time Comparison (Before vs. After Optimization)

From the review of the work, table 6 provides a detailed overview of the numerous approaches used for test case prioritization along with the key factors that each researcher carefully considered in the research.

Table 7. Comparative Analysis of Test Case Prioritisation Approaches

|  |  |  |  |
| --- | --- | --- | --- |
| Author | Year | Prioritization Technique Used | Key Factors Considered |
| Lu et al. [6] | 2019 | Ant Colony System with Sorting-Based Local Search | Coverage-based test case prioritization |
| Huang et al. [7] | 2019 | Abstract Test Case Prioritization using Repeated Small-Strength Level-Combination Coverage | Repeated small-strength level-combination coverage |
| Taneja et al. [9] | 2020 | Novel Technique for Test Case Minimization | Object-oriented testing, test case minimization |
| Bajaj and Sangwan [10] | 2020 | Nature-Inspired Approaches to Test Suite Minimization | Regression testing, nature-inspired approaches |
| Mohapatra et al. [11] | 2020 | Intelligent Local Search for Test Case Minimization | Intelligent local search, test case minimization |
| Zhou et al. [12] | 2020 | Beating Random Test Case Prioritization | Random test case prioritization |
| Dhareula and Ganpati [13] | 2020 | Flower Pollination Algorithm for Test Case Prioritization | Regression testing, flower pollination algorithm |
| Wang et al. [14] | 2020 | Regression Test Case Prioritization Based on Fixed Size Candidate Set ART Algorithm | Fixed-size candidate set ART algorithm, regression testing |
| Chi et al. [15] | 2020 | Relation-Based Test Case Prioritization | Regression testing, relation-based prioritization |
| Bajaj and Sangwan [16] | 2021 | Discrete Cuckoo Search Algorithms for Test Case Prioritization | Discrete cuckoo search algorithms |
| Present Approach | 2023 | Hungarian Algorithm-Based Optimization and Prioritization | Hungarian algorithm, test case prioritization, optimization |

# 4. Conclusions

This research introduces a novel approach for test case prioritization and optimization using the Hungarian algorithm, aiming to improve the efficiency of software testing. The combination of the random technique for test case generation and the Hungarian algorithm for prioritization offers significant benefits. The empirical results, based on the execution of test cases using functions such as square\_number (), cube\_number (), absolute\_value (), factorial (), and fibonacci\_number (), demonstrate a substantial reduction in both time and cost associated with software testing. Notably, the total testing time is optimized to 1.59e-06 seconds. This approach contributes to streamlining testing procedures, enhancing software quality, and efficiently allocating resources, addressing the critical needs of modern software development and testing environments. It promises to benefit software developers, testers, and end-users, marking a valuable advancement in the realm of software testing. The other optimization techniques like branch and bound, game theory may also be applied for optimization and prioritization of the test cases which may play a vital role for efficient development of the software products which are useful for the software industries.

# 5. References

1. Rhmann, W., Zaidi, T., and Saxena, V. (2015). Use of genetic approach for test case prioritization from UML activity diagram. *International Journal of Computer Applications*, *115*(4). DOI:[10.5120/20137-2232](https://ui.adsabs.harvard.edu/link_gateway/2015IJCA..115d...8R/doi:10.5120/20137-2232).
2. Rhmann, W., Zaidi, T., and Saxena, V. (2016). Test cases minimization and prioritization based on requirement, coverage, risk factor and execution time. *British Journal of Mathematics and Computer Science*, *14*(1), 1-9. DOI: 10.9734/BJMCS/2016/23269.
3. Wasiur Rhmann, V. S. (2016). Generation of test cases from UML sequence diagram based on extenics theory. *British Journal of Mathematics and Computer Science*, *16*(1). DOI: 10.9734/BJMCS/2016/25374.
4. Rhmann, W., and Saxena, V. (2016). Optimized and prioritized test paths generation from UML activity diagram using firefly algorithm. *International Journal of Computer Applications*, *145*(6), 16-22. DOI: 10.5120/ijca2016910718.
5. Alkawaz, M. H., and Silvarajoo, A. (2019). “A survey on test case prioritization and optimization techniques in software regression testing”. In 2019IEEE7thConferenceonSystems*,* ProcessandControl *(*ICSPC*)* (pp. 59-64). IEEE. DOI: <https://doi.org/10.1109/ICSPC47137.2019.9068003>
6. Lu, C., Zhong, J., Xue, Y., Feng, L., and Zhang, J. (2019). Ant colony system with sorting-based local search for coverage-based test case prioritization. *IEEE Transactions on Reliability*, *69*(3), 1004-1020. **DOI:**[10.1109/TR.2019.2930358](https://doi.org/10.1109/TR.2019.2930358).
7. Huang, R., Sun, W., Chen, T. Y., Towey, D., Chen, J., Zong, W., and Zhou, Y. (2019). Abstract test case prioritization using repeated small-strength level-combination coverage. *IEEE Transactions on Reliability*, *69*(1), 349-372. **DOI:**[10.1109/TR.2019.2908068](https://doi.org/10.1109/TR.2019.2908068).
8. Bajaj, A., and Sangwan, O. P. (2019). A systematic literature review of test case prioritization using genetic algorithms. *IEEE Access*, *7*, 126355-126375. **DOI:**[10.1109/ACCESS.2019.2938260](https://doi.org/10.1109/ACCESS.2019.2938260).
9. Taneja, D., Singh, R., Singh, A., and Malik, H. (2020). “A Novel technique for test case minimization in object-oriented testing”. Procedia Computer Science, Vol. 167, (pp. 2221-2228). DOI: <https://doi.org/10.1016/j.procs.2020.03.274>.
10. Bajaj, A., and Sangwan, O. P. (2020). “Nature-inspired approaches to test suite minimization for regression testing”. In Computationalintelligencetechniquesandtheirapplicationstosoftwareengineeringproblems (pp. 99-110). CRC Press.
11. Mohapatra, S. K., Mishra, A. K., and Prasad, S. (2020). “Intelligent local search for test case minimization”. JournalofTheInstitutionofEngineers *(*India*):* SeriesB, Vol. 101, (pp. 585-595). DOI: <https://doi.org/10.1007/s40031-020-00480-7>.
12. Zhou, Z. Q., Liu, C., Chen, T. Y., Tse, T. H., and Susilo, W. (2020). Beating random test case prioritization. *IEEE Transactions on Reliability*, *70*(2), 654-675. **DOI:**[10.1109/TR.2020.2979815](https://doi.org/10.1109/TR.2020.2979815).
13. Dhareula, P., and Ganpati, A. (2020). Flower pollination algorithm for test case prioritization in regression testing. In *ICT Analysis and Applications: Proceedings of ICT4SD 2019, Volume 2* (pp. 155-167). Singapore: Springer Singapore. DOI: https://doi.org/10.1007/978-981-15-0630-7\_16.
14. Wang, R., Li, Z., Jiang, S., and Tao, C. (2020). Regression Test Case Prioritization Based on Fixed Size Candidate Set ART Algorithm. *International Journal of Software Engineering and Knowledge Engineering*, *30*(03), 291-320. DOI: <https://doi.org/10.1142/S0218194020500138>.
15. Chi, J., Qu, Y., Zheng, Q., Yang, Z., Jin, W., Cui, D., and Liu, T. (2020). Relation-based test case prioritization for regression testing. *Journal of Systems and Software*, *163*, 110539. DOI: <https://doi.org/10.1016/j.jss.2020.110539>.
16. Bajaj, A., and Sangwan, O. P. (2021). Discrete cuckoo search algorithms for test case prioritization. *Applied Soft Computing*, *110*, 107584. DOI: https://doi.org/10.1016/j.asoc.2021.107584.
17. Boyar, T., Oz, M., Oncu, E., and Aktas, M. S. (2021). A novel approach to test case prioritization for software regression tests. In *Computational Science and Its Applications–ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part VII 21* (pp. 201-216). Springer International Publishing. DOI: https://doi.org/10.1007/978-3-030-87007-2\_15.
18. Khatibsyarbini, M., Isa, M. A., Jawawi, D. N., Shafie, M. L. M., Wan-Kadir, W. M. N., Hamed, H. N. A., and Suffian, M. D. M. (2021). Trend Application of Machine Learning in Test Case Prioritization: A Review on Techniques. *IEEE Access*, *9*, 166262-166282. DOI: https://doi.org/10.1109/ACCESS.2021.3135508.
19. Huang, Y., Shu, T., and Ding, Z. (2021). A learn-to-rank method for model-based regression test case prioritization. *IEEE Access*, *9*, 16365-16382. DOI: https://doi.org/10.1109/ACCESS.2021.3053163.
20. Hasnain, M., Pasha, M. F., Ghani, I., and Jeong, S. R. (2021). Functional Requirement-Based Test Case Prioritization in Regression Testing: A Systematic Literature Review. *SN Computer Science*, *2*(6), 421. DOI: https://doi.org/10.1007/s42979-021-00821-3.
21. Laaber, C., Gall, H. C., and Leitner, P. (2021). Applying test case prioritization to software microbenchmarks. *Empirical Software Engineering*, *26*(6), 133. DOI: https://doi.org/10.1007/s10664-021-10037-x.
22. Raamesh, L., Jothi, S., and Radhika, S. (2022). “Test case minimization and prioritization for regression testing using SBLA-based adaboost convolutional neural network”. TheJournalofSupercomputing, Vol. 78(16), (pp. 18379-18403). DOI: <https://doi.org/10.1007/s11227-022-04540-1>.
23. Vescan, A., Pintea, C. M., and Pop, P. C. (2022). Test case prioritization—ANT algorithm with faults severity. *Logic Journal of the IGPL*, *30*(2), 277-288. DOI: https://doi.org/10.1093/jigpal/jzaa061.
24. Dahiya, O., Solanki, K., Rishi, R., Dalal, S., Dhankhar, A., and Singh, J. (2022). Comparative Performance Evaluation of TFC-SVM Approach for Regression Test Case Prioritization. In *Proceedings of International Conference on Communication and Artificial Intelligence: ICCAI 2021* (pp. 229-238). Singapore: Springer Nature Singapore. DOI: https://doi.org/10.1007/978-981-19-0976-4\_20.
25. Demir, Z. C., and Emrah Amrahov, Ş. (2022). Dominating set-based test prioritization algorithms for regression testing. *Soft Computing*, *26*(17), 8203-8220. DOI: https://doi.org/10.1007/s00500-022-07236-z.
26. Bajaj, A., Sangwan, O. P., and Abraham, A. (2022). Improved novel bat algorithm for test case prioritization and minimization. *Soft Computing*, *26*(22), 12393-12419. DOI: https://doi.org/10.1007/s00500-022-07121-9.
27. Iqbal, S., and Al-Azzoni, I. (2022). Test case prioritization for model transformations. *Journal of King Saud University-Computer and Information Sciences*, *34*(8), 6324-6338. DOI: https://doi.org/10.1016/j.jksuci.2021.08.011.
28. Khaleel, S. I., and Anan, R. (2023). A review paper: optimal test cases for regression testing using artificial intelligent techniques. *International Journal of Electrical and Computer Engineering (IJECE)*, *13*(2), 1803-1816. DOI: 10.11591/ijece.v13i2.pp1803-1816.