Automating Test Case Generation for Object-Oriented Testing Using the Hybrid Cuckoo and Simulated Annealing Algorithms.

A DISSERTATION PRESENTED

BY

MERCY C. NWABUEZE-NWOJI

TO

THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELOR OF COMPUTER SCIENCE

IN THE SUBJECT OF

SOFTWARE ENGINEERING

UNIVERSITY OF GREENWICH

GREENWICH, LONDON

APRIL 2024

Automating Test Case Generation for Object-Oriented Testing Using the Hybrid Cuckoo and Simulated Annealing Algorithms.

ABSTRACT

The objective of the research is to automatically generate test cases for Object-oriented programs.

Software testing is important, however, there are very few ways to test complex software. OOP is also very important, yet there are few ways to test these types of programs.

Hence, this research was focused on the development of a hybrid metaheuristic algorithm for generating test cases in object-oriented programming (OOP) environments. The approach integrated two metaheuristic algorithms, Cuckoo Search (CS) and Simulated Annealing (SA), to leverage their respective strengths in global exploration and local optimization.

The Cuckoo Search Algorithm (CS) was chosen for its ability to explore a broad solution space using Levy flights, while the Simulated Annealing Algorithm (SA) specialized in refining solutions within local regions. The integration of these algorithms aimed at enhancing the effectiveness and efficiency of test case generation by addressing both global exploration and local optimization needs.

The research outlined the design and implementation of a robust fitness function tailored to the challenges of testing OOP software, including code coverage, fault detection capability, execution efficiency, and adherence to OOP principles.

Additionally, the project utilizes the design of a user-friendly graphical user interface (GUI) for the hybrid metaheuristic algorithm. The GUI was designed to be accessible to both technical and non-technical users, with features such as ease of use, clear labeling, adherence to legal and security standards, and real-time feedback and reporting.

Overall, the research aimed to provide a comprehensive solution for test case generation in OOP environments by integrating metaheuristic algorithms with a user-friendly interface and robust fitness function.

The experimental results present a comprehensive evaluation of the hybrid CS-SA algorithm compared to other metaheuristic algorithms, including CS, SA, TLBO, and GA, in terms of average coverage, average best fitness, and average execution time for test case generation. Overall, the experimental results highlight the superiority of the hybrid CS-SA algorithm in terms of coverage, best fitness, and execution time compared to other metaheuristic algorithms, making it a promising approach for test case generation in object-oriented programming environments.

This research aimed to automate test case generation for object-oriented programs, integrating Cuckoo Search and Simulated Annealing algorithms. It designed a user-friendly interface and robust fitness function. Results showed superior performance, positioning the hybrid approach as promising for efficient and effective testing in OOP environments.

Contents

List of Tables

List of Figures

Words	Abbreviation
Object-oriented programming	**OOP**
Cuckoo search	**CS**
Simulated annealing	**SA**
Genetic algorithm	**GA**
Particle swarm optimization	**PSO**
Teaching learning-based optimization	**TLBO**
Firefly algorithm	**FA**
Hill climbing	**HC**
Differential evolution	**DE**
Graphical user interface	**GUI**
Software engineering	**SE**
Artificial Intelligence	**AI**
Machine Learning	**ML**

Acknowledgements

I want to express my thanks to Almighty God who in His infinite mercies has made it possible for me to embark on this academic journey. My heartfelt thanks to my family whose support and intuitive understanding have been a beacon of encouragement.

I also extend a special thanks to my tutors, John Ewer, and Cornelia Boldyreff, for their invaluable guidance and insight throughout this journey. Their support has been pivotal in shaping both this project and my growth.

To all, your contributions have been instrumental to my achievements. Thank you for being an integral part of my story.

1

Introduction

1.1 BACKGROUND

The development and execution of software are heavily reliant on the software testing process. As highlighted by Sneha, K. and Malle, G.M., (2017), this process involves the execution of software to find bugs or errors in the program. This means that the process determines the quality of the software being tested and identifies anomalies present in the software before it is released to the clients or the public. This is further backed up by Jha, N. and Popli, R. (2021) who also describe this process as a "critical phase of Software engineering (SE)." This indicates that the software testing process is important to the overall software development life cycle. Hence, the significance of identifying and creating an effective means of testing especially if there is an increase in software complexity.

Object-oriented programming (OOP) is a paradigm that is built on the idea of objects and how they can hold both code and data. Efan et al. (2023) highlight its importance as a systematic approach to encapsulation of objects, classes, attributes, and methods that mirror real-world interactions. Similarly, Raut R. S. (2020) suggests OOP's abilities in building systems that replicate real-world dependencies and hierarchies, while Ashwin Urdhwareshe (2016) notes how it focuses on object design in contrast to procedural programming. Shuai Wang et al. (2024) further highlights its popularity by revealing that four of the top five programming languages are built on the OOP paradigm, demonstrating the paradigm's widespread adoption and significance in the software development industry.

Despite the importance and advantages of OOP, it does not come without its challenges, especially in testing. The core components of OOP i.e. polymorphism, inheritance, encapsulation, and method overriding, present challenges. For instance, polymorphism causes behavioral changes to classes due to dynamic binding, or even Inheritance which can give rise to new faults due to an increase in the level of the hierarchy, as noted by M. Ghoreshi and Haghighi, (2023). These elements of OOP add a layer of complexity that can hide the state of objects and complicate the interrelationships within the program, making OOP more difficult to test than procedural programs. These complexities give rise to a need for a multi-objective testing approach that can navigate the complexities of OOP testing.

This project recognizes the shortcomings of existing OOP testing approaches and proposes the use of metaheuristic algorithms. These algorithms are renowned for their ability to mimic the evolutionary and biological behaviors of living beings. As Panda, M., et. al, (2020) suggest, these algorithms excel at learning and adapting to discovering the optimal solutions, making them suited to addressing the challenge that testing OOP presents. This paper proposes a hybrid approach that combines two metaheuristics algorithms i.e. the cuckoo search algorithm (CS) and the simulated annealing algorithm (SA) to balance their explorative and exploitative capabilities which are necessary for effective OOP testing. The following sections will delve deeper into the reasons for selecting these algorithms and the method to implementing the hybrid approach.

2

Literature Review

2.1 CURRENT CHALLENGES IN TESTING OOP

Due to features of OOP like polymorphism, inheritance, encapsulation, and method overriding, testing Object-Oriented Programming (OOP) systems presents a variety of challenges. These features, while making software development easier, make testing much more difficult. Polymorphism allows for varied behaviors from the same interface, raising challenges in predicting object behavior due to dynamic method binding. Inheritance and method overriding introduce complexities in tracking derived class behaviors and the potential for introducing new faults. Encapsulation, which hides object state and behavior, further complicates direct testing of internal states or methods.

These complexities are not addressed properly through traditional testing tools and procedures, such as evolutionary algorithms and manual testing. They frequently make mistakes when navigating the complex world of object-oriented programming (OOP), which can result in ineffective testing procedures, missed software abnormalities, and large resource expenditures without ensuring thorough test coverage. This situation highlights a significant flaw in the way testing is currently done; advanced techniques that can handle the complex dynamics of object-oriented programming are required.

Therefore, this body of research highlights the necessity of switching to automated test case generation capable of multi-objective optimization to balance important testing objectives including optimizing code coverage, reducing execution time, and improving fault detection in the context of object-oriented programming. Making this change is crucial to creating more complex, effective, and efficient testing procedures that are suited to the difficulties presented by OOP systems. More of this will be discussed in the appendix.

2.2 METAHEURISTIC ALGORITHMS IN SOFTWARE TESTING

This section of the paper presents a detailed analysis of existing metaheuristic algorithms applied to software testing, specifically object-oriented testing covering their strengths, weaknesses, and overall impact.

There have been several studies done on the use of search-based software testing techniques such as Genetic algorithms, Ant colony optimizations, particle swarm optimization, etc. Thus far, each of these algorithms has displayed different unique strengths and weaknesses. Therefore, this comprehensive analysis lays the foundation for this decision to develop a hybrid algorithm for Object-oriented testing.

Throughout the research, there is evidence to suggest that the use of genetic algorithms and swarm optimization techniques are currently the most popular methods for generating test cases using metaheuristic algorithms. According to Jha, N. and Popli, R. (2021), there has been substantial research for automated test case generation. From Clarke's (1976) invention of a model checker that automatically generates test cases to discussions of the use of AI and ML having the capabilities to automate software testing by Cico, O., et. al (2023).

There is more research from Robert Binder (1999) on testing Object-oriented programs, which has led to more investigation on testing using metaheuristic algorithms.

Khamprapai, W. et al. (2021) propose the use of an enhanced multiple search genetic algorithm (EMSGA) to generate test cases that were able to also detect errors in the software being tested using a small number of test cases while getting maximum coverage throughout the code. This algorithm was created using a modified version of the genetic algorithm but with some additional processes. The EMSGA has a more effective best chromosome selection process which makes it more effective than the traditional genetic algorithm for test case generation. The termination criteria of reaching a maximum number of generations and achieving a desired amount of branch coverage must be met for the process to end. The results, which were measured against other algorithms, show that EMSGA outperforms other algorithms in terms of branch coverage, mutation score, and fewer test cases. The highest branch coverage of 0.5900 was achieved by

EMSGA, with the highest mutation score of 0.4174 and achieving values better than the average number of test cases which was 179.19.

Potluri, S., et. al (2020) propose the use of 2 different hybrid algorithms, namely, the particle swarm bee colony (PSBCA) and the firefly cuckoo algorithms (FCSA) to generate test cases for model-based testing. This hybrid is created through the combination of 4 different metaheuristic algorithms i.e., the particle swarm optimization and bee colony optimization for the PSBCA, the firefly algorithm, and the cuckoo search algorithm for the FCSA. The results of running these algorithms on several test data shows that both PSBCA and FCSA outperform the firefly algorithm and the cuckoo search algorithm in terms of fitness function value, meaning that they perform better test cases that cover more paths and have less redundancy. The proposed algorithms also outperform the firefly algorithm and the cuckoo search algorithm in terms of iterations as they require fewer iterations meaning that they are faster and more efficient in terms of finding a solution. However, the PSBCA has the best performance amongst the 4 algorithms tested as it has the lowest number of iterations, while FCSA has more iterations, but not as many as the FA and CSA.

The approach of using a hybrid ant colony optimization and negative selection algorithm (HACO-NSA) was suggested by G. Kumar and V. Chopra, (2022). This hybrid aimed to generate test data that maximizes coverage and enhances test data efficiency. The use of metrics such as average time, average generation, and average coverage were used to evaluate the efficacy of the algorithm against other algorithms such as random testing, ant colony optimization, and negative selection algorithms. After comparing the results of the hybrid algorithm against the other algorithms using 11 benchmark problems it is safe to suggest without a doubt that the hybrid algorithm outperforms the other 3 algorithms. For example, HACO-NSA achieved the highest average coverage of 99.81%, the lowest average generation of 5.18, the lowest average time of 0.13 seconds, and the highest success rate of 100% on the Triangle Type program. Therefore, the algorithm the authors proposed was able to improve path coverage, reduce test data size, and enhance the efficiency of test data generated.

Another method of automatic testing was proposed by Khari, M. and Kumar, P. (2017) who suggested the use of the cuckoo search algorithm to optimize test suites. The authors aimed to generate and optimize test data used in software testing and reduce the number of test data and

time while achieving maximum path coverage and code coverage. This is done by using a sphere function to measure the fitness of each test data. The authors also simplified the cuckoo's behavior by choosing a random nest for each egg, keeping the nests with high-quality eggs, and having a fixed number of available nests. The algorithm was compared to other algorithms like hill climbing (HC) and firefly algorithm (FA) to see how well they would generate test suites for 50 JAVA programs. Results of this comparison show that the cuckoo algorithm takes less time to execute than the FA and HC, and the cuckoo algorithm requires fewer iterations than the FA and HC algorithm to reach optimal solutions.

Panda, M., et al. (2015) propose the use of a cuckoo search algorithm to generate test data for software testing. Comparisons were made against the performance of the CS and other metaheuristic algorithms. The criteria used to measure the effectiveness of the algorithms were execution times and code coverage. The result of the testing shows that the CS can generate test data for paths even after 50 iterations, unlike the other algorithms it is compared to, i.e., particle swarm and gravitational search algorithms.

A hybrid firefly and differential evolution algorithm to generate test suites for object-oriented programs was then suggested by authors Panda, M., et al. (2020). They proposed that combining the exploitative features of the FA with the explorative capabilities of DE would help optimize the algorithms' ability to generate test cases. The algorithm was tested against the benchmark triangle classification problem and its results were compared to the performance of the individual FA, DE, and other metaheuristic algorithms. The result shows that the proposed algorithm performed better than other algorithms in terms of execution time, test suite quality, and diversity.

D. Li et al. (2022) present an innovative approach to software testing through the hybrid SA-DynaMOSA algorithm, enhancing the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) with simulated annealing's exploration and exploitation strengths. This method, evaluated using the SF110 dataset, shows superior performance in search efficiency and coverage across multiple criteria compared to the Alternate Variable Method-DynaMOSA (AVM-DynaMOSA). The integration of multi-objective optimization with metaheuristic algorithms like SA offers a promising direction for advancing test case generation, particularly in complex object-oriented programming scenarios.

In this paper, the authors Panda, M., Dash, and Sujata. (2020), compare several algorithms and how they can be implemented to generate test cases based on UML models of object-oriented programs. The paper discusses the efficiency of seven metaheuristic algorithms namely: genetic algorithm (GA), differential evolution (DE), particle swarm optimization (PSO), artificial bee colony (ABC), cuckoo search algorithm (CS), firefly algorithm (FA), and gravitational search algorithm (GSA). Hybrid metaheuristics were also introduced to compare these algorithms, these hybrids being: a hybrid cuckoo search and simulated annealing algorithm (CS-SA), a hybrid firefly and differential evolution algorithm (FA-DE), and the hybrid particle swarm optimization and gravitational search algorithm (PSO-GSA). The algorithms were implemented using the MATLAB programming language and applied the algorithms to the benchmark triangle classification problem. The authors observe that the PSO-GSA algorithm outperforms the other algorithms in generating stable and uniform test data for all paths of the triangle classification problem.

The duration of these reviews was from 2015 to 2022. The most used criteria were the time needed for execution and path/branch coverage. Moreover, from the research presented it can be deduced that the most used metaheuristic algorithm is genetic algorithm. The results show there is a need to generate optimized test cases, decrease the time complexity, detect faults, and have path or branch coverage.

2.3 COMPARATIVE ANALYSIS

Here, the result of the research will be discussed critically to examine the relationship between existing metaheuristic algorithms and how they have been used to generate test cases, and how this can inform the proposed hybrid algorithm’s methodology.

In comparing the existing metaheuristic algorithms used in the generation of test cases, it is revealed that there is a diverse landscape of varying techniques that have varying effectiveness. According to the review above, the GA and PSO have been favored for their proficiency in navigating complex solution spaces. However, D. Li et al. (2022)'s hybridization of SA with DynaMOSA as seen in the review above, has led to an algorithm that eclipsed the

AVM-DynaMOSA in generating test cases, aligning with the trend toward the hybridization of metaheuristics.

Similarly, Kumar and Chopra (2022) were able to create the Hybrid Ant Colony Optimization and Negative Selection Algorithm (HACO-NSA) which has been proven to excel in average coverage, execution time, and generations compared to its non-hybrid counterparts. It appears to have been the most popular choice as they can explore complex solution spaces effectively. However, Khamprapai W. et. al. (2021) identified limitations in the GA's convergence speed, prompting the development of the Enhanced Multiple Search Genetic Algorithm (EMSGA). These algorithms' superior error detection and branch coverage efficacy establishes a foundation for combining the cuckoo search and simulated annealing algorithms, which are central to this research.

This, however, cannot be fully attributed to an individual cuckoo or simulated annealing algorithm as one soon discovers that single metaheuristic algorithms are not as good as their hybrid counterparts. One piece of this evidence apart from the one above comes from the observations of Panda et al. (2015) noting the cuckoo search algorithm (CS)'s exceptional performance over PSO and Gravitational Search Algorithm (GSA) in path coverage, positioning CS as potentially more aligned with the project's goals than the widely used PSO. Subsequent research by Panda, Madhumita & Dash, and Sujata (2020) further bolstered the case for hybrid algorithms; their hybrid PSO-GSA displayed remarkable stability and uniformity in test case generation, surpassing the capabilities of singular algorithms.

Potluri et al. (2020) take this study further by creating a hybrid of the PSO and BCO algorithms as well as a hybrid of the FA and CS comparing them against their component algorithms and demonstrating these hybrids' superiority in fitness function values and path coverage over single algorithms, advocating for their implementation in test case generation for software testing.

The Simulated Annealing (SA) algorithm, chosen for this research, is renowned for its proficiency in optimizing solutions within the local search space, addressing a crucial aspect of multi-objective optimization. D. Li et al. (2022) innovative hybridization of SA with DynaMOSA demonstrates a significant leap in enhancing test case generation by efficiently balancing multiple objectives, thereby outperforming the Alternate Variable

Method-DynaMOSA (AVM-DynaMOSA). This advancement aligns with the growing trend towards metaheuristic hybridization. Similarly, Kumar and Chopra (2022) illustrate the effectiveness of the Hybrid Ant Colony Optimization and Negative Selection Algorithm (HACO-NSA), displaying its superiority in managing multiple testing objectives, including average coverage, execution time, and generational efficiency, over traditional non-hybrid approaches. This research underscores the transformative potential of multi-objective optimization in software testing, highlighting the strategic integration of algorithms to achieve comprehensive and effective testing outcomes.

Therefore, the potential benefits of hybrid metaheuristic algorithms in object-oriented software testing are supported by this research. It demonstrates the tendency towards hybridization for improved performance and supports the strategy of creating a new hybrid algorithm that will combine the best features of several different algorithms to maximize automated test case generation. This is consistent with the continuous search for techniques that minimize execution time, enhance fault detection, and accomplish thorough path and branch coverage which are essential elements in the progression of object-oriented testing automation.

2.4 SUMMARY

Through a comprehensive examination of the current literature, the dynamic landscape of software testing is uncovered, emphasizing the application of metaheuristic algorithms for test case generation in object-oriented programming (OOP). These reviews delve into the unique challenges presented by OOP, including polymorphism, inheritance, encapsulation, and method overriding, and evaluate the responses of various metaheuristic algorithms, including genetic algorithms, particle swarm, and ant colony optimization. While these algorithms excel in exploring extensive solutions for generating optimal test cases, they encounter limitations when addressing the multi-objective nature of test case generation in OOP contexts. This multifaceted challenge requires balancing conflicting objectives, such as maximizing code coverage while minimizing resource consumption and execution time.

The shift towards employing hybrid metaheuristics, like the integration of firefly with differential evolution or the combination of particle swarm optimization with gravitational search, displays

superior performance in execution time, path coverage, and fault detection. These advancements suggest their potential to navigate the complexities of multi-objective optimization more effectively than their standalone counterparts.

These insights have directed the methodology of this project towards overcoming the identified limitations in test case generation. By merging the cuckoo search algorithm's broad explorative capabilities with the simulated annealing algorithm's focused efficiency, this project aspires to forge a novel path in test case generation that is both efficient and adept at handling the multi-objective demands of OOP testing.

These literature reviews lay a robust foundation for comprehending the evolution of test case generation techniques and identify critical gaps in the domain of software testing. Inspired by the reviews, the proposed hybrid algorithm endeavors to bridge these gaps, offering a significant contribution to the enhancement of software testing methodologies, particularly by addressing the inherent multi-objective optimization challenges in test case generation.

3

Aims and Objectives of the Research

3.1 IDENTIFYING LIMITATIONS IN OOP TESTING PRACTICE

Despite research on manual and automated test case generation using hybrid and single metaheuristic algorithms, combining cuckoo search with simulated annealing for optimization remains unexplored, especially in the context of OOP testing, and even though one instance of its use was uncovered via the literature review, it was not discussed in depth. This combination could prove to be novel due to its potential to leverage the distinct advantages of both algorithms, offering a suitable approach to tackling the complexities of OOP testing. This fits into the broader context of the project as it aims to bridge the gaps in the field of software testing by focusing on multi-objective optimization strategies for test case generation through a hybrid cuckoo and simulated annealing algorithm. This is the cornerstone of the research, utilizing metaheuristic algorithms that enhance the efficiency, effectiveness, and comprehensiveness of software testing. The objective of this study is outlined as follows:

1. Design and Implement a Hybrid Metaheuristic Algorithm: This will be done by combining the strengths of two metaheuristic algorithms in a way that incorporates multi-objective optimization principles to account for potential tradeoffs. This will not only take advantage of the complementary advantages of these two metaheuristic algorithms but will also effectively navigate the trade-offs between different testing objectives, thus enhancing the algorithm's overall robustness and performance in the context of OOP test case generation.
2. Evaluate the performance of the proposed hybrid algorithm in comparison with existing methods: A thorough comparison of the hybrid algorithm's performance to

current approaches will be part of the evaluation process, and multi-objective efficiency, accuracy, and computing costs will all be carefully considered. The evaluation will use metrics like code coverage, execution time, and best fitness scores to measure how well the hybrid algorithm produces high-quality test cases using a suite of open-source object-oriented programs. The evaluation will show how well the hybrid algorithm balances the optimization of different testing objectives in comparison to previous metaheuristic algorithms.

The study aims to demonstrate that the hybrid metaheuristic approach can significantly enhance OOP test case generation to achieve better code coverage and optimize execution metrics. This would not only meet the research goals but also suggests the potential for metaheuristic algorithms in software testing, paving the way for future explorations.

4

Planning and WBS

This section outlines the project's planning phase, providing a clear roadmap for the design, testing, and implementation of the hybrid algorithm. It includes timelines and risk mitigation strategies to ensure successful completion and quality. The project's scope is broken down into manageable segments, as detailed in the accompanying Gantt chart:

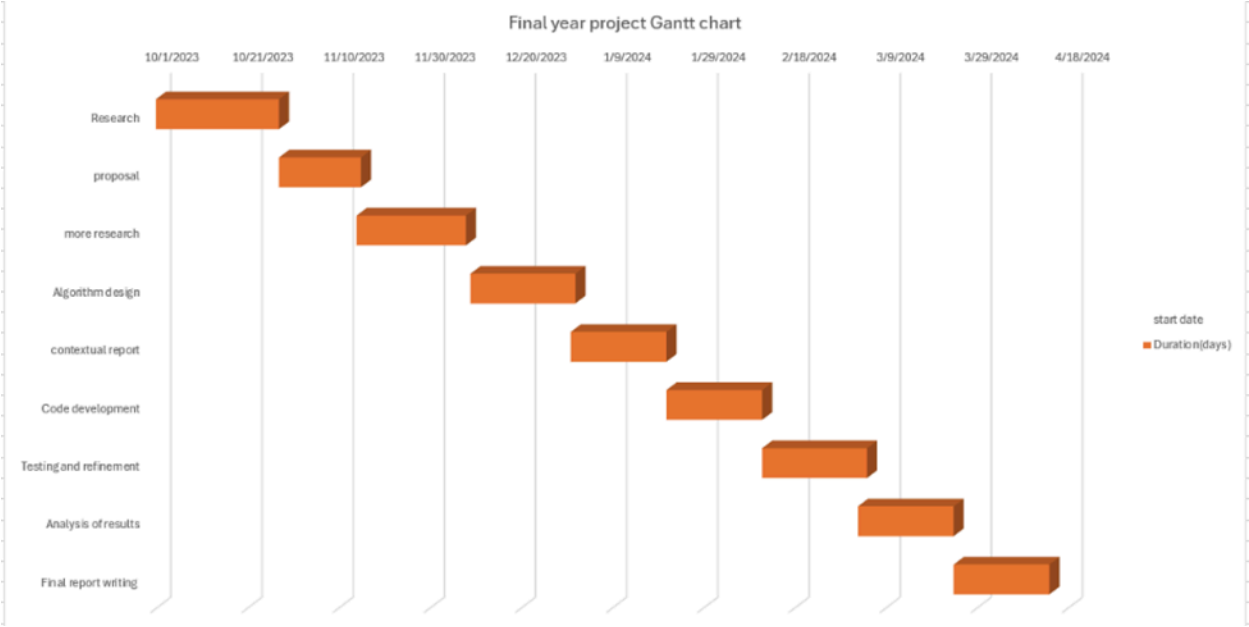


Figure 1: Gantt chart

4.1 TASKS AND JUSTIFICATION

1. The research was given the most days i.e., 27 days. Being the planning phase, extensive literature reviews were needed to understand what has been done

previously to enable the identification of research gaps, aiming to formulate a concrete problem statement and research objectives. This was vital for laying the groundwork for the project's direction and scope.

2. 18 days were assigned to writing the proposal as it involved outlining the research question, methodologies, and expected results.
3. After the proposal was approved, 24 days were allocated to researching more on this topic. This enabled the preliminary research for data and appropriate adjustments received from the proposal were attended to.
4. The algorithm design was then worked on for 23 days as it marked the transition into the tangible design of the proposed hybrid algorithm.
5. The contextual design report was written within 21 days. This was necessary as it included a comprehensive literature review and justification for the chosen algorithms.
6. The time allocated for the development of the code was also 21 days. This involved development of a user-friendly interface and this time was dedicated to the creation of the tool itself.
7. Testing and refinement took place within 23 days as it allowed for the validation and optimization of the algorithm through iterative testing and enhancements.
8. The results of the algorithms' performance were scrutinized for 21 days, and this was done so comparisons could be drawn with existing methods and interpreted.
9. The final report was the last stage of the report, and it took 21 days because it was important to discuss and document all the stages of the project giving a comprehensive report of the entire process. It serves as a summary of the entire work.

The planning phase was essential as it structured the development of the hybrid algorithm. This approach ensured the project met its objectives efficiently, ending with a final report summarizing the whole research and development process.

5

Requirements

This chapter highlights all the requirements, ethical considerations and risk analysis that were considered during the development of this product.

5.1 FUNCTIONAL REQUIREMENT

The functional requirements describe how the system is supposed to operate. In the Hybrid Test case generator's case, these requirements are crucial to ensure that the tool efficiently generates, manages, and analyzes test cases in terms of performance.

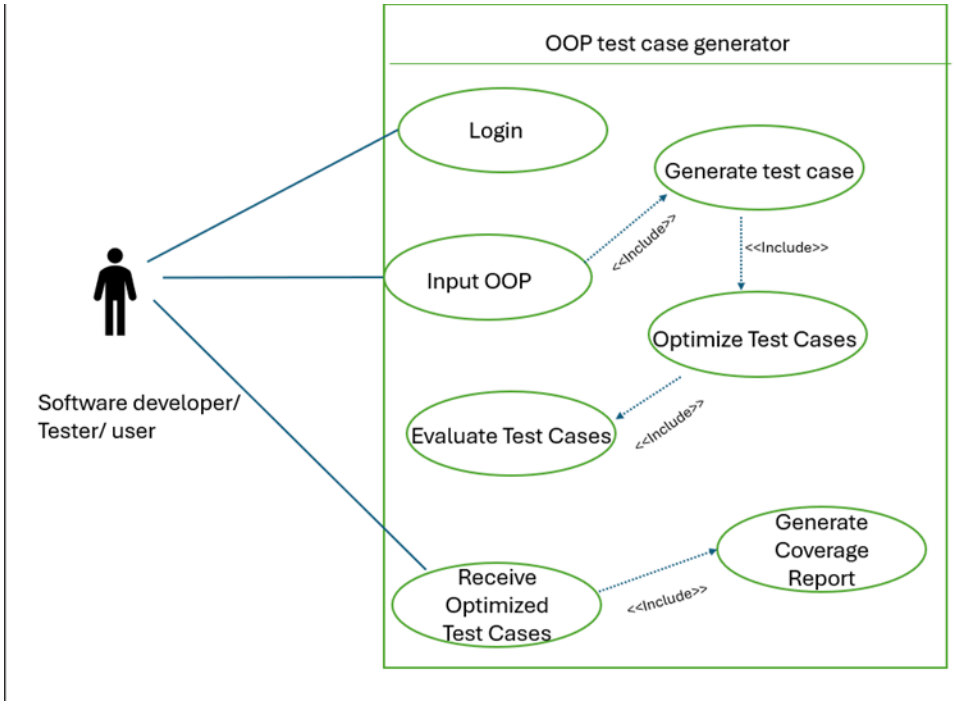


Figure 2:Use case diagram for the system.

The tool will allow both technical and non-technical users to:

1. Input specification: As shown in Fig 2, the User should be able to input OOP code, from which the system automatically generates optimized test cases. The test cases produced should ensure comprehensive coverage and efficient fault identification.
2. Automatically generate test cases: The tool should be able to automatically generate varied test cases based on user inputs and predetermined parameters, striving for 100% coverage alongside goals like reduced execution time and enhanced fault detection.
3. User-friendly: The interface will facilitate smooth navigation, test case generation, and management, providing guidance, templates, and a comprehensive coverage report suitable for all users. This ensures that generated test cases are seamlessly integrated into unit tests via the PyTest framework.
4. Reporting and analysis: The system output will include a comprehensive test suite covering all test scenarios, detailed reports, and analysis of the coverage rates, execution times, and best fitness score. This feature is crucial for the ongoing development and quality assurance of the OOP system which can then be used in the process of unit testing by the developer.

5.2 NON-FUNCTIONAL REQUIREMENTS

The Non-functional requirements, while not related to specific functionalities, are essential for ensuring the system's efficiency and user satisfaction.

1. Performance requirements: The generator should be able to perform efficiently under heavy loads with low execution times and high reliability. Hence, it must process inputs and generate test cases within a reasonable time limit and must make use of minimal computational resources without compromising the quality of generated test cases.

2. Usability constraints: the system requires that it should be easily navigated by the studio managers so that it does not delay their daily work. So, it needs to make the system easy to understand and reduce complexity, so that the time taken for the staff to master the use of the system is not too long. It should include a comprehensive help section to assist the user in utilizing the tool effectively.

3. Maintainability: The system should be designed for easy updates and maintenance, with a modular architecture that supports enhancements without disrupting existing functionalities.

4. Security constraints: The system needs strong security mechanisms such as a username and password in place to safeguard created test cases and sensitive source code. Verify adherence to data protection laws and that test results and user data are handled securely.

5.3 LESPI

In the creation of the hybrid metaheuristic test case generator, it's crucial to ensure that all testing is accurate and that faults are not missed, as undetected issues can lead to significant problems. Throughout the project, the Legal, Ethical, Social, and Professional (LESP) considerations deeply influence decision-making and the project's trajectory to ensure these risks are mitigated.

Legally, the project should adhere to relevant regulations and guidelines, such as those set out by the ACM/IEEE Code of Ethics and ISO/IEC/IEEE 29119 standards, to ensure the tool is responsible and liabilities are minimized. This legal adherence safeguards the project against potential breaches and reinforces the integrity of our testing tool.

Ethically, AL-Fedaghi (2018) highlights the intertwined nature of ethical and legal considerations, highlighting the importance of data privacy, responsible AI usage, and intellectual property rights. This project commits to ethical transparency, ensuring test case

generation respects user confidentiality and adheres to the principles of inclusive and secure software development.

Socially, the project acknowledges the role of technology in society, aiming for clear communication about the tool's capabilities to avoid misperceptions about automated testing's efficacy (StudySmarter UK). This approach fosters an understanding and acceptance of automated testing's role and limits within the broader tech ecosystem.

Professionally, the project is committed to respecting intellectual property by utilizing open-source code and licensed software appropriately, promoting innovation while remaining legally compliant. Incorporating LESPI considerations ensures the decisions made throughout the project align with societal values and professional standards, ensuring the hybrid metaheuristic test case generator is not only technically sound but also ethically and socially responsible.

5.4 RISK ANALYSIS

Risk	Type	Probability	Severity	Exposure	Priority	Mitigation
Data loss	Technical	Low	Catastrophic	Medium	Medium	1. Regular backup 2. Version control to track systems
Health/personal issues	People	Medium	Catastrophic	High	High	1. Inform instructors of deadline extensions
Improper management of tasks	Operational	Medium	Insignificant	Low	Low	1. Implement project management tools like Gantt chart
Literature analysis difficulties	People	Medium	Tolerable	Medium	Medium	1. Access to academic database 2. Training on research method
Improper integration of modules	Technical	High	Serious	Very high	High	1. Code reviews 2. Continuous integration systems
Unclear requirement issues	Operational	High	Serious	Very high	High	1. Clear communication 2. Regular meetings
Plagiarism/ academic misconduct	Ethical	Medium	Catastrophic	High	High	1. Plagiarism detection tools 2. Cite all sources to avoid plagiarism

Figure 3:Risk assessment.

An organized method to manage potential failures in the project is provided by the risks and mitigation strategies shown in the above table. How valid the project is depends on how catastrophic risks like "Data loss," "Health/personal issues," and "Plagiarism/academic misconduct" are mitigated. "Data loss" forces strict backup and security procedures because it compromises the core of the project. Mental health support is an important preventive measure since "health/personal issues" have a direct impact on team productivity. To prevent endangering the project's reputation, "plagiarism/academic misconduct" calls for the strict use of citation guidelines and copyright verification technologies.

Even though "Improper management of tasks" and "Literature analysis difficulties" are categorized under insignificant and tolerable severity, respectively, they can potentially cause disruptions to project flow and timetables, illustrating the importance of research training and strong project management systems.

Because of the serious nature of risks like "Improper integration of modules" and "Unclear requirements issues" that affect the functionality and goals of the project, there is a need for immediate action. These require transparent stakeholder communications and in-depth code reviews.

Early planning alone is not enough to mitigate these risks; constant watchfulness and frequent risk re-evaluation are necessary to adjust for project modifications. By taking the initiative, the project is guaranteed to adhere to ethical, legal, and professional standards in addition to staying on course.

In conclusion, this chapter effectively outlines the requirements, ethical guidelines and risk analysis for the developments of the test case generator. By detailing system operations, performance expectations, and usability, the project is set to meet high standards of efficiency and security. Incorporating legal, ethical, social, and professional considerations ensures compliance and ethical responsibility. A robust risk management plan further secures the project's successful execution and integrity.

6

Research Methodology

6.1 APPROACH TO DESIGN

This chapter aims to craft the approach of the design, using two metaheuristic algorithms which will be analyzed based on their advantages and limitations i.e. the Cuckoo search (CS) and the simulated annealing algorithm (SA).

6.1.1 CUCKOO ALGORITHM

The Cuckoo Search Algorithm (CS) is known for its exploratory capabilities, attributed to how it utilizes Levy flights to enable the algorithm to take large and random steps through the solution space to find the best solution needed. This mechanism mirrors the biological behavior of cuckoo birds, as described by Rajabioun, R., (2011), where cuckoos lay eggs in the nests of other species, thus exploring a wide range of potential solutions. Initially, the algorithm generates a diverse population of potential solutions (nests), where the discovery of cuckoo eggs by the host birds, determined by a probability 'Pa', symbolizes the iterative refinement of solutions—discarding the less fit ones and introducing new solutions via Levy flights.

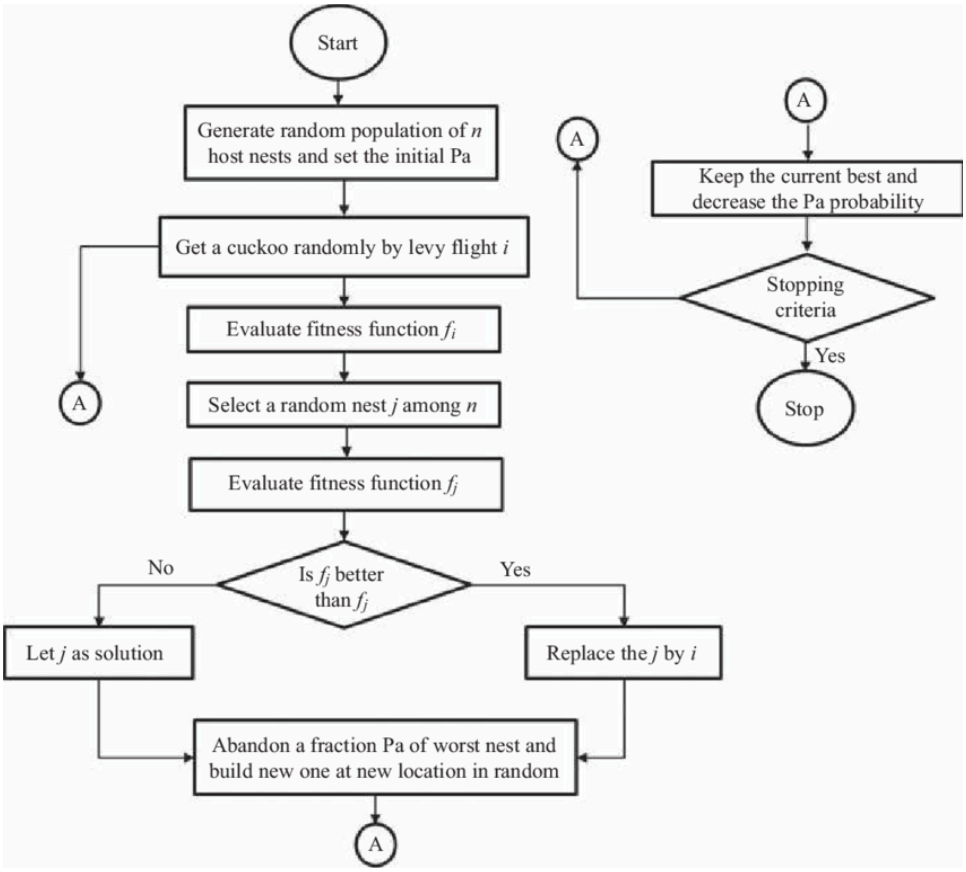


Figure 4:flowchart of the CS

The algorithm assesses the quality of solutions using a fitness function, ensuring the retention of only superior solutions. This iterative process, regulated by the probability 'Pa', facilitates extensive exploration, and maintains diversity within the solution space, thus averting premature convergence on local optima. The algorithm concludes once predefined criteria, such as a specific number of generations or a quality threshold for solutions, are met, guaranteeing the identification of an optimal solution.

Strengths of the Cuckoo Search Algorithm, as highlighted by Liu and Zhang (2021), include:

- Global Search Capability: Leveraging Levy flights, the CSA excels in exploring a broad solution space, unearthing diverse potential solutions.
- Adaptability: With minimal control parameters and an uncomplicated execution process, CS demonstrates versatility across various optimization challenges.

- Avoidance of Local Optima: Its unique mechanism to discard less fit solutions in favor of introducing new ones aids in avoiding local optima.

However, Liu and Zhang (2021), also describe its limitations as being:

- Balance Between Global and Local Search: Despite its prowess in global exploration, CS may falter in refining solutions within local search spaces, potentially hampering optimization efforts.
- Convergence Speed and Precision: The algorithm's expansive search strategy can sometimes result in slower convergence rates and diminished precision in pinpointing the most optimal solution.

Therefore, in combining the CS within a hybrid framework, the project aims to design an algorithm that fuses the extensive exploratory strength with the Simulated Annealing algorithm's meticulous local search capabilities in order to tackle the issue of balancing between global search and local search, in turn optimizing the solutions. Hence, the integration would aim to amplify the overall effectiveness and efficiency of the test case generation process, particularly within the complex testing field of OOP environments, addressing both global exploration and local optimization needs as highlighted by Panda et al. (2015) and Khari and Kumar (2017). This strategic hybridization seeks to overcome the inherent limitations of standalone metaheuristic algorithms by leveraging their collective strengths for superior test case generation.

6.1.2 SIMULATED ANNEALING ALGORITHM.

The Simulated Annealing Algorithm (SA) excels in the optimization of solutions within local regions, a process described as exploitation by Khari and Kumar (2017). This approach ensures the algorithm's focus on refining solutions to identify the most optimal outcomes within a constrained solution space, closely mimicking the physical process of annealing in metallurgy where controlled cooling improves the structure of a material.

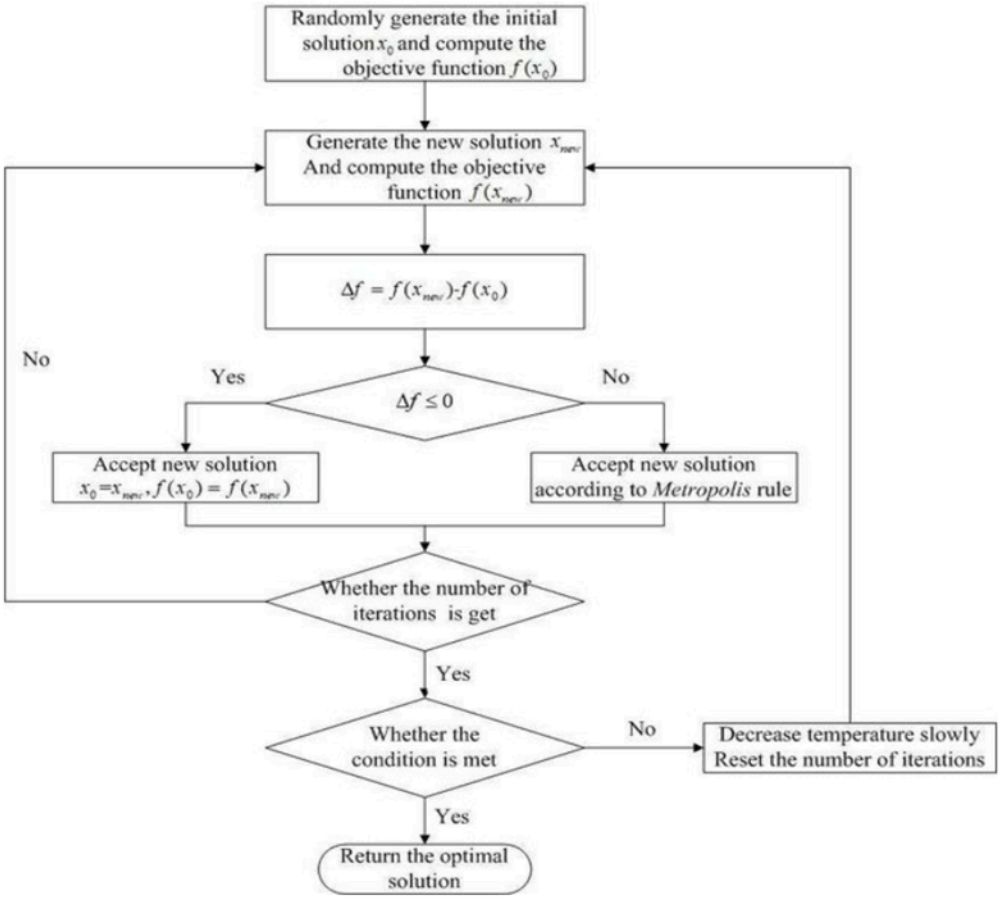


Figure 5: flowchart of SA

As shown in Figure 5, the algorithm starts by randomly generating an initial solution, denoted as $x_{0,}$ and computing its objective function $f(x_0)$. The objective function evaluates how good the solution is. It then generates a new solution (x_{new}) and computes the objective function for this new solution $f(x_{new})$.

The change in the objective function Δf is calculated by subtracting the objective function of the current solution from that of the new solution,
$f(x_{new}) - f(x_0)$.

The flowchart then shows that if Δf is less than or equal to 0, it means the new solution is as good as or better than the current one. In this case, the new solution is accepted, x_0 is set to new, and their objective functions are equated.

If Δf is greater than 0, the new solution is worse than the current one. In this case, the new solution may still be accepted but according to the Metropolis rule, which typically involves a probability function that depends on Δf and a control parameter called "temperature". This rule allows the algorithm to occasionally accept worse solutions to escape local optima.

The algorithm iterates through solution generation and evaluation, adjusting the temperature according to a predetermined schedule to gradually decrease the acceptance probability of worse solutions. This allows for a fine-tuning phase, converging on the most optimal solution found when the termination criteria, such as a specified quality of solution or iteration count, are met.

Strengths of the Simulated Annealing Algorithm as highlighted by Alkhateeb and Abed-alguni (2019), include:

- Local Optimization: Specializes in intensive search within local solution spaces to refine solutions to near optimality.
- Escape from Local Optima: Through the probabilistic acceptance of worse solutions, SA can navigate out of local optima, ensuring a more comprehensive search for the global optimum.

Alkhateeb and Abed-alguni (2019), also discuss the Limitations of the algorithm being:

- Convergence Speed: Depending on the cooling schedule and the complexity of the solution space, SA might exhibit slower convergence to the optimal solution.
- Dependence on Temperature Parameter: The efficiency of SA heavily relies on the temperature control parameter, which requires careful calibration to balance between exploration and exploitation effectively.

By integrating the Simulated Annealing Algorithm with the Cuckoo Search Algorithm into a cohesive hybrid framework, this research endeavors to harness SA's capability for intensive local search and optimization alongside CS's global exploratory strength. This hybridization targets a comprehensive optimization approach that effectively addresses the multifaceted challenges of

generating test cases for object-oriented software, aiming to synergize CS's broad search capabilities with SA's precision in solution refinement. This combination is projected to significantly enhance the generation of test cases by ensuring broad coverage of the solution space while meticulously optimizing each potential solution for the intricate testing demands of object-oriented programming systems, as emphasized by the objectives outlined in this study.

6.1.3 FITNESS FUNCTION

The core of the hybrid algorithm's efficacy lies in the design and implementation of a robust fitness function. In their work Panda., M. et al. (2015) describe this process as capturing vital information and guiding the process of searching for an optimal solution. This claim is backed up by Khari, M., and Kumar, P., (2017) as they describe it as being able to find the optimal solution for a given problem.

Hence, to be able to navigate the complexities of testing object-oriented software, particularly when addressing challenges like polymorphism, inheritance, method overriding, and encapsulation, the design of an effective fitness function is paramount. This fitness function serves as the foundational element guiding the hybrid metaheuristic algorithm—combining aspects of the CS and SA—to evaluate and select optimal test cases.

In the context of the hybrid metaheuristic algorithm, the fitness function would be designed to assess test cases based on a set of objective criteria directly aligned with the testing challenges identified:

1. Code Coverage: Measures the extent to which the test cases cover the software code, including all paths (branch coverage), executed lines of code (line coverage), and executed conditions within decision points (condition coverage). High code coverage of 80% or more is indicative of a test case's thoroughness.
2. Fault Detection Capability: Evaluate the ability of a test case to identify and highlight faults within the software. This includes detecting unintended behaviors resulting from polymorphism, inheritance anomalies, incorrect method overriding, and breaches in encapsulation principles.

3. Execution Efficiency: Considers the computational resources and time required to execute the test cases. Optimal test cases should not only be effective in uncovering software anomalies but also efficient in terms of execution time and resource usage.
4. Adherence to OOP Principles: Specifically evaluates how well the test cases address the unique challenges posed by OOP concepts. This involves assessing the integrity of polymorphic behaviors, the correctness of inheritance and method overriding, and the robustness of encapsulation practices.

6.1.4 GUI DESIGN

The hybrid metaheuristic algorithm test case generator's graphical user interface (GUI) was designed to facilitate access to advanced testing approaches based on both functional and non-functional requirements. Key details consist of:

1. Ease of Use and Accessibility: The GUI makes the system easy to use for both technical and non-technical users by bridging the complexity of the algorithm with a simple interface. This method guarantees that users won't become sidetracked by the underlying complexity and can concentrate on creating and saving test cases.

2. User-Centric Design: The GUI makes it easy to navigate and interact with sections that are labeled clearly and with a clean structure. Easy access to tool guides and test templates ensures a smooth user experience by helping users at every stage.

3. Adherence to Legal, Ethical, and Security Standards: Copyright laws and privacy rules are given priority in the design. In addition to authentication mechanisms to secure system access, it includes strong security features including data encryption and secure session management to safeguard users' code and test cases.

4. Real-Time Feedback and Reporting: The GUI provides thorough reports on coverage and efficacy as well as in-depth feedback on the test case creation. This function offers practical insights into areas for software enhancement, which helps with the ongoing development and improvement of the program.

In summary, the chapter outlines the combined use of the CS and SA algorithms to improve test case generation in OOP testing. The integration aims to utilize the global search capabilities of the CS with the local optimization strengths of SA. It also utilizes a GUI to streamline user-tool interaction while maintaining the highest levels of usability, security, and ethical conduct. It hopes to positively impact software testing practices.

7

Algorithm Implementation

7.1 DEVELOPING THE HYBRID ALGORITHM.

This chapter highlights the techniques used to implement the combined CS and SA algorithm into a hybrid test case generator. It details the libraries used, how the algorithms were combined, how the OOP concepts were evaluated and how faults were detected.

7.1.1 IMPLEMENTATION OF LANGUAGE AND TOOLS

As stated in this paper's approach, the Python programming language was the primary tool for this algorithm's implementation. This is due to the versatility and extensive libraries the programming language had to offer as well as its ease of use which made it an ideal choice for executing the proposed methodologies. Hence, the libraries and methods of implementing the algorithm will be discussed in the following sections.

7.1.2 LIBRARIES USED IN IMPLEMENTATION

The libraries used were the random library, math library, SciPy.Special, NumPy, Ast, and time libraries as shown in Fig 6.

```
import random
import math
from scipy.special import gamma
import numpy as np
import ast
import time
```

Figure 6:Imported Libraries

7.1.3 RANDOM LIBRARY

Renowned for its capabilities in generating random numbers, the random library was used in the implementation of random processes within the algorithm. For instance, it provided the means to introduce randomness in the CS's step size and the SA's neighbor selection process, this is crucial for navigating the solution space and avoiding local optima.

7.1.4 MATH LIBRARY

```
1 usage
def acceptance_probability(self, old_cost, new_cost):
    # Calculate the acceptance probability
    if new_cost < old_cost:
        return 1.0
    else:
        return math.exp((old_cost - new_cost) / self.temperature)
```

Figure 7:Example usage of the math Library in the SA algorithm

To perform mathematical computations needed for optimizing the algorithm, the math library was employed. It includes functions that calculate exponentials, logarithms, and other common mathematical operations, which are essential in evaluating acceptance probabilities (Fig 7) in SA which is to be used in the optimization process in the hybrid algorithm.

7.1.5 SCIPY. SPECIAL LIBRARY

```
def levy_flight(self):
    sigma = (gamma(1 + self.beta) * np.sin(np.pi * self.beta / 2) / (
                gamma((1 + self.beta) / 2) * self.beta * 2 ** ((self.beta - 1) / 2))) ** (1 / self.beta)
    u = np.random.normal( loc: 0, sigma, size=self.nest_size)
    v = np.random.normal( loc: 0,  scale: 1, size=self.nest_size)
    step = u / abs(v) ** (1 / self.beta)
    return step
```

Figure 8:levy flight implementation

This library, known for its collection of mathematical algorithms and convenience functions, was used for its specialized gamma function. As shown in the algorithm's implementation in Fig 8, the gamma function is part of the mechanism to calculate the step sizes in Lévy flights, enabling the CS algorithm to perform global searches with heavy-tailed step size distributions.

7.1.6 NUMPY LIBRARY

The NumPy library, renowned for its array operations, was essential for managing and manipulating large multi-dimensional arrays and matrices. It was utilized in the implementation process to store and process populations of solutions efficiently, facilitating vectorized computations that are much faster than their pure Python counterparts.

7.1.7 AST LIBRARY

This library is typically used for processing trees of the Python abstract syntax grammar. However, during the implementation process, it extracts and parses the given source code into an abstract syntax tree as shown in Fig 9.

```python
class ProgramAnalysis:
    def __init__(self, program_code):
        self.program_code = program_code
        self.class_inheritance = {}  # Store inheritance relationships
        self.class_methods = {}  # Store methods and their parameters for each class
        self.coverage = set()  # To track covered methods

    3 usages
    def extract_structure(self):
        try:
            tree = ast.parse(self.program_code)
            self._parse_ast(tree)
        except SyntaxError as e:
            raise ValueError(f"Error parsing the Python code: {e}")
```

Figure 9:Extraction function for analysis by the AST Function.

This enables the algorithm to analyze and generate test cases based on the structural properties of the code, reflecting a direct manipulation of the code's syntactic elements and, hence, having a significant effect on the functionality of our program analysis component as illustrated in Fig 10.

```
1 usage
def _parse_ast(self, node):
    for item in ast.walk(node):
        if isinstance(item, ast.ClassDef):
            self._process_class_definition(item)

1 usage
def _process_class_definition(self, class_node):
    class_name = class_node.name
    self.class_inheritance[class_name] = [base.id for base in class_node.bases if isinstance(base, ast.Name)]
    self.class_methods[class_name] = {method.name: self._parse_method_parameters(method)
                                      for method in class_node.body if isinstance(method, ast.FunctionDef)}

1 usage
def _parse_method_parameters(self, method_node):
    return {arg.arg: self._get_default_value(method_node, arg) for arg in method_node.args.args}
```

```
1 usage
def _get_default_value(self, function_node, arg):
    defaults_index = len(function_node.args.args) - len(function_node.args.defaults)
    arg_index = function_node.args.args.index(arg)
    if arg_index >= defaults_index:
        return repr(function_node.args.defaults[arg_index - defaults_index])
    return None
```

Figure 10:The analysis of the source code by the AST function

7.1.8 TIME

Finally, the time library was used to record the durations of algorithm execution and to manage pauses in execution when necessary. In the context of the hybrid algorithm as illustrated in Fig 11, it provides a means to evaluate the performance of the optimization algorithm in terms of speed and efficiency, which is pivotal in assessing the overall effectiveness of the algorithm.

```
start_time = time.perf_counter()
analysis = ProgramAnalysis(program_code)
analysis.extract_structure()
analysis.identify_test_scenarios()
hybrid_tool = HybridAlgorithm(user_input_code=program_code)
test_cases = hybrid_tool.generate_test_cases(program_code) # Generate test cases
best_fitness_score = float('inf')
best_test_case = None
for test_case in test_cases:
    current_fitness_score = hybrid_tool.cuckoo.get_fitness(test_case)
    print(f"Test Case: {test_case},\n Fitness Score: {current_fitness_score}")
    if current_fitness_score < best_fitness_score:
        best_fitness_score = current_fitness_score
        best_test_case = test_case
end_time = time.perf_counter()
execution_time = end_time - start_time
print(f"Best Fitness Score: {best_fitness_score}")
print(f"Best Test Case: {best_test_case}")
print(f"The time taken to generate test cases was {execution_time}")
```

Figure 11:execution time of the algorithm's process.

7.2 IMPLEMENTATION OF CUCKOO AND SIMULATED ANNEALING AS A HYBRID

As stated in several sections of this report, the implementation of the CS and SA as hybrids combines the explorative strengths of the CS with the exploitative features of SA. The CS initiates the process by exploring the solution space through Lévy flights which is illustrated in Fig 8. The levy flight which is a random walk process ensures that there are jumps in the diverse search space. This helps to prevent the risk of premature convergence on local optima by encouraging the exploration of new and potentially more promising regions of the solution space.

```
def update_nests(self):
    for nest in self.nests:
        step_size = self.levy_flight()
        new_nest = nest + step_size * np.random.rand(*nest.shape)
        new_fitness = self.get_fitness(new_nest)
        if new_fitness < self.get_fitness(nest):
            nest[:] = new_nest
    # abandon worst nests or worst solutions
    for nest in self.nests:
        if np.random.rand() < self.pa:
            nest[:] = np.random.rand(*nest.shape)

5 usages (4 dynamic)
def find_best_solution(self):
    for nest in self.nests:
        fitness = self.get_fitness(nest)
        if fitness < self.best_fitness:
            self.best_fitness = fitness
            self.best_nest = nest
    return self.best_nest, self.best_fitness
```

Figure 12: The best solution being found

Once the CS has found its best solutions, the SA takes over the process to ensure exploitation takes place. This is done by fine-tuning the solutions which were initially found in the CS as illustrated in Fig 13. The SA process allows for an occasional uphill move to avoid the local optimum in pursuit of the global optimum.

```
    # Further optimization with Simulated Annealing
    self.annealing.initial_solution(self.cuckoo.best_nest)
    optimized_solution = self.annealing.anneal(self.cuckoo.get_fitness)

    convergence_generation = last_improvement_generation
    return optimized_solution, best_fitness_per_generation, convergence_generation
```

Figure 13: Fine-tuning of the initial solutions by SA

The temperature parameter in Simulated Annealing is then set to gradually decrease as shown in Fig14. This ensures the reduction of the probability of accepting worse solutions over time, hence converging to an optimal solution.

```
1 usage
def anneal(self, fitness_function):
    current_solution = self.current_solution
    current_cost = fitness_function(current_solution)

    while self.temperature > 1:
        new_solution = self.get_neighbour(current_solution)
        new_cost = fitness_function(new_solution)

        if self.acceptance_probability(current_cost, new_cost) > random.random():
            current_solution = new_solution
            current_cost = new_cost

        self.temperature *= 1 - self.cooling_rate

    return current_solution
```

Figure 14:The annealing process gradually decreases the initial temperature of the SA process

This hybrid approach harnesses the strengths of both algorithms, employing CS's global search capability to locate promising areas of the search space and SA's local search capability to refine the solutions within those areas. The combination results in a robust optimization framework that is both wide-ranging in its search and precise in its solution refinement.

7.3 IMPLEMENTATION OF THE FITNESS FUNCTION

Being implemented in Python, the fitness function serves as the backbone for any optimization algorithm, providing a measure of how well a solution meets the defined objectives. In the context of our hybrid metaheuristic algorithm, which merges the CS and SA techniques, the

fitness function is pivotal for evaluating the effectiveness of generated test cases against the multifaceted requirements of OOP testing.

```python
def fitness(self, test_case):
    min_fitness_score = 0.001  # Set a minimum threshold for the fitness score
    base_score = 0.5
    # Check for missing keys and assign default values if necessary
    classes = test_case.get('classes', [])
    subclass = test_case.get('subclass', None)
    superclass = test_case.get('superclass', None)
    class_reference = test_case.get('class_reference', None)
    # Evaluate positive aspects of the test case
    polymorphism_score = self.evaluate_polymorphism(classes, test_case['method_name'])
    method_overriding_score = self.evaluate_method_overriding(subclass, superclass, test_case['method_name'])
    encapsulation_score = self.evaluate_encapsulation(class_reference) * 0.5
    inheritance_score = self.evaluate_inheritance(class_reference) * 0.5
    # Sum positive aspects to form the base evaluation score
    evaluation_score = polymorphism_score + method_overriding_score + encapsulation_score + inheritance_score
    # Evaluate potential faults and reduce the evaluation score accordingly
    fault_penalty = self.evaluate_faults(test_case)
    random_variation = random.uniform(-0.5, b: 0.5)  # Adjust the range as needed to create impact of randomness
    evaluation_score += random_variation
    # Ensure a minimum fitness score even in the presence of faults
    final_score = max(base_score - evaluation_score + fault_penalty, min_fitness_score)
    return final_score
```

Figure 15:The overall implementation of the fitness function

The fitness function, discussed in the design and approach section of the text, has been implemented in our code to quantitatively assess the quality of the test cases by evaluating the various aspects of OOP. This ranges from evaluating the extent to which test cases exploit polymorphism, to the adherence of encapsulation, correctly implementing method overriding, etc.

7.3.1 EVALUATING POLYMORPHISM

```python
def evaluate_polymorphism(self, classes, method_name):
    behaviors = set()
    for cls in classes:
        try:
            instance = cls()
            behavior = getattr(instance, method_name)()
            behaviors.add(behavior)
        except AttributeError:
            continue  # Skip if method not implemented
    # Score based on the number of unique behaviors
    polymorphism_score = len(behaviors) * 10  # Assigning a higher score for greater diversity
    return max(polymorphism_score, 1)  # Ensure a minimal score for the attempt
```

Figure 16:Polymorphism fitness function

Here, polymorphism is evaluated by the unique behaviors obtained from a specific method across different classes. The evaluate_polymorphism function, as shown in Figure16, captures the essence of this OOP concept, ensuring methods across classes exhibit diverse implementations.

The function operates as follows:

- A set called behaviors is created to keep track of the unique behaviors exhibited by the method_name.
- The function iterates over a list of classes, attempting to instantiate each class and call the method_name on the instance.
- If the method exists and executes without error, the behavior is added to the behaviors set. This captures the distinct output or behavior from each class.
- If the method doesn't exist (AttributeError is raised), the class is skipped.
- The polymorphism_score is calculated based on the number of unique behaviors observed, multiplied by 10, rewarding a greater diversity of behaviors.
- The function ensures there's a minimum score of 1, even if no diverse behaviors are detected, to recognize the attempt to implement polymorphism.

By integrating evaluate_polymorphism into the primary fitness function depicted in Figure 15, the algorithm is tuned to prefer test cases displaying effective polymorphism, aligning with our multi-objective OOP testing approach.

7.3.2 EVALUATING ENCAPSULATION

In the case of Encapsulation, the fitness function assesses this concept to determine how well a class in object-oriented programming hides its internal state and behavior from the outside world, which is a key principle of encapsulation. This is done through the evaluate_encapsulation function, as illustrated in Fig 17.

```
1 usage
def evaluate_encapsulation(self, cls_reference):
    encapsulation_score = 0
    attributes = [attr for attr in dir(cls_reference) if attr.startswith('_') and not attr.startswith('__')]
    for attr in attributes:
        getter = f'get{attr[1:].capitalize()}'
        setter = f'set{attr[1:].capitalize()}'
        if hasattr(cls_reference, getter) and hasattr(cls_reference, setter):
            encapsulation_score += 5  # Encourage encapsulation practices
    return max(encapsulation_score, 1)  # Minimum score for attempt
```

Figure 17:Encapsulation fitness function.

Here's a breakdown of how the evaluate_encapsulation function works:

- It initializes an encapsulation_score with a value of zero.
- It then creates a list of attributes by inspecting the cls_reference class for all attributes that start with a single underscore (indicating they are intended to be protected or private) but not a double underscore (which usually denotes special Python methods or name mangled attributes).
- For each attribute, the function dynamically constructs the names of a getter and setter method using string formatting. The convention assumes that getter and setter methods for an attribute are named get<AttributeName> and set<AttributeName>, respectively, with <AttributeName> being the capitalized version of the actual attribute name (without the leading underscore).
- If both a getter and a setter method exist for an attribute, the encapsulation_score is incremented by 5 to reward the class for following good encapsulation practices.

- Finally, the function ensures a minimum score of 1 to acknowledge any attempt at encapsulation, even if no getter and setter methods are found.

This approach to evaluating encapsulation in the fitness function is significant because it encourages and rewards the proper implementation of encapsulation, which contributes to the maintainability and robustness of the object-oriented code. The higher the encapsulation_score, the better the test case aligns with the OOP principle of encapsulation, ensuring that state and behavior modification are controlled through well-defined interfaces.

7.3.3 EVALUATING INHERITANCE

With Inheritance, the evaluate_inheritance function as shown in Fig 18, is created with a focus on assessing how well inheritance is implemented in a class.

```
def evaluate_inheritance(self, cls_reference):
    def inheritance_depth(cls):
        if not cls:
            return 0
        if cls.__bases__:
            return 1 + max(inheritance_depth(base) for base in cls.__bases__)
        return 1

    depth = inheritance_depth(cls_reference)
    # A minimal positive score for attempting inheritance
    return max(depth, 0.1)
```

Figure 18: Inheritance fitness function

Here is a breakdown of how it works:

- An inner function inheritance_depth is created to calculate the depth of inheritance for a given class.
- If a class has no base classes (meaning it doesn't inherit from any other class), inheritance_depth returns 0, indicating no inheritance.

- If a class does have base classes, the function calculates the maximum depth of inheritance by recursively calling inheritance_depth on each base class and adding 1 for the current level of inheritance.
- If a class is at the top of the inheritance hierarchy (no further base classes), inheritance_depth returns 1.

The outer function evaluate_inheritance then calls inheritance_depth to get the depth of inheritance for the cls_reference and assigns a score to it. A minimal positive score (0.1) is given for any attempt at implementing inheritance, ensuring that even a class that does not inherit from others or only has a single level of inheritance receives some acknowledgment in the fitness evaluation.

This scoring system aligns with object-oriented principles by encouraging the design of class hierarchies. It rewards test cases where the class structure reflects deeper inheritance chains, which can be indicative of well-thought-out object models that take advantage of polymorphic behavior. This aspect of the fitness function, like the others (polymorphism, encapsulation, and method overriding), contributes to the overall assessment of how well a test case represents and handles the object-oriented features of the software being tested.

7.3.4 EVALUATING METHOD_OVERRIDING

The evaluate_method_overriding function, as illustrated in Fig 19 is implemented to assess whether method overriding is correctly implemented in a subclass-superclass relationship.

```
1 usage
def evaluate_method_overriding(self, subclass, superclass, method_name):
    subclass_method = getattr(subclass, method_name, None)
    superclass_method = getattr(superclass, method_name, None)
    # Score for successful overriding
    return 10 if subclass_method and superclass_method and subclass_method != superclass_method else 1
```

Figure 19:Method Overriding fitness function.

It works as follows:

- The function attempts to retrieve the method with the name method_name from both the subclass and superclass using the getattr function. If the method does not exist, None is returned.
- It then evaluates whether method overriding is successful by checking two conditions:

1. The method exists in both the subclass and superclass (subclass_method and superclass_method is not None).
2. The method in the subclass is different from the method in the superclass (subclass_method is not equal to superclass_method).

If both conditions are met, this means the subclass has successfully overridden the method from the superclass, which is a key aspect of polymorphism in object-oriented programming. The function then returns a score of 10 to indicate this success.

However, if the conditions are not met — meaning either the method is not present in both classes, and it is not overridden — the function returns a minimal score of 1. This ensures that some credit is given for the attempt, even if the overriding is not properly implemented.

By assessing method overriding, this function adds an important dimension to the fitness function from Fig 15: it rewards test cases that demonstrate the proper use of one of the fundamental concepts of object-oriented programming, which is the ability of a subclass to provide a specific implementation of a method that is already defined in its superclass.

7.3.5 EVALUATING FAULTS

Lastly, the evaluate_faults as illustrated in Fig 20 plays a critical role in the fitness function by penalizing test cases that exhibit certain shortcomings. Here's a breakdown of its mechanism and how it integrates with the overall fitness function:

```python
def evaluate_faults(self, test_case, weights=None):
    fault_score = 0
    # Scale factor to moderate the impact of faults on the overall fitness score
    scale_factor = 0.5  # Adjust this value based on desired impact
    # Standard penalties
    if test_case.get('is_negative', False) and not test_case.get('error_handled', False):
        fault_score += 20  # Increased penalty
    if test_case.get('test_type') == 'polymorphism' and test_case.get('num_polymorphic_methods', 0) < 3:
        fault_score += 15  # Increased penalty
    if test_case.get('test_type') == 'inheritance' and test_case.get('inheritance_depth', 0) < 2:
        fault_score += 12  # Increased penalty
    # Additional fault evaluations
    if test_case.get('test_type') == 'encapsulation' and test_case.get('encapsulation_score', 10) < 5:...
    if test_case.get('execution_time', 0) > 50:...
    if not test_case.get('has_edge_case_tests', False):...
    if test_case.get('relies_on_happy_path', False):...
    if test_case.get('input_validation', True) == False:...
    if not test_case.get('is_well_documented', False):...
    # Apply scale factor to the total fault score
    scaled_fault_score = fault_score * scale_factor
    # Consider additional weights for fault detection, if provided
    if weights:
        scaled_fault_score *= weights.get('fault_detection', 1)  # Use provided weight or default to 1
    return scaled_fault_score
```

Figure 20:fault detection fitness function

- Initialization: fault_score is set to 0 as the starting point for accumulating penalties.
- Scale Factor: A scale_factor of 0.5 is used to moderate the impact of the penalties on the fitness score. This can be adjusted depending on how significant the faults must be in the overall fitness evaluation.
- Standard Penalties: Specific penalties are added to the fault_score if certain conditions are met, relating to the type of test case and the presence or absence of desired features or behaviors. For example, if a test case is negative (is_negative) and error handling is not implemented (error_handled is False), a significant penalty is added. The same goes for test cases related to polymorphism, inheritance, and encapsulation if they don't meet certain criteria.
- Additional Fault Evaluations: Additional checks are performed for other potential issues, such as long execution times, lack of edge case tests, reliance on happy path testing, improper input validation, and insufficient documentation. Each of these aspects contributes to the fault score if the test case is found lacking.
- Application of Scale Factor: The total fault_score is then scaled down (or up) by the scale_factor. This scaled fault score reflects the aggregated impact of all the penalties,

tempered by the scale factor to ensure that faults don't disproportionately affect the fitness score.

- Weights for Fault Detection: If additional weights for fault detection are provided (weights argument), they are applied to the scaled fault score to further fine-tune the impact based on the importance of fault detection in the context of the fitness function.

The final scaled_fault_score is integrated into the fitness function by subtracting it from the evaluation scores from other positive aspects of the test case (such as polymorphism, encapsulation, etc.). This ensures that test cases with faults are given a lower fitness score, reflecting their reduced desirability. By penalizing faults, the evaluate_faults function ensures that the fitness score truly represents the quality of a test case. High-quality test cases with fewer faults will have higher fitness scores after accounting for both positive evaluations and penalties, guiding the optimization algorithm to prefer these more desirable solutions.

This function ensures that the fitness score reflects not just the presence of good OOP practices, but also the absence of issues that could undermine the effectiveness of the test cases, therefore contributing to a comprehensive assessment of test case quality within the Fitness function.

In summary, these implementations pave the way for the experiment in which the test cases generated by the hybrid algorithm are scrutinized against other metaheuristic algorithms namely: Genetic, SA, CS, TLBO, and the hybrid algorithm itself. They will be compared based on their average best fitness scores, average coverage percentage, and average execution time.

8

Experiment and Results

8.1 EXPERIMENTAL SETUP

To test the effectiveness of the hybrid algorithm against the other algorithms mentioned in the previous chapter, the values of the parameters used in the CS and SA parts of the hybrid algorithm must be set. Therefore, this chapter discusses the experiments performed on the hybrid to evaluate its effectiveness against other metaheuristic algorithms in terms of OOP testing.

The choice of parameter values in this algorithm is critical in balancing the exploration and exploitation phases during the search process and this is illustrated in Fig 21.

In terms of population size, a value of 20 was chosen as shown below, to allow for a sufficiently diverse set of solutions without putting excessive strain on the computational resources. This size is considered adequate, especially for the complexity of testing OOP as it provides a good variety of nests to explore the solution space effectively.

The nest size which represents the dimensionality of the problem, which is generating test cases, was set to 5. This number strikes a balance between a too-simplistic model that doesn't capture the necessary detail and a too-complex model that could cause an overfitting or unnecessary computational burden.

With the discovery probability of the worst nests, pa, the value of 0.5 was set. This represents a 50-50 chance for the worst nest being discovered by the algorithm and in turn abandoning this

worst nest. This value, as illustrated in the table below, provides an equilibrium between the exploration of new areas and the exploitation of known good solutions.

parameters	Description	Values
	Cuckoo search	
Population_size	The number of nests (solutions) within the search space. Each nest represents a potential solution to the optimization problem.	20
Nest_size	The dimensionality of the problem, indicating the number of parameters that define a solution.	5
pa	The probability of discovering a worse nest. This parameter controls the fraction of nests to be replaced by new ones in each generation.	0.5
beta	A parameter influencing the step sizes of Lévy flights, impacting the exploration capability of the algorithm.	1.0
nests	A NumPy array representing the initial population of nests. Each row corresponds to a nest (solution), and each column to a dimension (parameter) of the problem.	n/a
Best_nest	The best solution found so far, represented as a NumPy array.	n/a
Best_fitness	The fitness score of the best solution found so far.	n/a
	Simulated annealing	
Initial_temperature	The initial temperature, influencing the probability of accepting worse solutions initially, allowing exploration of the solution space.	10,100
Cooling_rate	The rate at which the temperature decreases, controlling the annealing schedule and the convergence of the algorithm.	0.03

Figure 21:Parameter values of CS and SA of the hybrid

A beta value of 1.0 was chosen as illustrated in fig 21, because it ensures that the steps are neither too small, which could make the search too local and slow, nor too large, which would potentially skip over viable solutions. As the description states in the table, it influences the steps that the algorithm must take to explore the solution space for the best test cases to be identified.

An initial temperature of 10,100 was chosen as it allows for a greater probability of accepting worse solutions in the initial stages of the annealing process. This would then help the SA process in its thorough exploration of the solution space and prevent the algorithm from becoming trapped in local optima early on.

A cooling rate of 0.03 was decided upon as illustrated in fig 21. This is because it ensures a gradual cooling, allowing the SA algorithm to spend more time searching the solution pool before it begins to cool down and become more selective in the solutions it accepts. This value also helps in fine-tuning the solution as the algorithm converges.

As noted in the table, there is also a lack of values for Nests, Best_nests, and Best_fitness due to the following:

1. Nests: The initial population of nests is not predefined statically since it is generated at the start of each optimization run. Therefore, each nest is randomly initialized to ensure a broad and unbiased coverage of the search space.
2. Best_nest and Best_fitness: Both 'Best_nest' and 'Best_fitness' are outcomes of the algorithm's execution. They are not set initially but are discovered as the algorithm iterates over different solutions. The 'Best_nest' will hold the best solution found during the execution, and 'Best_fitness' will record the corresponding fitness score. These are the results of the search process and therefore cannot have predefined values.

Coupled with setting the parameters, the hybrid CS SA will be tested using these eight OOPs illustrated in Fig 22.

	List of OOP python programs being tested
Program 1	Predicting forest fires with pytorch
Program 2	Triangle classification problem
Program 3	Six-sided die
Program 4	Mine sweeper
Program 5	Barber_shop
Program 6	Spreadsheet_converter
Program 7	GUI application1
Program 8	Performance sensitive application

Figure 22:List of OOP Python programs selected for testing with the hybrid CS-SA algorithm.

Each program will be assessed and compared to the other metaheuristic programs mentioned in the previous section of this report. The aim will be to identify how each program's object-oriented design patterns influence its adaptability to optimization algorithms. The metrics used for comparison—average best fitness scores, coverage percentage, and execution time—will yield quantitative data to measure the effectiveness and efficiency of the hybrid CS-SA. These comparative analyses will not only validate the hybrid algorithm's performance but also highlight the potential for OOP testing to be optimized through tailored metaheuristic strategies. By drawing from a range of computational experiments, this study seeks to establish a comprehensive benchmark for algorithmic performance in the domain of test case generation, providing valuable insights for future research and practical application in software engineering.

8.2 RESULTS

The experimental results provide a comparative analysis of the hybrid CS-SA against other metaheuristic algorithms: CS, SA, TLBO, and GA. The comparison is grounded on three metrics: average best fitness scores, average coverage, and average execution time, which serve as indications of the algorithm's performance in optimizing test case generation.

8.2.1 Average Coverage

As a metric, the coverage of the hybrid reflects the extent to which the test cases explore the different paths or states of the Python programs from Fig. 22. As seen in the table and histogram

of Figure 23 and Table 1, the hybrid CS-SA algorithm shows exemplary performance, achieving 100% coverage with each run across the tested programs.

Metaheuristic algorithms	coverage								
	Program1	Program 2	Program 3	Program 4	Program 5	Program 6	Program 7	Program 8	Average
Hybrid algorithm	100	100	100	100	100	100	100	100	100
Cuckoo algorithm	100	100	75	33.33	100	80	66.67	100	81.8712
Simulated annealing	50	66.67	41.67	22.22	100	60	33.33	60	54.2362
TLBO	50	66.67	41.67	27.78	100	80	33.33	80	59.9312
Genetic algorithm	25	66.67	50	16.67	100	60	16.67	60	49.3762

Table 1: Average Coverage

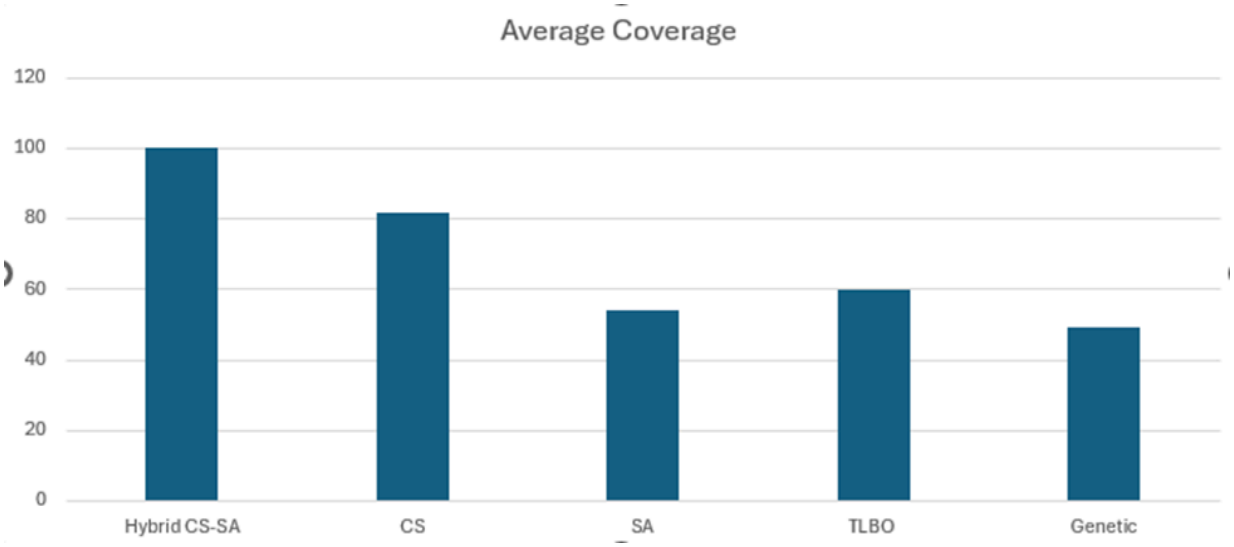


Figure 23: Results of coverage analysis

This consistent result indicates the hybrid's ability to ensure complete path coverage, a critical factor in thorough software testing. The average coverage of the CS demonstrated a high efficacy

with an average of 81.8712%, also suggesting its competence in test case exploration. However, the SA and TLBO algorithms exhibited moderate coverages with averages of 54.2362% and 59.9312%, respectively. GA has the lowest average coverage of 49.3762%, indicating its limitations in achieving extensive path coverage in test cases.

8.2.2 Average best fitness

The best fitness score is an assessment of the quality of the test case generated, with lower scores indicating higher quality. The hybrid CS-SA algorithm outperforms others with the lowest average fitness score of 3.62708 as shown in fig 24. This underscores the hybrid's precision in crafting optimal test cases. The CS and SA followed with an average of 4.0467 and 8.7698 reaffirming their effectiveness in the context of the hybrid algorithm.

Metaheuristic algorithms	Best fitness								
	Program1	Program2	Program3	Program 4	Program 5	Program 6	Program 7	Program 8	Average
Hybrid algorithm	4.09545	4.06553	3.96531	3.96393	4.73221	4.01231	4.15300	3.9986	3.627801
Cuckoo algorithm	4.14639	3.96309	4.09715	4.14293	3.95362	4.08420	3.96623	4.0204	4.046751
Simulated annealing	5.80024	9.53091	15.6310	9.87250	9.26965	9.45212	5.46989	5.3611	8.798426
TLBO	15.60542	27.07980	21.9346	21.41597	18.76547	21.50992	24.8609	25.591	22.09538
Genetic algorithm	26.99160	27.38259	26.6230	27.14998	25.91036	25.90836	26.4265	26.2085	26.57511

Table 2:Average Best Fitness

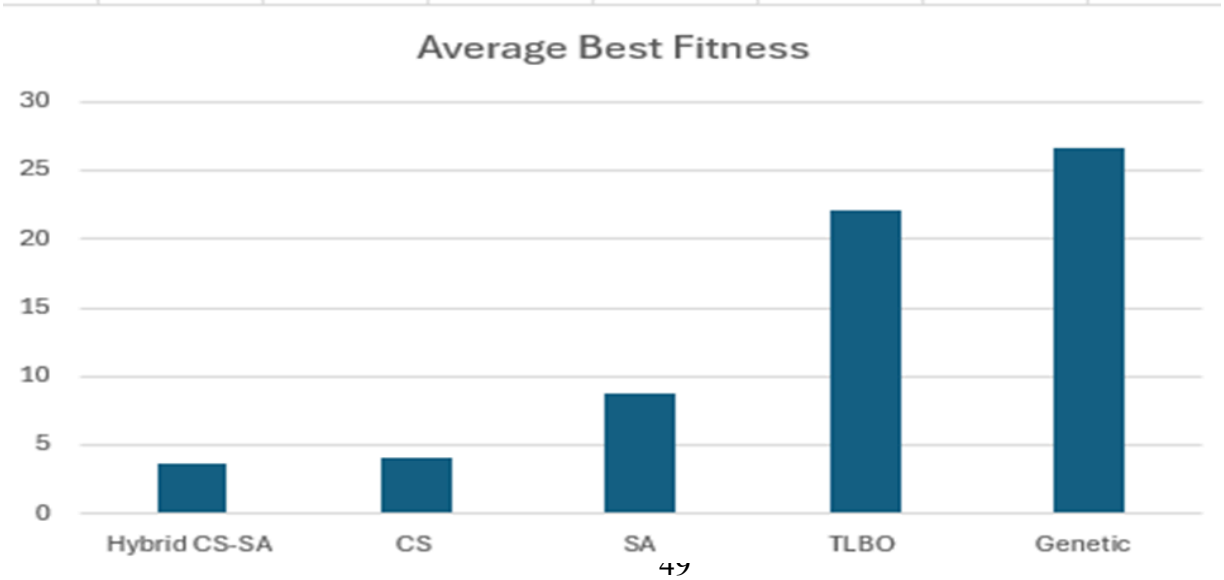


Figure 24: Result of best fitness analysis

On the other hand, TLBO and GA display a significant deviation from these results with an even higher average of 22.09538 and 26.57511 respectively, suggesting less optimal test case generation compared to the hybrid and the CS algorithms.

8.2.3 Average Execution Time

The execution time is a vital measure of efficiency in algorithm performance. Here, the hybrid boasts remarkable efficiency, reflecting the fastest average time as seen below in Fig 25. With an average of 0.00552125 seconds, this indicates not only the hybrid's capability to generate high-quality test cases but also its time efficiency, which is crucial in iterative development environments.

Metaheuristic algorithms	Execution time								
	Program 1	Program2	Program3	Program4	Program 5	Program 6	Program 7	Program 8	Average
Hybrid algorithm	0.00330	0.00126	0.01077	0.01819	0.00212	0.00145	0.00430	0.00278	0.00552125
Cuckoo algorithm	0.00535	0.00177	0.00223	0.07507	0.00230	0.00176	0.00475	0.00337	0.012075
Simulated annealing	0.24	0.19	0.24	0.14	0.14	0.22	0.20	0.29	0.2075
TLBO	0.05223	0.04671	0.04388	0.12396	0.03877	0.04416	0.04394	0.05625	0.0562375
Genetic algorithm	0.31403	0.44270	0.34356	0.33457	0.44369	0.32514	0.32458	0.37125	0.36244

Table 3: Average Execution Time

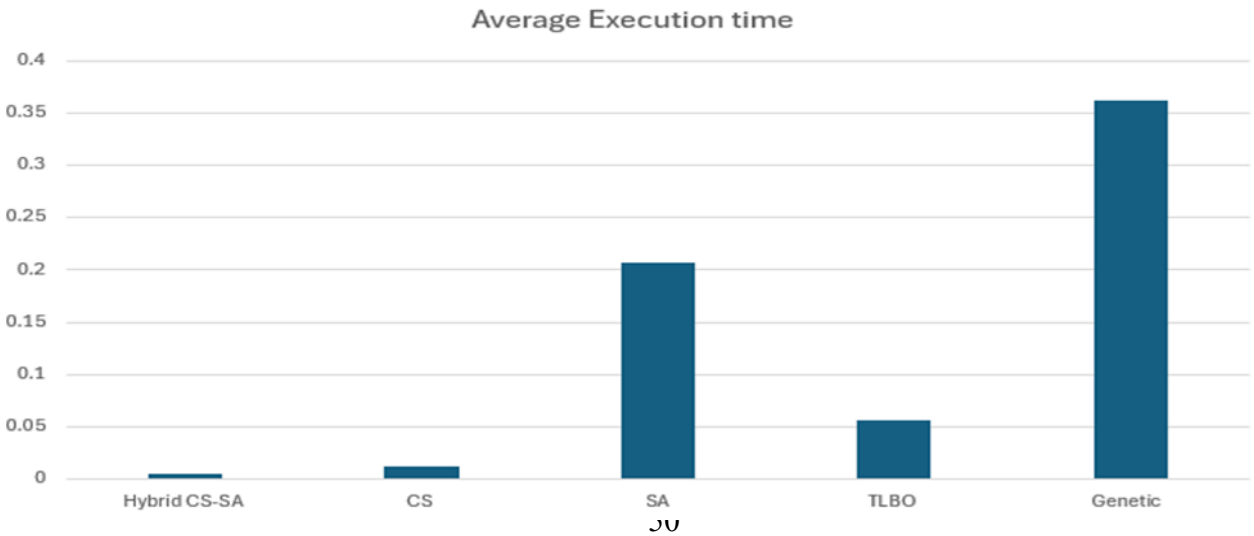


Figure 25: Result of Execution Time Analysis

The CS algorithm also showed an impressive performance, with an average of 0.012075 seconds, which is still twice as long as the hybrid algorithm. The SA, while having a moderate best fitness as discussed previously, now has a lagged execution time with an average of 0.2075 seconds. The TLBO and GA also exhibited higher execution times averaging at 0.0562375 and 0.36244 seconds, respectively. This indicates their lower efficiency in generating test cases quickly.

In summary, this chapter details the experiments performed on the hybrid algorithm and the results of testing the hybrid CS-SA algorithm against other metaheuristic algorithms in OOP testing. The parameters were optimized to ensure effective exploration and exploitation. The hybrid algorithm excelled in all key metrics: it consistently achieved 100% coverage and outperformed others in fitness scores and execution speed, demonstrating its superiority in generating optimal and efficient test cases for OOP environments.

9

Discussion

9.1 ANALYZING THE EFFICIENCY OF THE HYBRID ALGORITHM.

Based on the results of the experiment, the Hybrid CS-SA algorithm is superior in terms of coverage, best fitness, and execution speed. This helps to establish the fact that it is a highly effective tool in optimizing test case generation for software testing. Based on the design approach, the hybrid effectively leveraged the strengths of the CS and SA algorithms to overcome their weaknesses and lead to enhanced performance in terms of test case coverage, quality, and execution time. Therefore, this chapter discusses and analyzes the results of the experiments from the previous chapter.

The comparative analysis reveals that while the CS algorithm showed promising results, the distinct advantage of the hybrid approach is evident, suggesting that the integration of CS and SA strengths leads to a combined effect that improves performance. The moderate performance of SA and TLBO and the lower scores of GA emphasize the variability in the effectiveness of different metaheuristic algorithms and highlight the importance of algorithm selection based on the specific needs of the test case generation task.

Meanwhile, the hybrid CS-SA not only achieved 100% coverage consistently but also recorded the lowest average best fitness scores and the fastest execution times among the tested algorithms. These results highlight the hybrid's robustness and its ability to ensure comprehensive path coverage and high-quality test case generation with remarkable time efficiency. Such attributes are essential in software development environments where time constraints and the need for thorough testing are critical.

9.2 ADDRESSING THE OBJECTIVES: OUTCOMES AND INSIGHTS.

The experimental outcomes confirm the effectiveness of integrating CS and SA as a hybrid model, which outperforms the individual performance of CS, SA, TLBO, and GA algorithms in test case generation. The objectives of achieving optimal coverage, superior test case quality, and time-efficient algorithm execution were met, providing valuable insights into the potential of hybrid metaheuristic algorithms in software testing.

- Coverage Insight: The 100% coverage achieved by the hybrid CS-SA indicates its exceptional capability in exploring all paths or states within the Python programs tested. This is a significant improvement over the other algorithms, suggesting that the hybrid approach is more suited for thorough software testing.

- Quality Insight: The superiority of the hybrid algorithm in generating high-quality test cases, as indicated by the lowest average best fitness scores of 3.62708, highlights its precision and effectiveness. This suggests that the hybrid approach can better identify and generate test cases that are more likely to uncover errors or issues within the software, enhancing the overall quality of the testing process.
- Efficiency Insight: The rapid execution time of 0.00552125 per second which the hybrid CS-SA algorithm demonstrates shows its efficiency and suitability for iterative development environments, where quick turnaround times are essential. This efficiency does not compromise the quality or coverage of test cases, making the hybrid algorithm an ideal choice for software testing in fast-paced development settings.

In conclusion, the hybrid CS-SA algorithm represents a significant advancement in metaheuristic optimization for software testing. Its ability to achieve full coverage, generate high-quality test cases, and operate efficiently positions it as a valuable tool in improving the software development lifecycle. Future research could explore the integration of other metaheuristic algorithms to further enhance test case generation and optimization.

10

Reflection and Future Work

This chapter aims to detail the author's reflections in terms of what was learnt, challenges faced during the project, successes of the project, future work, and practical implications for the project.

10.1 LESSONS LEARNED FROM THE RESEARCH PROCESS.

Reflecting on the entire journey of the university program, both within and beyond this research project's specifics, the lessons learned offered a broader perspective on the skills, knowledge, and insights gained. Here are the lessons learned from both the research and the program of study:

1. Interdisciplinary Approach: The importance of integrating knowledge from various courses of study became evident during the project. This enabled me to display how solutions to complex problems often require a multi-faceted approach. Interdisciplinary understanding was crucial in navigating the challenges presented by modern software development and testing, especially in this project.
2. Ethical Implications and Professional Responsibility: Through the software engineering management course, lectures on ethics and professionalism have instilled an appreciation for the broader implications of software development, including privacy, security, and societal impact. These were reflected throughout the report.
3. Communication and Presentation Skills: Presenting projects, writing research papers, and engaging in discussions refined communication skills, both written and verbal. This helped me engage with the supervisors and peers on the project matter and strengthen my presentation skills.
4. Project Management and Organizational Skills: Managing deadlines, coordinating research activities, and balancing academic responsibilities honed project management and organizational skills necessary for this project. These were crucial

for successfully planning the project, managing time effectively, and ensuring the timely delivery of software solutions.

5. Understanding User Needs and Usability: With the course on introduction to software engineering, lectures on understanding user needs and requirements facilitated this project's development.

10.2 CHALLENGES AND LIMITATIONS ON THE PROJECT

The project encountered several limitations and challenges which provided critical insight for future research and practical applications.

These Limitations are presented as follows:

1. Incorrect coverage setup for GUI: Although the coverage report functioned correctly within the algorithm's program console, it did not successfully display in the GUI setup. However, this discrepancy does not imply that the coverage results obtained during the algorithm's evaluation are inaccurate.

2. Focus on test case generation only: The scope of the algorithm is limited to the generation of test cases, without extending to the actual execution and validation of the software being tested, which is necessary to confirm the utility of the generated test cases.

3. Preset Parameters for the Hybrid CS-SA: The parameters within the hybrid are pre-set and lack the flexibility for user-adjusted optimization, which could hinder the algorithm’s effectiveness across varying contexts and requirements.

4. Limited to OOP Test Cases: The current version of the code was specifically designed for generating test cases in OOP scenarios, which may not be applicable or optimal for procedural or functional programming paradigms.

5. Limited to Python programming language: The test case generation framework developed as part of this research is tailored specifically for programs written in the Python language. Hence, it does not cater to the structures of other programming languages like C++, Java, or JavaScript.

6. No consideration for abstraction and method Overloading: The algorithm does not account for complexities such as abstraction and method overloading, which are also inherent to OOP,

potentially affecting the relevancy and applicability of the generated test cases for more sophisticated OOP constructs.

7. No Actual test data: It's important to note that this project primarily focuses on generating appropriate test suites and test cases, and as such, it outputs placeholder test data which users are expected to replace with actual, concrete values. Quality test data is essential for the practical execution of test cases and thus, its absence could be perceived as a limitation of this work. However, this deliberate scoping decision was made to focus efforts on the underlying architecture of test case generation. However, this also highlights an area for future improvements in this work.

8. Scalability: The performance of the hybrid algorithm on larger and more complex systems remains to be tested. The current study was limited to a controlled set of programs, and scalability is a common challenge in optimization problems.

10.3 SUCCESSES OF THE PROJECT

The project successfully established the efficacy of a hybrid CS-SA algorithm for generating test cases in an object-oriented programming environment. Through comprehensive testing and comparative analysis with other metaheuristic algorithms, the hybrid algorithm demonstrated superior performance across key metrics, affirming its potential as a significant contribution to the field of automated software testing.

10.4 FUTURE WORK AND APPLICATION

Addressing the outlined challenges, the future work will be centered around:

1. Expanding the capabilities from test case generation to include execution, validation and actual test data.
2. Implementing a user interface for dynamic parameter adjustment to tailor the algorithm to specific needs.
3. Adapting the approach to support a broader range of programming paradigms beyond OOP.
4. Integrating advanced OOP features such as abstraction and method overloading into the test case generation process.

5. Algorithm Scalability: Investigating the performance of the hybrid algorithm on a broader range of software applications, including large-scale and enterprise systems, will be essential.
6. Future Work for Language Extensibility: Given the current limitation of the hybrid CS-SA algorithm's applicability to only Python programs, future research should aim to extend the framework to other programming languages i.e., C++, Java, JavaScript, etc.

10.5 PRACTICAL IMPLICATIONS FOR OOP TESTING

The practical implications of this work are significant:

- Efficiency in Software Testing: The hybrid CS-SA algorithm can reduce the time and resources required for test case generation, leading to more efficient development cycles.
- Quality Assurance: Improved test case generation directly correlates to better software quality, as more thorough testing can lead to the discovery and resolution of defects before deployment.
- Automation: Incorporating the hybrid algorithm into testing tools can enhance the automation of test case generation, reducing the need for manual intervention and the potential for human error.

In summary, this chapter reflects on the project's insights, outlining lessons learned, challenges faced, and future directions. Key takeaways include the importance of interdisciplinary approaches, ethical considerations, and effective project management. Challenges such as limited GUI functionality and scalability provide focus areas for further research. The project demonstrated the effectiveness of the hybrid CS-SA algorithm in OOP test case generation, highlighting its potential to improve automated software testing. Practical implications emphasize the benefits of improved efficiency and quality assurance in software testing through automation.

11

Conclusion

This project developed and provided a thorough examination of a hybrid CS-SA algorithm's ability to optimize test case generation for object-oriented programming. The empirical results clearly show the hybrid algorithm outperforms traditional metaheuristic algorithms in terms of coverage, best fitness scores, and execution time. These findings highlight the potential of hybrid metaheuristic algorithms in improving the efficacy and efficiency of software testing processes.

The project's success opens new doors for software testing, reinforcing the importance of optimization algorithms in the continuous pursuit of quality and excellence in software development. Moreover, the identified challenges and limitations pave the way for future investigations, ensuring ongoing progress in automated software testing techniques.

The culmination of this research marks not only the completion of an extensive study but also the beginning of a new chapter in which the hybrid CS-SA algorithm could become an integral part of the software testing landscape, promising higher standards of software quality and reliability for the future.

APPENDICES

APPENDIX A

GUI user guide

This is a user guide for the hybrid metaheuristic algorithm test case generator. This tool is designed with a focus on the non-functional requirements of security and the LESPI considerations raised in the previous chapters. This guide walks the users through the features of the Graphical user interface (GUI), ensuring you can navigate the tool effectively and understand the security measures in place.

As shown in Fig 27 below, the user can log in and a screen displaying the terms and conditions as well as copyright information is displayed to give the user a clear view of what is and what is not allowed with the tool. It also explains what the tool can and cannot do.

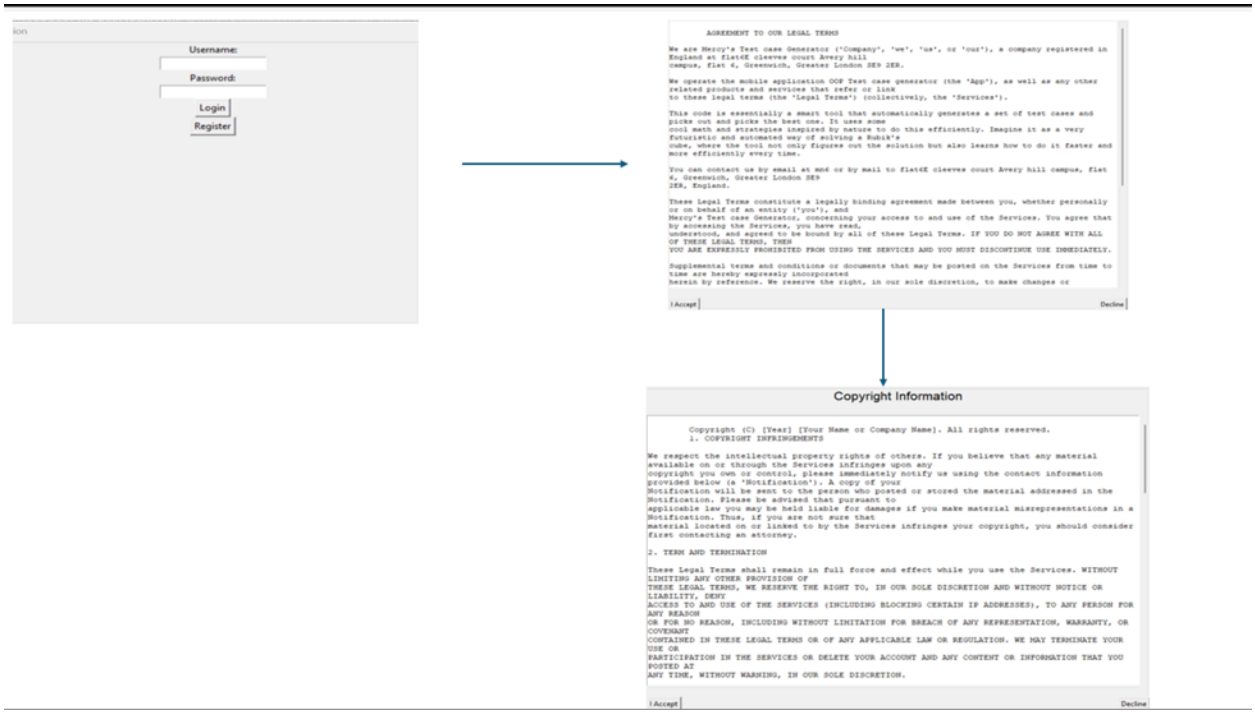


Figure 26: Security checks and legal information

If the user disagrees with any of the terms and conditions or the copyright information, they are referred to the login page until they accept the terms and conditions as shown in Fig. 28 below.

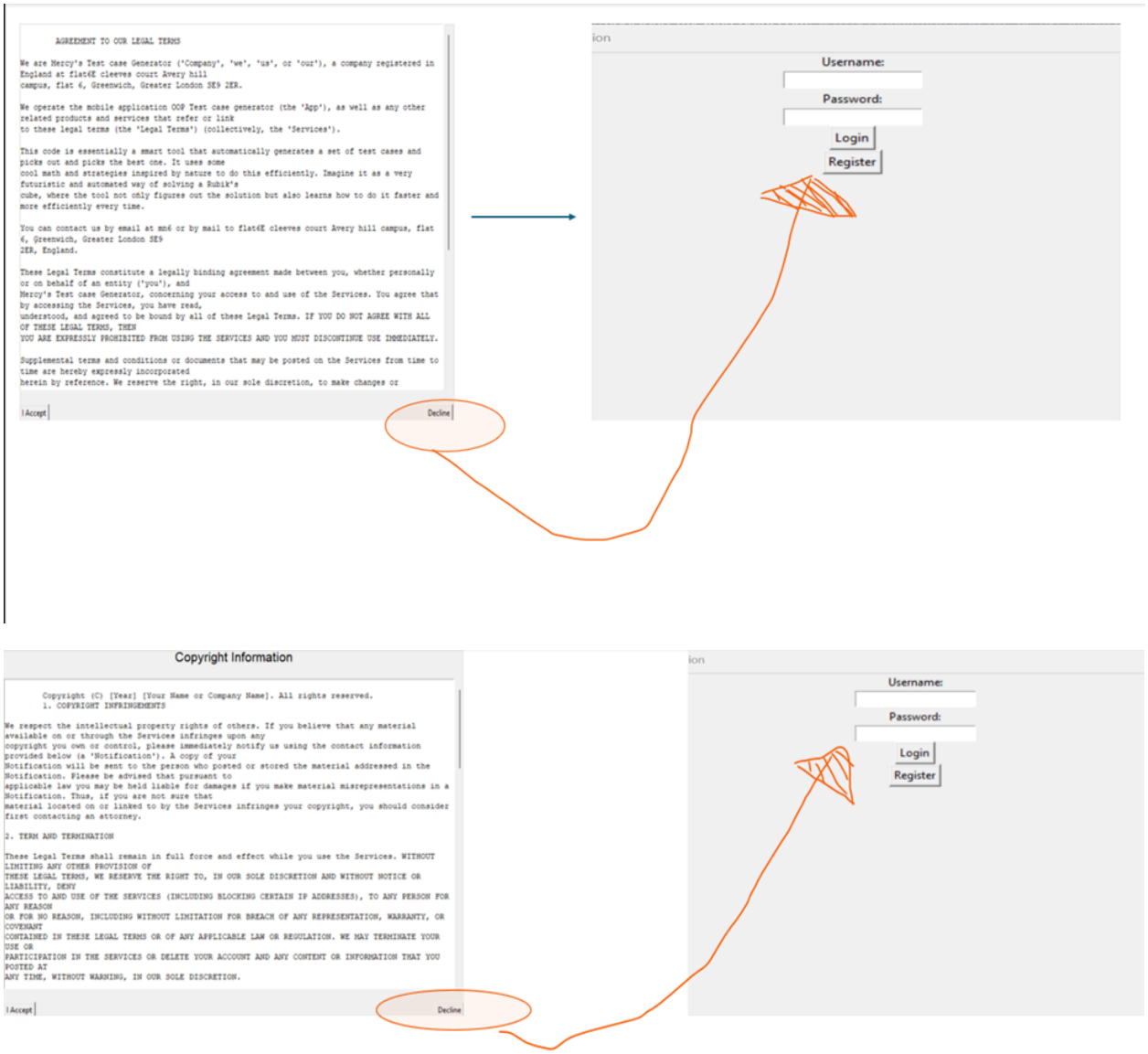


Figure 27: The mechanism set in place for users to agree to terms and conditions.

The main Test case generator

To cater to both technical and non-technical users, the design includes a user-friendly interface with the input field at the top of the screen and the display field at the bottom of the screen as

seen in the figure below. If the user needs to save these test suites created by the application, they can save this with the save button at the bottom of the screen as seen in Fig. 29. This interface provides intuitive access to the tool's capabilities, making advanced testing methodologies accessible to a broader audience.

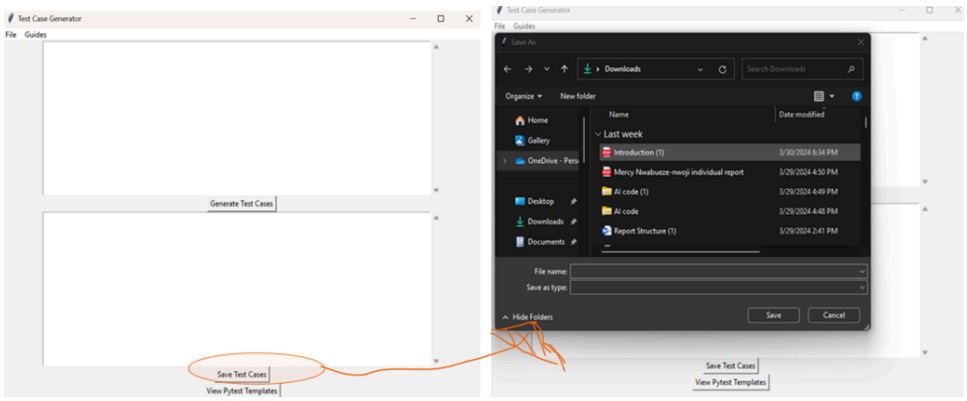


Figure 28: User input code

Key Design Features

Accessible and User-Centric Interface: The GUI is designed with user-friendliness in mind, boasting a clean and organized layout that simplifies the process. Clearly labeled sections facilitate effortless navigation for inputting code, initiating test case generation, and reviewing results. Additionally, tool guides and readily available test templates, displayed in Fig. 30, streamline user interaction, guiding them through each step seamlessly. By reflecting professionalism, the GUI encourages best practices in software testing and development, contributing positively to the advancement of the field's body of knowledge.

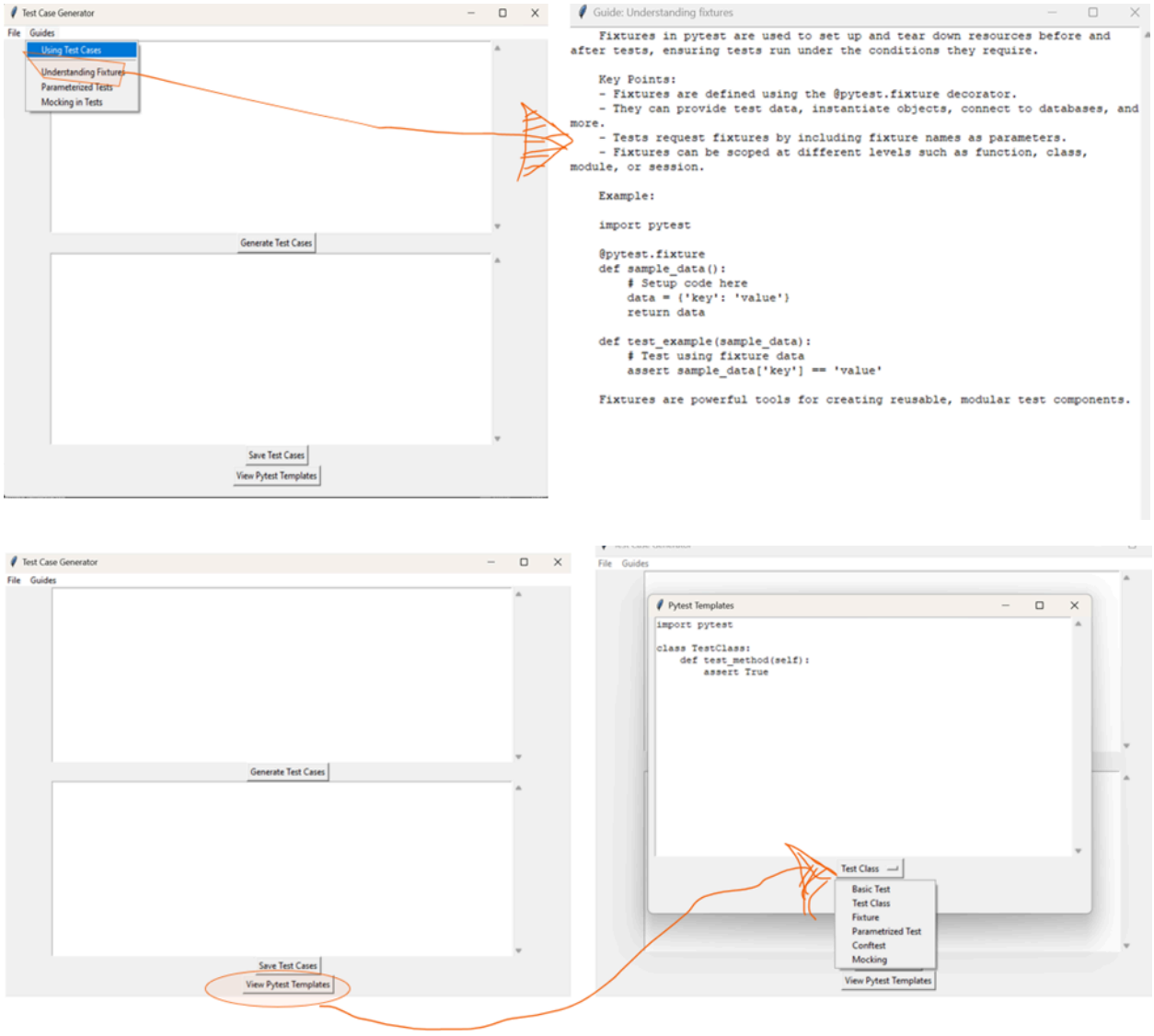


Figure 29: Guide on how to use the tool and templates for creating Pytests.

Legal, Ethical, and Security Standards: The system's design prioritizes adherence to legal and ethical standards, encompassing copyright laws and privacy regulations. It pledges not to retain user data without explicit consent and safeguards all user-generated content as their rightful property. Moreover, robust security measures, such as data encryption and secure session management, are integrated into the GUI to uphold the confidentiality and integrity of users'

code and test cases. Authentication protocols are enforced before system access, thereby fortifying the protection of sensitive information, as illustrated in Fig.31.

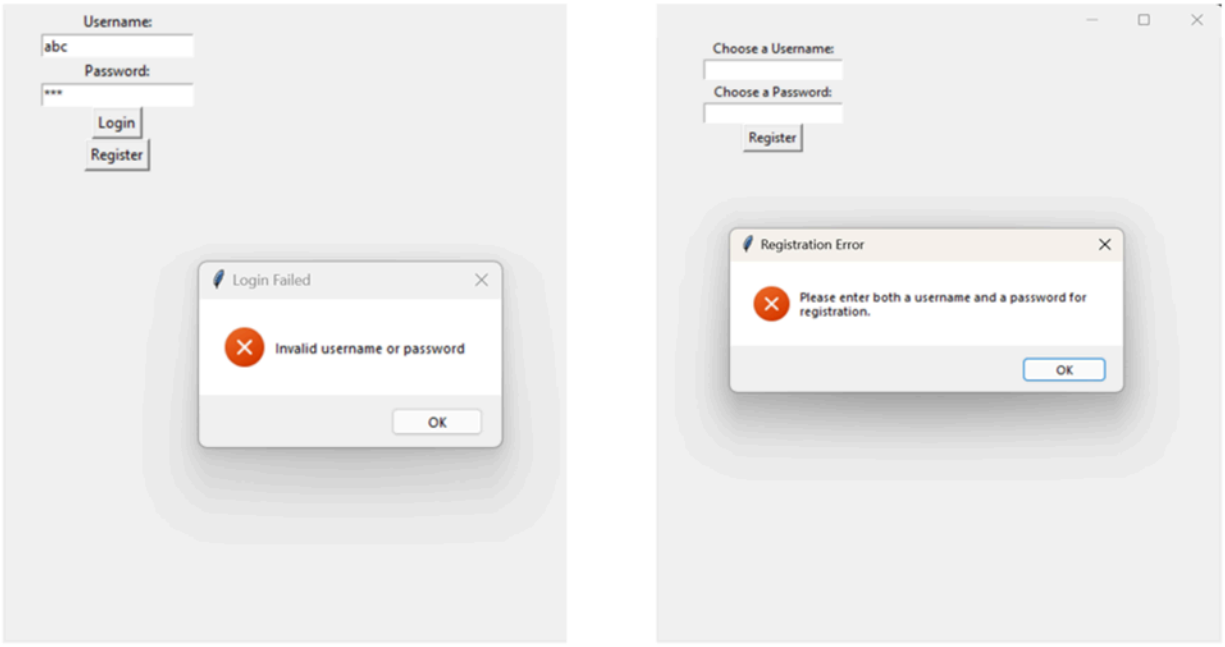


Figure 30:Security measures set in place to ensure secure sessions and prevent data from being stolen.

Real-Time Feedback and Reporting: The interface will provide detailed feedback on the test case generation process, along with comprehensive reports on the test cases' coverage, effectiveness, and potential areas of improvement as illustrated in Fig. 32. This feature supports the continuous development and refinement of software.

Figure 31: Receiving feedback.

In summary, the design of the GUI for the hybrid metaheuristic algorithm test case generator is integral to the tool's success, directly impacting its usability, security, and overall effectiveness. By prioritizing user experience while upholding the LESPI considerations, the tool aims to set a standard for accessible, ethical, and professional software testing solutions.

Using the Tool

Generating Test Cases

3. Input your code into the top field of the main screen.
4. Click on the "Generate Test Cases" button to initiate the process.
5. View the generated test cases in the display field at the bottom of the screen.
6. Use the "Save" button to download and store your test suites for later use.

APPENDIX B

Integration into the software testing Life Cycle

The integration of the hybrid CS-SA algorithm into the software testing life cycle enhances the efficiency and effectiveness of various types of software testing. Not only does it reduce the chances of manual error as highlighted in the report, but it also streamlines the test case generation process, allowing for a more focused comprehensive approach to automated and manual testing methodologies. By leveraging the strengths of both the CS and SA algorithms, the hybrid significantly improves upon the limitation of testing OOP concepts, offering robust solutions that can adapt to the complexities of modern software development.

The hybrid CS-SA algorithm's capability to achieve 100% coverage ensures that all paths and states within the software are explored, thus minimizing the risk of undetected bugs or issues. This level of thoroughness is particularly beneficial in agile and DevOps environments, where rapid iterations and continuous integration/continuous deployment (CI/CD) practices demand both speed and accuracy in testing. The algorithm's efficiency in generating high-quality test cases quickly aligns perfectly with these methodologies, enabling teams to maintain a high pace of development without compromising software quality.

The benefits of the integration into the software testing life cycle would include:

- Reduction of Manual Errors: Automating the test case generation process minimizes the reliance on manual testing efforts, which can be prone to oversights and errors. This automation ensures a more reliable and consistent testing phase, critical for maintaining software quality.

- Efficiency in Test Generation: The hybrid algorithm's ability to quickly generate test cases directly impacts the overall testing timeline, facilitating faster development cycles and quicker time-to-market for software products.

- Adaptability to Various Testing Needs: Whether it's regression testing to check for unintended side effects, stress testing to evaluate system performance under extreme conditions, or security testing to uncover vulnerabilities, the hybrid CS-SA algorithm

can be tailored to meet specific testing objectives, enhancing its utility across the software development lifecycle.

- Cost-Effectiveness: By optimizing the test case generation process, the hybrid algorithm reduces the amount of time and resources required for testing. This efficiency translates into cost savings for the development project, especially significant in large-scale or long-term projects.

- Enhanced Software Quality: The comprehensive coverage and high-quality test cases generated by the hybrid algorithm contribute to a more robust and reliable software product. This not only improves user satisfaction but also reduces the potential costs associated with post-release patches and fixes.

In conclusion, the integration of the hybrid CS-SA algorithm into the software testing life cycle represents a significant advancement in the field of software testing. By enhancing the efficiency, effectiveness, and reliability of the testing process, it provides a solid foundation for producing high-quality software that meets the dynamic needs of users and stakeholders.

How It Fits into All the Different Types of Software Testing

The section elaborates on how the hybrid algorithm aligns with different types of software testing, further highlighting its adaptability in meeting diverse setting requirements.

1. Unit testing: Since this focuses on testing individual components of the software to ensure they function as expected, the hybrid's job here would be to generate precise test cases that cover a wide range of input scenarios for each component, thereby identifying even the most obscure bugs that could compromise the software's integrity.
2. System testing: The entire software system is tested to verify that it complies with the specified requirements. So, the hybrid is expected to generate comprehensive test cases that assess the system's functionality, reliability, and stability under different conditions, ensuring that the software meets all performance and operational requirements before it is released to the market.

3. Performance testing: With this testing type, the software's performance is evaluated under specific conditions. Hence the hybrid can be used to generate test cases that simulate various loads and stress conditions on the software, providing insights into its scalability, reliability, and speed.

APPENDIX C

Challenges in testing OOP

Now while object-oriented programming concepts such as polymorphism, encapsulation, inheritance, and method overriding have streamlined software development, they simultaneously pose a significant challenge when it comes to testing these programs.

M. Ghoreshi and Haghighi, (2023) define polymorphism as allowing objects to be viewed as instances of their parent class rather than their class, it facilitates the binding of dynamic methods and allows for a variety of execution-related behaviors. This concept gives the program the flexibility to exhibit different behaviors based on a specific situation. Figure 1 gives an example of what polymorphism looks like:

```python
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass

# Child class
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Child class
class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Function that works with any Animal object
def animal_sound(animal):
    print(f"{animal.name} says: {animal.make_sound()}")

# Creating instances of the child classes
dog_instance = Dog("Buddy")
cat_instance = Cat("Whiskers")

# Calling the function with different types of objects
animal_sound(dog_instance)
animal_sound(cat_instance)
```

Figure 32: Depiction of Polymorphism.

The parent class Animal has the constructor "__init__," which initializes the name and specific properties. It also has the method "make_sound" that does not have an implementation meaning that any subclass inheriting from the Animal class is expected to provide its different implementations. The dog and the cat classes are the children's classes of Animal, and they provide a specific implementation to the "make_sound" method which is different from each other i.e., the Dog makes the "woof" sound, and the cat "Meows." Therefore, the object Animal takes on several forms such as "Dog" and "Cat." Now this is particularly advantageous in

programs as it allows a single name, method, or operator to be associated with several types of operators like Nwokoro, I. et. Al. (2021) suggests.

However, this could prove to become an issue if an object's behavior changes unexpectedly due to the dynamic binding of methods from different classes in the inheritance hierarchy.

In the case of inheritance, this concept allows classes to be reused and extends the definition of other classes. M. Ghoreshi and Haghighi, (2023) highlight this as "Many classes reuse the definition of other classes to define themselves." An example of this is illustrated in Figure 33.

```
# Copyright (C) Deepali Srivastava - All Rights Reserved
# This code is part of Python course available on CourseGalaxy.com

class Person:
    def greet(self):
        print('I am a Person')

class Teacher(Person):
    def greet(self):
        print('I am a Teacher')

class Student(Person):
    def greet(self):
        print('I am a Student')

class TeachingAssistant(Student, Teacher):
    def greet(self):
        print('I am a Teaching Assistant')

x = TeachingAssistant()
x.greet()

print(TeachingAssistant.__mro__)
print(TeachingAssistant.mro())
print(x.__class__.__mro__)
```

Figure 33: Depiction of inheritance

The class “Person” is the base class that has the “greet” method which outputs “I am a person.” The classes “Teacher,” “Student” and “TeachingAssistant.” These classes have a similar function “greet” which overrides each other. Hence, the subclasses inherit the properties and methods of the base class. This is important as it provides reusability of the base class instead of trying to recreate it similarly as Raut, R.S., (2020) suggests. Therefore, there is an increased likelihood of new faults being introduced to the program when there is an increase in the level of inheritance.

For method overriding, it is said to enable a subclass to modify a method that exists in its parent class. This shows how capable and flexible OOP code is. This is also evident in the image depicting inheritance. The subclass “Teacher” both inherits and overrides the method “greet,” so that when “greet” is called on an instance of “Teacher,” it will print "I am a Teacher "instead of “I am a Person” as seen in the previous iteration of the class.

However, M. Ghoreshi and Haghighi, (2023) also underline that this complicates the testing process "In comparison to procedural programs, OO programs have significantly shorter and simpler methods; instead of using lengthy and complex methods, these programs implement needed functionalities via interactions between dozens of simple methods and relationships between classes.” This approach introduces subtle interactions between methods of multiple classes to fulfill a single purpose, especially when a method in a class uses an overridden method from a parent class. Procedural testing techniques, which consider each method independently, could miss these important interactions, producing high code coverage that mistakenly suggests thorough testing. This gap emphasizes the need for testing methodologies that consider the intricate network of connections between methods key to OOP, especially when testing overridden methods.

Finally, for encapsulation, M. Ghoreshi and Haghighi, (2023) highlight OO programs as using this feature to significantly ‘rely on user-defined types’ which include examples like state space and behaviors that can form different objects during the execution phase.

In Figure 34, encapsulation allows the “Game” class to maintain a controlled environment for the game's state and behaviors and this is depicted through the methods “run”, “check_if_game_over”, and “_take_turns.”

```python
from player import Player


class Game:
    counter = 0

    def __init__(self, num_players=2, target_score=100):
        self.target_score = target_score
        self.players = [Player(i + 1) for i in range(num_players)]
        Game.counter += 1
        self.num = Game.counter

    def run(self):
        print(f"---- {self} START ----".upper())
        self._take_turns()
        print(f"---- {self} END ----".upper())

    def check_if_game_over(self):
        game_over = False
        winner = None
        high_score = 0
        for player in self.players:
            if player.score > high_score:
                winner = player
                high_score = player.score
            if player.score >= self.target_score:
                game_over = True
        return game_over, winner

    def _take_turns(self):
        while True:
            for player in self.players:
                player.take_turn()
            game_over, winner = self.check_if_game_over()
            if game_over:
                print(f"{winner} wins!")
                return

    def __str__(self):
        return f"Game {self.num}"
```

Figure 34:Depiction of Encapsulation

Consequently, this feature can bundle attributes and methods within objects, hiding their internal state. This coupled with what the authors stated could mean that in terms of testing, this feature could increase the program's complexity. This is because the very nature of encapsulation is to hide information in terms of behavior and internal states making it difficult for tests to directly access and verify the correctness of the encapsulated data or the internal workings of methods.

To address these complexities, this study attempts to tackle the specific challenges posed by object-oriented programming concepts as discussed.

The development of software testing paradigms has observed certain gaps present in traditional methodologies while testing OOPs. OOP introduces its own kind of complexity through mechanisms like polymorphism, inheritance, encapsulation, and method overriding, which are not suitably taken care of under standard testing paradigms.

Sumit K. et al. (2018) highlight the limitations of evolutionary algorithms which is a progressive stride in automated testing. They acknowledge its propensity to stagnate at local optima and encounter issues with maintaining population diversity. These shortcomings signal the inability of such algorithms to navigate the elaborate behavioral landscapes of OOP effectively.

This is further backed by Mitchell O. (2022) stating that for automated test case generation (TCG) to be viable in industrial settings, a pivot is needed from sheer test coverage to the eminence of the test cases themselves. This signifies the inadequacy of prevailing metrics, which may overlook the qualitative nuances integral to OOP's dynamic behaviors.

Up to 50% of developers' time is wasted on manual test case creation, which is further criticized for being inefficient, and prone to errors Mitchell, O. (2022). This method is not sustainable in an OOP context, where complex interactions between objects are common, and it increases the risk of oversights and represents a considerable resource drain.

Dongcheng Li et al. (2022) further emphasize this overreliance on manual effort in traditional software testing. The manual workload is incongruent with the automated mechanisms that characterize modern software development, particularly within OOP, where the automated generation and testing of dynamic behaviors can be pivotal.

Seema Rani (2020) highlights that the efficient automation of test case generation, particularly for structural testing i.e. an aspect at the heart of OOP, is still an endeavor fraught with challenges. This statement encapsulates the crux of the issue: traditional methods fail to keep pace with the complexity and structural nuances of OOP, rendering them insufficient for comprehensive testing.

In summary, the literature does point to agreement on one central critique: the testing practices, whether manual or automated, fall short in their engagement with OOP's inherent complexities.

This gap in the efficacy of testing methodologies necessitates an exploration of more sophisticated approaches. To produce test cases of greater quality that are resilient to OOP's dynamic execution routes and can detect even the smallest error brought about by its fundamental features, it should be possible to traverse the complexities of OOP.

APPENDIX D

Development and self-testing of the Hybrid Metaheuristic Testing Tool

In developing the hybrid metaheuristic testing tool, a primary objective was to address the unique challenges posed by object-oriented programming (OOP) in software testing. OOP concepts like polymorphism, inheritance, encapsulation, and method overriding introduce complexities that require sophisticated test case generation and evaluation strategies.

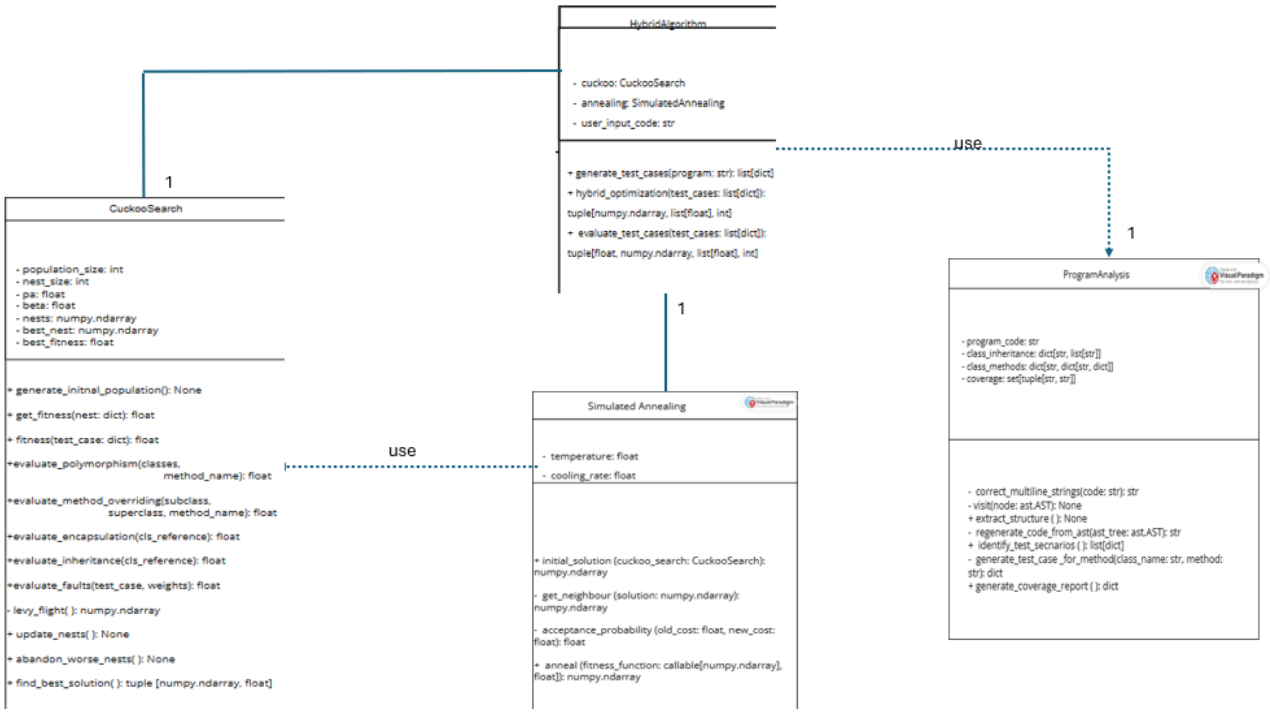


Figure 35:class hierarchy of the tool

The Hybrid CS-SA algorithm implements several key OOP principles including encapsulation, polymorphism, composition.

In terms of Composition, the HybridAlgorithm class is a prime example where it contains instances of CuckooSearch and SimulatedAnnealing. This shows a strong ownership where HybridAlgorithm controls the creation, usage, and lifecycle of these objects. They share an object-oriented approach to solving different parts of the problem i.e. test case generation.

The use of encapsulation is obvious in classes such as the Cuckoosearch and SimulatedAnnealing. They manage their own state internally and expose functionality through public methods while keeping key details private. Examples of these are illustrated above with private functionalities such as levy_flight(), get_neighbour(), acceptance_probability().

Dynamic typing was implemented to allow methods like get_fitness in the Cuckoosearch class to operate polymorphically on different types of input, providing flexibility in how objects are handled.

In terms of dependency the HybridAlgorithm class uses ProgramAnalysis class as illustrated above, indicating a use relationship. The same goes for the SimulatedAnnealing which uses the CuckooSearch for initial solutions.

Testing of the Tool

1. Performance testing

The tool was subjected to rigorous performance testing, including a self-application method, to validate its capabilities and effectiveness. The tool's performance test was conducted using the triangle classification benchmark program used in evaluating the test case generator. This was carried out in relation to the non-functional requirements in order to validate the system based on its performance.

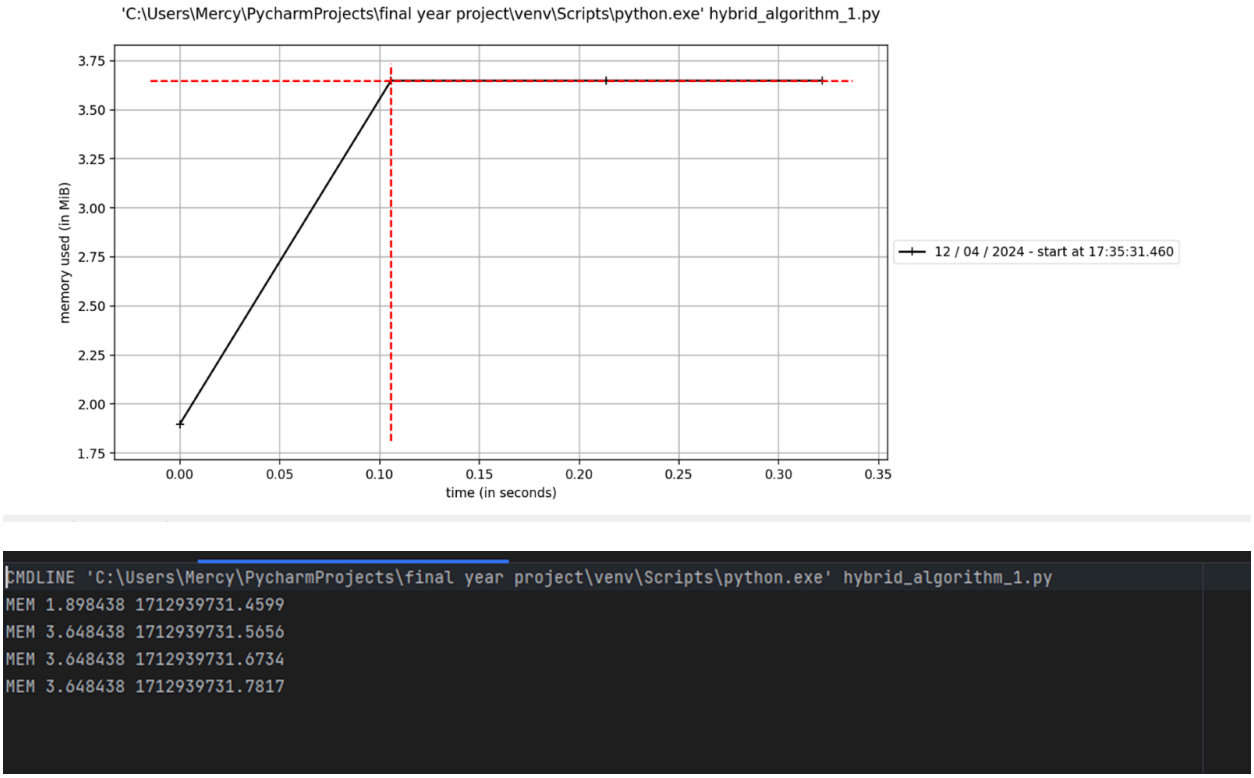


Figure 36:Memory profiling of the tool using Triangle classification

Memory profiling was conducted to monitor the tool's memory consumption throughout its runtime, identifying any instances of inefficiency or potential leaks. The illustration above suggests that there is an initial memory of about 1.8 megabytes which then sharply rises to approximately 3.65 MiB in a very short time span (less than 0.1 seconds), after which the memory usage plateaus.

This suggests that the hybrid CS-SA algorithm quickly allocates a chunk of memory at the beginning of the program's run. The flat line after the spike implies that the hybrid maintains a steady memory usage after the initial allocation of memory. The dashed line indicates the threshold set for alerting or predicting memory usage trend.

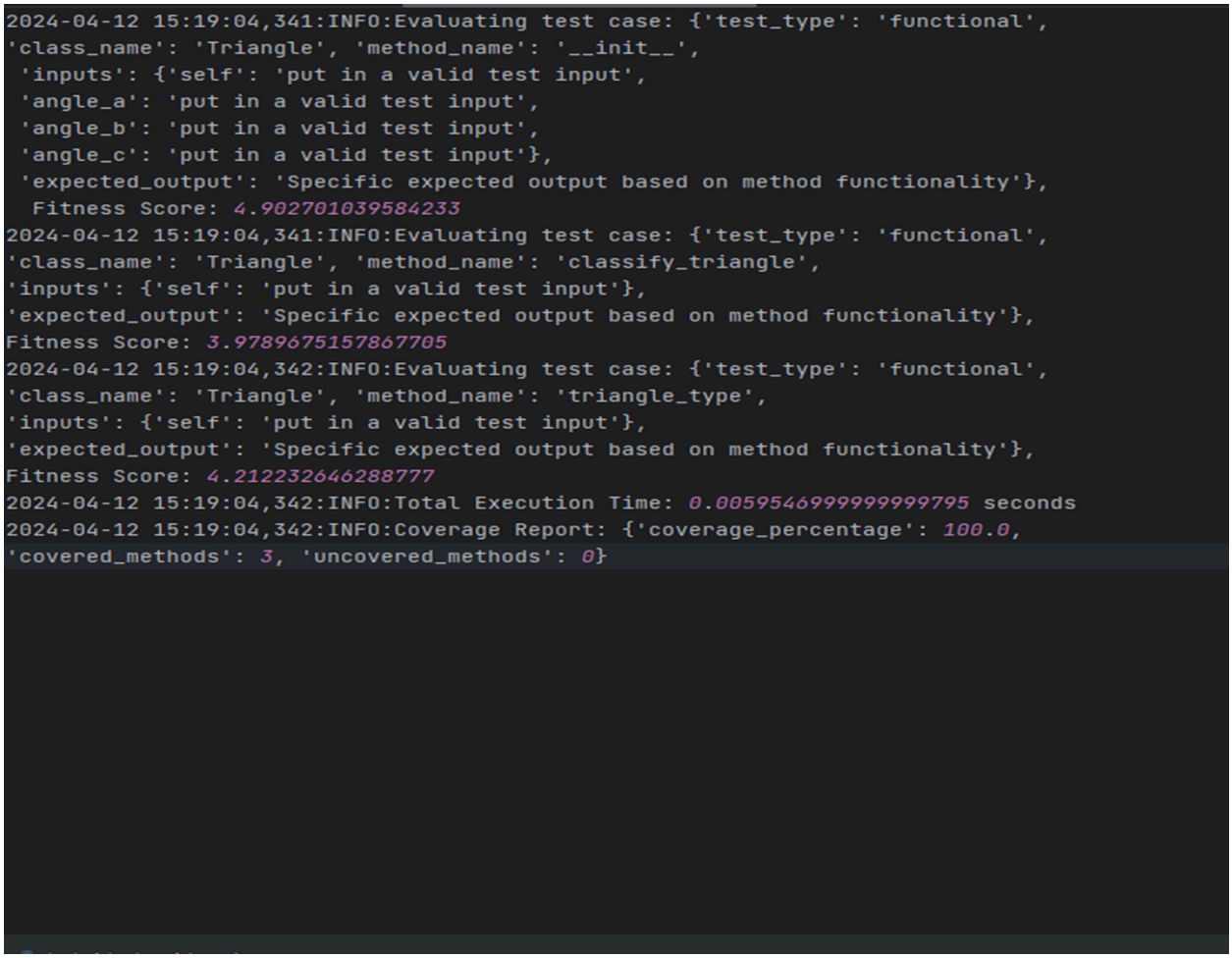


Figure 37: Logging Information

Additionally, execution logs were utilized to provide detailed insights into the functionality and efficiency of the tool's performance. Based on the information provided in the execution logs, the tool appears to execute with high efficiency and thorough functional coverage. The following is a continuation of the report detailing these aspects:

1. Test Coverage and Functionality: The software testing process focused on the Triangle class, successfully invoking all relevant methods (__init__, classify_triangle, and triangle_type), as indicated by the 100% method coverage reported in the logs. This comprehensive method coverage is an excellent indicator of the test suite's ability to assess the functionality across the class's interface.

2. Fitness Score: The fitness score associated with each test case suggests the use of a robust testing strategy, potentially employing advanced techniques such as genetic algorithms or mutation testing. The provided fitness scores (ranging from approximately 3.97 to 4.92) may signify the effectiveness of test inputs in validating the expected functionality of the methods.
3. Efficiency: The execution time logged for the test cases is remarkably swift, with the total execution time being less than a hundredth of a second (0.00595 seconds). This level of efficiency in test execution enables rapid feedback loops, which are vital in a continuous integration and continuous deployment (CI/CD) pipeline.

2. Unit testing

To assess the individual components of the hybrid tool, a unit test was performed, but this process was started off by generating test cases for the tool itself using the tool. This approach to generating test cases for itself demonstrates a form of recursion where the testing tool is applied to the very code it constitutes. This carried out to ensure:

1. The test cases generated are pertinent and comprehensive, addressing the nuances of OOP.
2. The optimization process leads to the discovery of the best possible solutions.
3. The fitness function adequately reflects the multi-faceted objectives of the tool.
4. The performance of the tool is quantified in terms of execution speed and memory usage.
5. The code coverage is thoroughly reported, indicating the effectiveness of the test cases.

The result of this was successful with a best fitness of 3.978 and an execution time of 0.0407 and a coverage report of 100%.

```
Best Fitness Score: 3.978161436852149
```

```
The time taken to generate test cases was 0.04073909999999997
```

```
Coverage Report:
Coverage Percentage: 100.00%
Covered Methods: 40
Uncovered Methods: 0
```

However, some changes had to be made to the program as the algorithm couldn't create test cases for methods outside of class "ProgramAnalysis". Hence, the methods outside of the class were moved into the ProgramAnalysis class.

This difficulty in testing the program itself refers to one of the challenges in the main report that talks about the algorithm only working on OOP programs.

After these test cases were generated, each of the classes were isolated i.e. 'CuckooSearch', 'SimulatedAnnealing', and 'ProgramAnalysis.' These classes were tested separately to focus on the individual functionalities and methods of the hybrid algorithm.

Functional testing is carried out by methods being tested in isolation to ensure that they behave as expected under controlled conditions. For instance,in the TestCuckooSearch, methods like test_generate_initial_population and test_get_fitness are functional tests that ensure the specific functions of the CuckooSearch class are working as intended.

```python
class TestCuckooSearch(unittest.TestCase):
    def setUp(self):
        # Assume reasonable defaults for initialization for testing
        self.cs = CuckooSearch(population_size=20, nest_size=5, pa=0.5, beta=1.0)

    def test_init(self):
        # Check if initialized correctly
        self.assertIsInstance(self.cs, CuckooSearch)

    def test_generate_initial_population(self):
        self.cs.generate_initial_population()
        self.assertEqual(self.cs.nests.shape, second: (20, 5))  # Assuming nests is a numpy array

    def test_get_fitness(self):
        # Assuming get_fitness should return a numeric value
        nest = {'score': 1.0}  # Mocked nest dictionary
        fitness = self.cs.get_fitness(nest)
        self.assertIsInstance(fitness, float)
```

Mocks and Stubs were utilized to mock dependencies and certain behaviors, this was so the tests could be carried out in a controlled environment. This was particularly evident in the tests for the

class ProgramAnalysis where methods like the ‘extract_structure’ and ‘_parse_ast’ were mocked.

```
def test_extract_structure(self):
    """Test the method that extracts structure from the code."""
    self.pa.extract_structure = MagicMock(return_value=None)  # Mocking behavior
    self.pa.extract_structure()
    self.pa.extract_structure.assert_called()

def test_parse_ast(self):
    """Test parsing of the AST from the program code."""
    # Assuming '_parse_ast' is an internal method, you may not typically test it directly
    node = MagicMock()  # Mock node
    self.pa._parse_ast = MagicMock()
    self.pa._parse_ast(node)
    self.pa._parse_ast.assert_called_with(node)
```

Integration tests were then utilized to check how different classes worked together. For example: The ‘TestHybridAlgorithmIntegration’ tested how the HybridAlgorithm class manages its components i.e. CuckooSearch and SImulatedAnnealing. It also checks how well the interactions of these are orchestrated such as generating test cases, Performing hybrid Optimizations and evaluating the test cases.

```python
class TestHybridAlgorithmIntegration(unittest.TestCase):
    def setUp(self):
        # Mock user input code as a string of Python code.
        user_input_code = """..."""
        # Initialize the HybridAlgorithm with the mock user input code.
        self.hybrid_algorithm = HybridAlgorithm(user_input_code=user_input_code)
        # Assuming CuckooSearch and SimulatedAnnealing are part of HybridAlgorithm
        self.hybrid_algorithm.cuckoo = MagicMock(spec=CuckooSearch)
        self.hybrid_algorithm.annealing = MagicMock(spec=SimulatedAnnealing)
    def test_init(self):
        # Test if HybridAlgorithm is initialized correctly.
        self.assertIsInstance(self.hybrid_algorithm, HybridAlgorithm)

    def test_generate_test_cases(self):
        # Test if test cases are generated correctly.
        self.hybrid_algorithm.generate_test_cases = MagicMock(return_value=['test_case1', 'test_case2'])
        test_cases = self.hybrid_algorithm.generate_test_cases(self.hybrid_algorithm.user_input_code)
        self.hybrid_algorithm.generate_test_cases.assert_called_with(self.hybrid_algorithm.user_input_code)
        self.assertEqual(test_cases, second: ['test_case1', 'test_case2'])

    def test_hybrid_optimization(self):
        # Test the hybrid optimization process.
        self.hybrid_algorithm.hybrid_optimization = MagicMock(return_value=('optimized_solution', ['fitness_history']))
        result = self.hybrid_algorithm.hybrid_optimization(['test_case1', 'test_case2'])
        self.hybrid_algorithm.hybrid_optimization.assert_called_with(['test_case1', 'test_case2'])
        self.assertEqual(result, second: ('optimized_solution', ['fitness_history']))

    def test_evaluate_test_cases(self):
        # Test evaluation of test cases.
        self.hybrid_algorithm.evaluate_test_cases = MagicMock(return_value=(0.1, 'optimized_solution', ['fitness_data'], 5))
        score, solution, fitness_data, convergence_gen = self.hybrid_algorithm.evaluate_test_cases(['test_case1', 'test_case2'])
        self.hybrid_algorithm.evaluate_test_cases.assert_called_with(['test_case1', 'test_case2'])
        self.assertEqual(score, second: 0.1)
        self.assertEqual(solution, second: 'optimized_solution')
        self.assertEqual(fitness_data, second: ['fitness_data'])
        self.assertEqual(convergence_gen, second: 5)
```

With the test case generator, we were able to design unit tests that cover various methods and scenarios including initializations, behavior under expected conditions and interactions with mocked objects. In the end, all the tests were passed.

```
============================= test session starts ==============================
collecting ... collected 25 items

unittest_for_Hybrid_SA_algorithm_test.py::TestCuckooSearch::test_generate_initial_population PASSED [  4%]
unittest_for_Hybrid_SA_algorithm_test.py::TestCuckooSearch::test_get_fitness PASSED [  8%]
unittest_for_Hybrid_SA_algorithm_test.py::TestCuckooSearch::test_init PASSED [ 12%]
unittest_for_Hybrid_SA_algorithm_test.py::TestCuckooSearch::test_levy_flight PASSED [ 16%]
unittest_for_Hybrid_SA_algorithm_test.py::TestSimulatedAnnealing::test_acceptance_probability PASSED [ 20%]
unittest_for_Hybrid_SA_algorithm_test.py::TestSimulatedAnnealing::test_anneal PASSED [ 24%]
unittest_for_Hybrid_SA_algorithm_test.py::TestSimulatedAnnealing::test_get_neighbour PASSED [ 28%]
unittest_for_Hybrid_SA_algorithm_test.py::TestSimulatedAnnealing::test_initial_solution PASSED [ 32%]
unittest_for_Hybrid_SA_algorithm_test.py::TestHybridAlgorithmIntegration::test_evaluate_test_cases PASSED [ 36%]
unittest_for_Hybrid_SA_algorithm_test.py::TestHybridAlgorithmIntegration::test_generate_test_cases PASSED [ 40%]
unittest_for_Hybrid_SA_algorithm_test.py::TestHybridAlgorithmIntegration::test_hybrid_optimization PASSED [ 44%]
unittest_for_Hybrid_SA_algorithm_test.py::TestHybridAlgorithmIntegration::test_init PASSED [ 48%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_define_expected_outputs PASSED [ 52%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_extract_structure PASSED [ 56%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_generate_coverage_report PASSED [ 60%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_generate_negative_input PASSED [ 64%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_generate_test_case_for_method PASSED [ 68%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_get_default_value PASSED [ 72%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_identify_test_inputs PASSED [ 76%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_identify_test_scenarios PASSED [ 80%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_init PASSED [ 84%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_mark_covered PASSED [ 88%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_parse_ast PASSED [ 92%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_parse_method_parameters PASSED [ 96%]
unittest_for_Hybrid_SA_algorithm_test.py::TestProgramAnalysis::test_process_class_definition PASSED [100%]

============================== 25 passed in 0.40s ==============================
```

Reason for choosing the programs for testing.

As discussed in the main body of the report, eight programs were chosen to generate test cases for. This is because the diversity of the python programs allowed for a variety of test scenarios to evaluate the effectiveness and adaptability of the hybrid CS-SA algorithm. The programs chosen include a spectrum of applications, from data-driven models with the “predicting forest fires with PyTorch,” to classic algorithms such as the triangle classification problem and even a GUI application that visualizes the binary search tree.

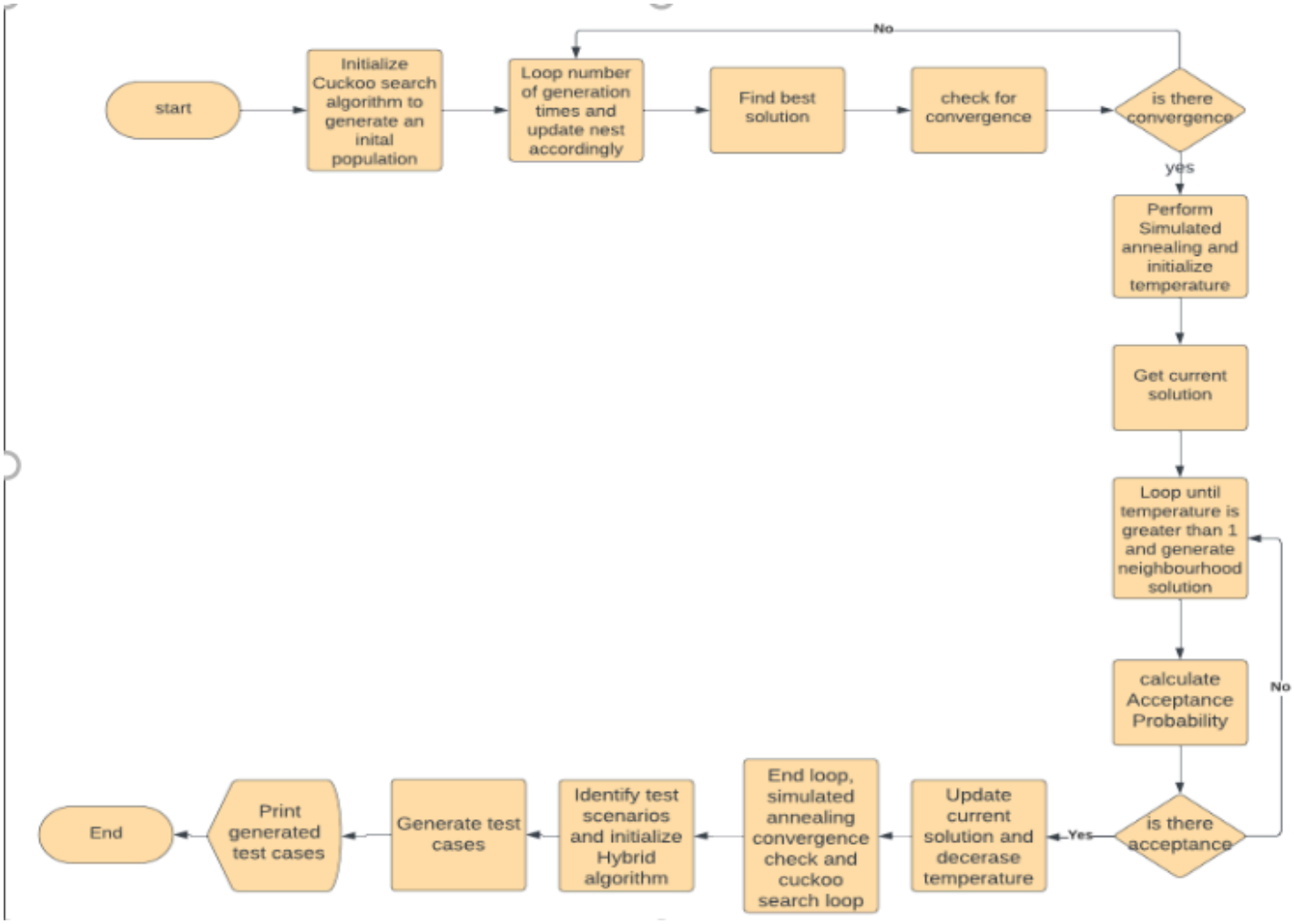
Most of these python programs were sourced from Github which is an open-source platform. Testing these programs enabled the assessment of the hybrid CS-SA algorithm against the other metaheuristic algorithms to provide quantitative data to determine the hybrid’s performance. By

testing the algorithm against these benchmark programs, we were able to uncover insights into how OOP testing can be enhanced with tailored metaheuristic strategies.

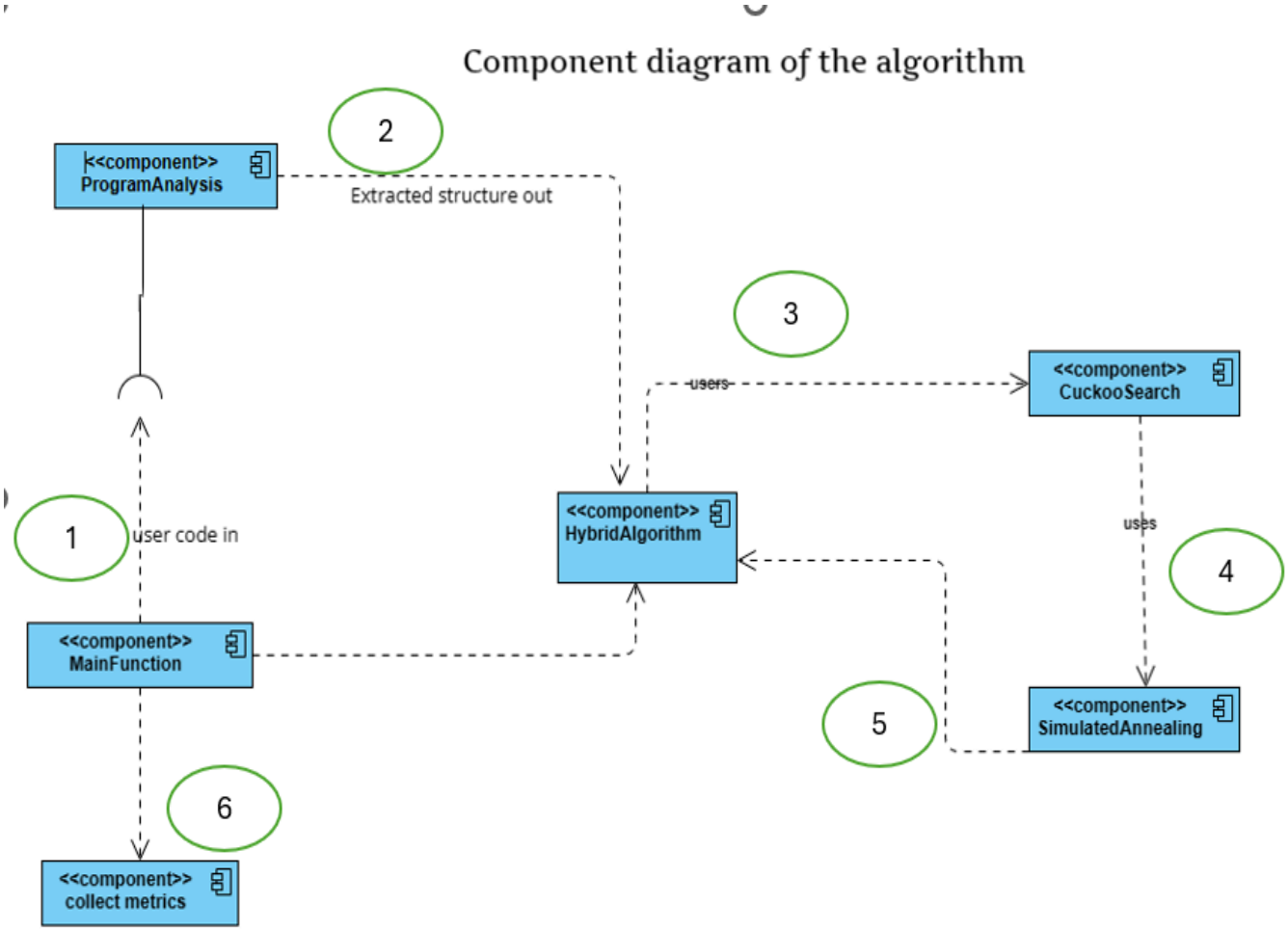
APPENDIX E

Program's flow chart

To supplement the design of the Hybrid metaheuristic algorithm presented in this paper, a flowchart and component diagram have been designed. This allows for an easy view of the algorithm's flow and architecture in relation to the 4 + 1 architecture view model. The code of this application can be found in the zip folder accompanying this report.



Program's Component diagram



References

Al-Fedaghi, Sabah. “Existential Ontology and Thinging Modeling in Software Engineering.” *SSRN Electronic Journal*, 2018, https://doi.org/10.2139/ssrn.3360382.

ALIRZASLHI. “OOP in Python (Part 2),” Kaggle, Aug. 2023, www.kaggle.com/discussions/getting-started/428640. Accessed 25 Mar. 2024.

Alkhateeb, F. and Abed-alguni, B.H. (2019) ‘A Hybrid Cuckoo Search and Simulated Annealing Algorithm’, Journal of Intelligent Systems, 28(4), pp. 683–698. Available at: https://doi.org/10.1515/jisys-2017-0268.

Al-Masri, O. and Al-Sorori, W.A., 2022. Object-Oriented Test Case Generation Using Teaching Learning-Based Optimization (TLBO) Algorithm. *IEEE Access*, *10*, pp.110879-110888.

Arianne, ariannedee . “OOP-Python,” GitHub, 7 Feb. 2024, github.com/ariannedee/oop-python/blob/main/Refactoring/refactoring_4_encapsulation/game.py. Accessed 25 Mar. 2024.

Ashwin Urdhwareshe. “Object-Oriented Programming and Its Concepts.” International Journal of Innovation and Scientific Research, vol. 26, no. 1, 2 Aug. 2016, pp. 1–6. Accessed 14 Mar. 2024.

Cico, Orges, et al. “AI-Assisted Software Engineering: A Tertiary Study.” *12th Mediterranean Conference on Embedded Computing (MECO),*6 June 2023, https://doi.org/10.1109/meco58584.2023.10154972. Accessed 11 Apr. 2024.

Efan, et al. “A Systematic Literature Review of Teaching and Learning on Object-Oriented Programming Course.” *International Journal of Information and Education Technology*, vol. 13, no. 2, 1 Jan. 2023, pp. 302–312, https://doi.org/10.18178/ijiet.2023.13.2.1808.

Gunji, Balamurali. “Flowchart of the Basic Cuckoo Search Algorithm,” Research Gate, Dec 2018, www.researchgate.net/publication/329987127_Optimal_path_planning_of_mobile_robot_using_hybrid_cuckoo-bat_algorithm_in_assorted_environment. Accessed 25 Mar. 2024.

Jha, Nisha, and Rashmi Popli. “Artificial Intelligence for Software Testing-Perspectives and Practices.” *2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, July 2021, https://doi.org/10.1109/ccict53244.2021.00075.

Khamprapai, Wanida, et al. “Performance of Enhanced Multiple-Searching Genetic Algorithm for Test Case Generation in Software Testing.” *Mathematics*, vol. 9, no. 15, 27 July 2021, p. 1779, https://doi.org/10.3390/math9151779. Accessed 8 Nov. 2022.

Khari, M., & Kumar, P. (2017). An effective meta-heuristic cuckoo search algorithm for test suite optimization. Informatica, 41(3).

Kumar, Gagan, and Vinay Chopra. “Hybrid Approach for Automated Test Data Generation.” *Journal of ICT Standardization*, 2 Dec. 2022, https://doi.org/10.13052/jicts2245-800x.1043. Accessed 22 Dec. 2022.

Li, Dongcheng, et al. “Improving Search-Based Test Case Generation with Local Search Using Adaptive Simulated Annealing and Dynamic Symbolic Execution.” *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, 1 Aug. 2022, https://doi.org/10.1109/dsa56465.2022.00047. Accessed 2 Apr. 2024.

Liu, P. and Zhang, S. (2021) 'A novel cuckoo search algorithm and its Application', *Open Journal of Applied Sciences*, 11(09), pp. 1071–1081. doi:10.4236/ojapps.2021.119079.

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2019). A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education*, 62(2), 77-90. doi:10.1109/TE.2018.2864133 .

M. Ghoreshi, and H Haghighi. "Object Coverage Criteria for Supporting Object-Oriented Testing." *Software Quality Journal*, vol. 31, no. 4, 30 June 2023, pp. 1369–1414, https://doi.org/10.1007/s11219-023-09643-3. Accessed 4 Feb. 2024.

Nwokoro, I. *et al.* (2021) 'Object-oriented programming and software development paradigm', *Academia Letters* [Preprint]. doi:10.20935/al3443.

Panda, M. *et al.* (2020) 'Test suit generation for object-oriented programs: A hybrid Firefly and differential evolution approach', *IEEE Access*, 8, pp. 179167–179188. doi:10.1109/access.2020.3026911.

Panda, M., Sarangi, P.P. and Dash, S. (2015) 'Automatic Test Data Generation using metaheuristic cuckoo search algorithm', *International Journal of Knowledge Discovery in Bioinformatics*, 5(2), pp. 16–29. doi:10.4018/ijkdb.2015070102.

Panda, Madhumita, and Sujata Dash. "Test-Case Generation for Model-Based Testing of Object-Oriented Programs." *Services and Business Process Reengineering*, 1 Jan. 2020, pp. 53–77, https://doi.org/10.1007/978-981-15-2455-4_3. Accessed 11 Apr. 2024.

Panichella, Annibale, et al. "Automated Test Case Generation as a Many-Objective Optimization Problem with Dynamic Selection of the Targets." *IEEE Transactions on Software Engineering*, vol. 44, no. 2, 1 Feb. 2018, pp. 122–158, orbilu.uni.lu/bitstream/10993/30978/1/tse2017.pdf, https://doi.org/10.1109/tse.2017.2663435. Accessed 24 May 2019.

Potluri, S. et al. (2022) 'Optimized test coverage with hybrid particle swarm bee colony and Firefly Cuckoo search algorithms in model-based software testing', 2022 First International Conference on Artificial Intelligence Trends and Pattern Recognition (ICAITPR) [Preprint]. doi:10.1109/icaitpr51569.2022.9844208.

Rajabioun, R. (2011). Cuckoo optimization algorithm. Applied Soft Computing, 11(8), 5508-5518. doi: 10.1016/j.asoc.2011.05.008

Raut, R. S. (2020). Research Paper on Object-Oriented Programming (OOP). International Research Journal of Engineering and Technology (IRJET), 7(10), 1453.

Shakya, S., & Smys, S. (2020). Reliable automated software testing through hybrid optimization algorithm. Journal of Ubiquitous Computing and Communication Technologies (UCCT), 2(03), 126-135.

Singh, Nehul, et al. "Object Oriented Programming: Concepts, Limitations and Application Trends." *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, 22 Oct. 2021, https://doi.org/10.1109/iscon52037.2021.9702463. Accessed 2 May 2022.

Sneha, Karuturi, and Gowda M Malle. "Research on Software Testing Techniques and Software Automation Testing Tools." 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS), Aug. 2017, ieeexplore.ieee.org/abstract/document/8389562/, https://doi.org/10.1109/icecds.2017.8389562.

Srivastava, Deepali. "Multiple Inheritance in Python," GitHub, 10 Feb. 2022, github.com/Deepali-Srivastava/object-oriented-programming-in-python/blob/main/lectures-code/multiple-inheritance-2.py. Accessed 25 Mar. 2024.

Wang, Shuai, et al. “OOP: Object-Oriented Programming Evaluation Benchmark for Large Language Models.” *ArXiv (Cornell University)*, 12 Jan. 2024, https://doi.org/10.48550/arxiv.2401.06628. Accessed 2 Apr. 2024.

Zhou, Ai-Hua, et al. “Traveling-Salesman-Problem Algorithm Based on Simulated Annealing and Gene-Expression Programming,” Research Gate, 25 Dec. 2018, www.researchgate.net/publication/329917885_Traveling-Salesman-Problem_Algorithm_Based_on_Simulated_Annealing_and_Gene-Expression_Programming/figures?lo=1. Accessed 25 Mar. 2024