



# **ENHANCING MUTATION TESTING: SEARCH-BASED OPTIMIZATION TO IMPROVE TESTING QUALITY**

**SERHAT UZUNBAYIR**

Thesis for Ph.D. Program in Computer Engineering

Graduate School

Izmir University of Economics

İzmir

2024

# **ENHANCING MUTATION TESTING: SEARCH-BASED OPTIMIZATION TO IMPROVE TESTING QUALITY**



**SERHAT UZUNBAYIR**

THESIS ADVISOR: ASST. PROF. DR. KAAAN KURTEL

A Ph.D Thesis

Submitted to

the Graduate School of Izmir University of Economics

the Department of Engineering

İzmir

2024

## ETHICAL DECLARATION

I hereby declare that I am the sole author of this thesis and that I have conducted my work in accordance with academic rules and ethical behaviour at every stage from the planning of the thesis to its defence. I confirm that I have cited all ideas, information and findings that are not specific to my study, as required by the code of ethical behaviour, and that all statements not cited are my own.

Name, Surname:

Serhat UZUNBAYIR

Date:

12.01.2024

# ABSTRACT

## ENHANCING MUTATION TESTING: SEARCH-BASED OPTIMIZATION TO IMPROVE TESTING QUALITY

Uzunbayır, Serhat

Ph.D. Program in Computer Engineering

Advisor: Asst. Prof. Dr. Kaan KURTEL

January, 2024

Software testing is a crucial phase in the software development lifecycle. Without thorough testing activities, the product may be ineffective or unreliable. Because any changes in the source code require the re-execution of test suites, it is important that the code coverage aligns with the requirements of the project. Mutation testing, a fault-oriented white-box unit testing technique, is the process that enables the evaluation of the quality of test suites and identify weaknesses in test procedures. Although effective, the application of mutation testing faces a range of challenges: high costs, the presence of equivalent mutants, and the redundancies in test suites. This study therefore aims to explore the progress of mutation testing and its position within software engineering, tracing its evolution from classic methodologies to the integration of

artificial intelligence and innovative search-based hybrid techniques. This involves delving into the traditional principles of mutation testing, its problems, providing an in-depth analysis of mutation testing tools, particularly for C#, and evaluating their functionalities. This leads into the introduction of a novel hybrid method, combining two meta-heuristics: genetic algorithms and ant colony optimization, the aim of which is to optimize the test suite reduction problem in search-based mutation testing. The problem of equivalent mutants is also addressed by utilizing genetic algorithms to enhance the efficiency of higher-order mutation testing. Consequently, this research contributes to mutation testing in solving problems mentioned above. It proposes innovative approaches that integrate advanced computational techniques, and thus paving the way for more effective software quality assurance practices.

Keywords: software testing, mutation testing, search-based mutation, genetic algorithms, ant colony optimization, meta-heuristics.

# ÖZET

## MUTASYON TESTİNİ GELİŞTİRME: TEST KALİTESİNİN İYİLEŞTİRİLMESİ İÇİN ARAMA TABANLI OPTİMİZASYON

Uzunbayır, Serhat

Bilgisayar Mühendisliği Doktora Programı

Tez Danışmanı: Dr. Kaan KURTEL

Ocak, 2024

Yazılım testi, yazılım geliştirme yaşam döngüsünün önemli bir aşamasıdır. Kapsamlı test faaliyetleri olmadan ortaya çıkan ürün kullanışsız veya güvenilmezdir. Kaynak kodundaki değişiklikler test paketlerinin yeniden yürütülmesini gerektirdiği için, kod kapsamı projenin gereksinimleriyle uyumlu olmalıdır. Hata odaklı bir şeffaf kutu birim test tekniği olan mutasyon testi, test paketlerinin kalitesinin değerlendirilmesi ve test prosedürlerindeki zayıflıkların belirlenmesi için kullanılır. Mutasyon testinin uygulanması her ne kadar etkili olsa da, yüksek maliyetler, eşdeğer mutantların varlığı ve test paketlerindeki test fazlalıkları nedenlerinden dolayı uygulamada zorluklar göstermektedir. Bu çalışmada, yazılım mühendisliğinde mutasyon testi araştırılmış, klasik metodolojilerden yapay zeka ve yenilikçi hibrit tekniklerin entegrasyonuna

kadar gelişiminin izini sürülmüştür. Mutasyon testinin geleneksel ilkeleri ve problemleri incelenmiş ve C# programlama dili için mutasyon test araçlarının derinlemesine analizi yapılmıştır. Test grubu azaltma problemini optimize etmek için iki metasezgisel yöntemi (genetik algoritmalar ve karınca kolonisi optimizasyonu) birleştiren arama tabanlı mutasyon testi için yeni bir hibrit yöntem sunulmuştur. Eşdeğer mutantlar sorunu, daha üst düzey mutasyon testlerinin verimliliğini artırmak için genetik algoritmalar kullanılarak ele alınmıştır. Sonuç olarak bu çalışma, test kalitesinin iyileştirilmesi için mutasyon testine katkı sağlamaktadır. Gelişmiş hesaplama tekniklerini entegre eden, böylece daha etkili, verimli ve gelişmiş yazılım kalite güvence uygulamalarının önünü açan yaklaşımlar önermiştir.

Anahtar Kelimeler: yazılım testi, mutasyon testi, arama tabanlı mutasyon, genetik algoritmalar, karınca koloni optimizasyonu, metasezgiseller.

This thesis work is dedicated to my precious wife Cemre...





## ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor, Asst. Prof. Dr. Kaan KURTEL, for his unwavering support, guidance, and patience throughout my research. His insightful feedback and constructive criticism have been invaluable in shaping the direction of this thesis.

I would like to thank my committee members, Prof. Dr. Şaban EREN and Prof. Dr. Hüseyin AKCAN, for their time and expertise in reviewing and providing feedback on my work. I would like to thank Prof. Dr. Cem EVRENDİLEK and Asst. Prof. Dr. Serap ŞAHİN for their valuable ideas, contributions, and feedback during my thesis committee meetings. I would like to extend my special thanks to Assoc. Prof. Dr. Senem KUMOVA METİN, Assoc. Prof. Dr. Kaya OĞUZ, and Assoc. Prof. Dr. Zeynep Nihan BERBERLER for their belief in my ability to complete this study. They were consistently supportive and available to help whenever I needed them.

I would like to express my heartfelt thanks to my beloved wife, Cemre UZUN-BAYIR, who inspires me to become a better person every day. Her unwavering support has been my guiding light; she never gave up on me, even in my lowest moments. Her strength and patience have been my rock, and without her, I would be truly lost. Her love and belief in me have made all the difference in my journey, and for that, I am eternally grateful.

I would like to express my deepest gratitude to Dr. Erenus YILDIZ, whose guidance, contributions, and support have been invaluable, particularly during the most challenging times. His wisdom and encouragement have been fundamental to my progress.

I am also immensely grateful to Dr. Erdem OKUR, a constant source of support and motivation. He was always there when I felt lost, pushing me to take decisive action and inspiring me to move forward with confidence.

Additionally, my sincere thanks go to my colleagues Çınar GEDİZLİOĞLU, Melek Büşra TEMUÇİN, Duygu GEÇKİN, and Hande AKA UYMAZ. Their unwavering encouragement and support have been pivotal to my research activities.

Finally, I would like to thank to my parents Gül UZUNBAYIR, Ömer UZUNBAYIR, and my sister Gülşah UZUNBAYIR for their love and endless support. Also my friends, especially Levent Tolga EREN, for their support and encouragement throughout this journey. Their unwavering faith in me has been a constant source of motivation and inspiration.

Thank you all for your contributions to this work.



## TABLE OF CONTENTS

ABSTRACT.....	iv
ÖZET .....	vi
DEDICATION .....	viii
ACKNOWLEDGEMENT .....	ix
TABLE OF CONTENTS.....	xi
LIST OF TABLES .....	xv
LIST OF FIGURES .....	xvii
LIST OF ABBREVIATIONS .....	xviii
CHAPTER 1: INTRODUCTION.....	1
1.1. <i>Thesis Statement</i> .....	1
1.2. <i>What is Mutation Testing?</i> .....	1
1.3. <i>Problems of Mutation Testing</i> .....	4
1.4. <i>Research Questions</i> .....	6
1.5. <i>Contributions</i> .....	7
1.6. <i>Organization of the Thesis</i> .....	8
1.7. <i>List of Publications</i> .....	9
CHAPTER 2: PRELIMINARIES AND RELATED WORK.....	11
2.1. <i>Fundamental Principles of Mutation Testing</i> .....	11
2.2. <i>Mutation Analysis</i> .....	12
2.2.1. <i>Mutation Process</i> .....	12
2.2.2. <i>Cost of Mutations</i> .....	17
2.3. <i>Cost Reduction Techniques Literature Review</i> .....	18
2.3.1. <i>Computational Cost Reduction Techniques</i> .....	19
2.3.2. <i>Manual Cost Reduction</i> .....	31
2.4. <i>Challenges and Current Trends for Mutation Testing</i> .....	37
2.4.1. <i>Challenges and Interests of Mutation Testing</i> .....	37
2.4.2. <i>A Hot Topic: Artificial Intelligence Supported Mutation Testing</i>	39
2.4.3. <i>State of the Art Models for Mutation Testing Using AI</i> .....	49

2.5. Conclusion.....	50
<b>CHAPTER 3: AN ANALYSIS ON MUTATION TESTING TOOLS .....</b>	<b>51</b>
3.1. Mutation Testing Tools for Different Programming Languages .....	51
3.2. Mutation Operators for C# .....	53
3.3. Mutation Testing Tools for C# .....	55
3.3.1. Nester.....	55
3.3.2. Stryker.....	55
3.3.3. NinjaTurtles .....	57
3.3.4. VisualMutator .....	57
3.3.5. PexMutator .....	58
3.3.6. CREAM .....	58
3.4. An Analysis of Mutation Tools for C# Based On Tool Characteristics....	59
3.5. A Case Study: Cross-Evaluation of the Tools .....	61
3.5.1. Methodology .....	62
3.5.2. Research Questions .....	62
3.5.3. Subject Programs .....	63
3.5.4. Results.....	63
3.6. Conclusion.....	67
<b>CHAPTER 4: EVOCOLONY: A HYBRID APPROACH .....</b>	<b>69</b>
4.1. Search-Based Mutation Testing.....	69
4.2. Test Case Reduction Problem .....	72
4.3. Genetic Algorithms .....	76
4.4. Ant Colony Optimization.....	78
4.5. Methodology .....	81
4.5.1. Research Questions .....	81
4.5.2. EvoColony: A Hybrid Approach to Search-Based Mutation Testing .....	81
4.6. Experimental Design.....	87
4.6.1. Test Environment.....	87
4.6.2. Test Data.....	89
4.6.3. Benchmark Algorithms .....	89

4.6.4. <i>Results and Evaluation</i> .....	90
4.7. <i>Conclusion</i> .....	99
CHAPTER 5: LEVERAGING MUTANTS IN HIGHER-ORDER.....	100
5.1. <i>First-Order and Equivalent Mutants</i> .....	100
5.2. <i>Higher-Order Mutation Testing</i> .....	101
5.3. <i>Methodology</i> .....	103
5.3.1. <i>A Genetic Algorithm for Higher-Order Mutant Generation</i> .....	104
5.3.2. <i>A Random Search Algorithm for Higher-Order Mutant Generation</i> .....	110
5.3.3. <i>Research Questions</i> .....	111
5.4. <i>Experimental Design</i> .....	111
5.4.1. <i>Test Environment</i> .....	112
5.4.2. <i>Subject Programs</i> .....	113
5.4.3. <i>Genetic Algorithm Parameter Settings</i> .....	114
5.5. <i>Results</i> .....	115
5.6. <i>Conclusion</i> .....	117
CHAPTER 6: CONCLUSION .....	119
6.1. <i>Summary</i> .....	119
6.2. <i>Future Work</i> .....	120
6.3. <i>Final Remarks</i> .....	121
REFERENCES .....	134
CURRICULUM VITAE.....	135

## LIST OF TABLES

Table 1.	An original program and its mutant.....	13
Table 2.	Mutation testing results of the example.....	17
Table 3.	An example of higher-order mutation. ....	23
Table 4.	An example of equivalence mutation. ....	31
Table 5.	Mutation testing tools. ....	52
Table 6.	Some traditional mutation operators (Jia and Harman, 2010). ....	53
Table 7.	Some mutation operators for C# (Derezińska, 2006).....	54
Table 8.	Feature comparison of mutation testing tools for C#.....	60
Table 9.	Subject programs for the case study. ....	63
Table 10.	NinjaTurtles vs. other tools. ....	64
Table 11.	Stryker vs. other tools.....	64
Table 12.	Nester vs. other tools.....	64
Table 13.	VisualMutator vs. other tools.....	65
Table 14.	PexMutator vs. other tools. ....	65
Table 15.	Relative test suite evaluation.....	65
Table 16.	Average results for mutation score.....	66
Table 17.	Average results for disjoint mutant sets.....	66
Table 18.	Reference mutant set. ....	67
Table 19.	Tool rankings. ....	67
Table 20.	Subject programs details.....	89
Table 21.	Genetic and ant colony parameters of EvoColony.....	91
Table 22.	EvoColony results.....	92
Table 23.	Comparative test results. ....	93
Table 24.	An original program and its first-order mutant.....	101
Table 25.	An original program and its an equivalent mutant. ....	101
Table 26.	Original program and its higher-order mutant.....	102
Table 27.	Subject programs. ....	114
Table 28.	Ratio of generated equivalent mutants. ....	116

Table 29. Execution cost of different selection strategies..... 116

Table 30. Percentage of the generated higher-order mutants from each mutation order..... 117



## LIST OF FIGURES

Figure 1. The place of mutation testing in software development.....	3
Figure 2. Thesis structure and preparation process. ....	9
Figure 3. Traditional mutation process. ....	13
Figure 4. Mutation testing cost reduction techniques. ....	20
Figure 5. Compiler-based run-time optimization.....	29
Figure 6. Compiler-integrated run-time optimization. ....	29
Figure 7. Mutant schemata run-time optimization. ....	30
Figure 8. Byte-code translation run-time optimization. ....	30
Figure 9. Utilizing AI in mutation testing.....	41
Figure 10. Meta-heuristic optimization algorithms categorization with examples..	71
Figure 11. Test suite reduction process.....	72
Figure 12. Test case reduction using detection matrix.....	74
Figure 13. Test case reduction using detection matrix (cont'd).....	75
Figure 14. Traditional genetic algorithm steps. ....	78
Figure 15. Traditional ant colony algorithm steps. ....	80
Figure 16. The EvoColony algorithm steps. ....	85
Figure 17. Test data setup.....	88
Figure 18. Reduced rest suite size comparison. ....	94
Figure 19. Run-time performance comparison for BubbleSort.....	95
Figure 20. Run-time performance comparison for Calendar. ....	96
Figure 21. Run-time performance comparison for TriangleType. ....	96
Figure 22. Run-time performance comparison for ArrayOperations.....	96
Figure 23. Run-time performance comparison for TemperatureConverter. ....	97
Figure 24. Run-time performance comparison for QuadraticSolver. ....	97
Figure 25. Run-time performance comparison for HashTable.....	97
Figure 26. Run-time performance comparison for BinarySearch.....	98
Figure 27. Run-time performance comparison for BankAccount.....	98
Figure 28. Run-time performance comparison for AutoDoor.....	98



Figure 29. Flow chart of the proposed genetic algorithm. .... 109  
Figure 30. Test environment to create first-order mutants. .... 112  
Figure 31. Experiment details to create higher-order mutants. .... 113



## LIST OF ABBREVIATIONS

ACO	: Ant Colony Optimization
AI	: Artificial Intelligence
ANN	: Artificial Neural Networks
CI/CD	: Continuous Integration & Continuous Delivery
CISQ	: Information and Software Quality Consortium
CNN	: Convolutional Neural Network
DL	: Deep Learning
EC	: Evolutionary Computation
GA	: Genetic Algorithm
GA-SP	: Genetic Algorithm Single-Point Crossover
GA-DP	: Genetic Algorithm Double-Point Crossover
GCN	: Graph Convolutional Network
HTML	: Hypertext Mark-up Language
JSON	: JavaScript Object Notation
KLOC	: Thousands of Lines of Code
LINQ	: Language Integrated Query
LOC	: Lines of Code
ML	: Machine Learning
MLP	: Multilayer Perceptron
NSGA-II	: Non-Dominated Sorting Genetic Algorithm II
RS	: Random Search
RQ	: Research Question
SDLC	: Software Development Life Cycle
SHOM	: Strong Higher-Order Mutant
SOTA	: State of the Art
TDD	: Test Driven Development
XML	: Extensible Markup Language

# CHAPTER 1: INTRODUCTION

## *1.1. Thesis Statement*

The objective of this thesis is to enhance the effectiveness and applicability of mutation testing. The study seeks to optimize mutation testing methodologies by focusing on three key areas: evaluating mutation testing tools for C#, reducing the number of test cases in a test suite using two meta-heuristic approaches in search-based mutation testing, and employing genetic algorithms for higher-order mutation testing to deal with equivalent mutants.

## *1.2. What is Mutation Testing?*

In the software development lifecycle, testing emerges as a pivotal stage for evaluating software quality. It is a strategic process integral to the development and maintenance of reliable products. The objective of software testing is not to demonstrate that a system is free of errors, but rather to identify and reveal faults within the system. To do that, various testing strategies both manual to automated methods, are employed in software testing. Currently, there is a noticeable shift towards automation in testing activities (Adams and McIntosh, 2016), which consequently increases the overall costs. Statista reported that in its fiscal year of 2023, Apple Inc. invested an unprecedented 29.92 billion U.S. dollars in research and development, which include testing, marking an increase of approximately 3.5 billion dollars over the previous year (Laricchia, 2023). Furthermore, another report in 2022 emphasized that according to the Information and Software Quality Consortium (CISQ), the annual cost of poor software quality in the United States grown to be no less than \$2.41 trillion (Krasner, 2022). Consequently, the volume of testing activities has also increased. A Google report indicates that on an average day, there are approximately 13K projects, 800K builds, and 150 million tests conducted (Memon et al., 2017). These real-world phenomena demonstrate that software testing is crucial and represents a significant financial investment involved in ensuring software quality.

The widespread adoption of software products, compounded by the expanding list of functional requirements and specifications, presents a major challenge in the process of thorough testing. Traditional methods such as exhaustive testing, which attempts to cover all possible data combinations, are becoming increasingly impractical due to the endless expansion in the number of potential test inputs. This has led to the adoption of more sophisticated testing strategies, such as code coverage criteria such as condition, branch and statement coverage, which are essential in assessing the confidence level of a software system. Nonetheless, achieving 100% code coverage is a demanding task, requiring significant computational resources as well as considerable human effort. This situation arises for three main reasons: firstly, certain requirements have different coverage criteria; secondly, the test case generation process cannot be fully automated, and finally, there are some test cases that require human involvement (Kintis, 2016).

Mutation testing, a type of fault-based white-box testing first introduced in the late 1970s, has gained an importance for its effectiveness in identifying faults and improving test suite quality. Figure 1 depicts the position of mutation testing in the scope of software development. This method involves mutation analysis which modifies the original program code through a set of predefined mutation operators, and creates an environment that mimics potential programmer errors. At the end of this procedure, a unique metric, the mutation adequacy score, is calculated to measure its effectiveness. In an empirical study conducted by Chekam et al. (2017), mutation testing was compared with statement and branch coverage, leading to the conclusion that mutation testing was the more effective of the two in detecting faults within a test suite. Additionally, Chen et al. (2020) explored how the size of the test set affects its efficacy and suggested a method for controlling test size during the evaluation of test adequacy. These findings highlight the necessity of minimizing the test suite size while maintaining its effectiveness.

Mutation analysis has long been a focus in software engineering research (Usaola and Mateo, 2010), and has found application in various industry projects (Jia and Harman, 2010; Papadakis et al., 2019). Large companies, including Google and Facebook, are investigating the application of mutation testing to their products, as evidenced by studies such as Petrović et al. (2021), and Beller et al. (2021). However,

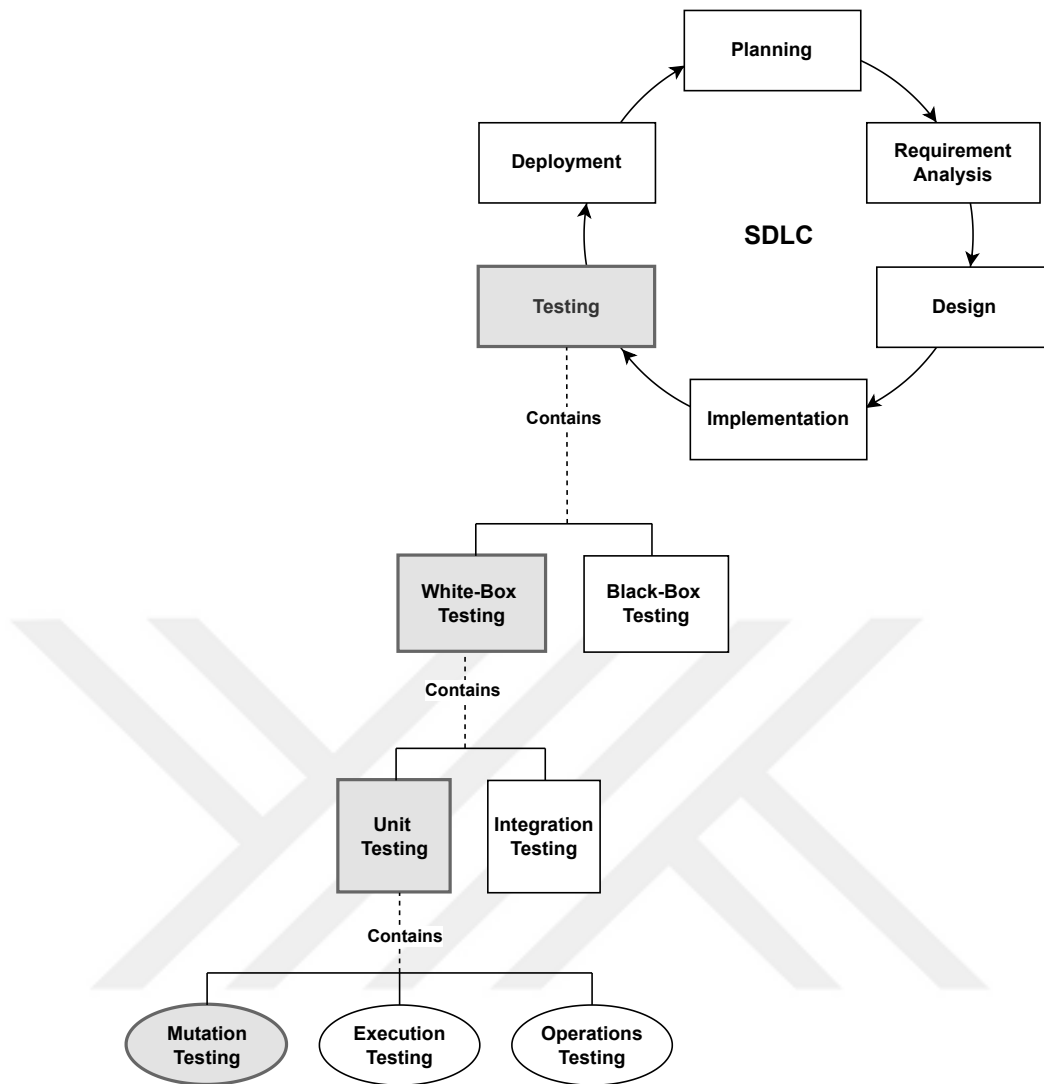


Figure 1. The place of mutation testing in software development.

such analysis is costly, complex, and time-consuming, due to the needs for high computational resources, stemming from the large number of mutants. Therefore, this field faces a wide range of testing challenges. These include, assessing the quality of test suites (Nayyar et al., 2015), reducing the number of test cases (Noemmer and Haas, 2020; Palomo-Lozano et al., 2018), identifying faults within programs (Papadakis and Le Traon, 2014; Pearson et al., 2017), comparing test coverage criteria (Andrews et al., 2006), and automating mutation processes with comparative studies in widely-used programming languages such as Java (Kintis et al., 2018) and C# (Uzunbayir and Kurtel, 2019).

In recent years, the field of mutation testing has witnessed a drastic shift with the integration of artificial intelligence and meta-heuristic optimization techniques such

as genetic algorithms and ant colony optimization (Lima and Vergilio, 2018). This type of an approach is called search-based mutation testing and has the potential to automate and improve the testing process, particularly in object-oriented programming languages. Although mutation testing is an effective method for detecting faults in test suites, its widespread adoption has been hindered by its time-consuming and complex nature. This study, therefore, underscores the importance of mutation testing in modern software development but also aims to contribute to the field by exploring mutation testing tools and search-based methodologies. More details about mutation testing, principles, procedures, cost reduction techniques, and existing studies are discussed in Chapter 2.

### ***1.3. Problems of Mutation Testing***

There are many problems in the field of mutation testing. Below is a listing and brief summary of many open problems in the area of mutation testing based on the nature of the approach and recent academic activities. Some problems have been intensively studied in the last decade (Papadakis et al., 2019; Jia and Harman, 2010), whereas others have emerged more recently (Örgård, 2022; Panichella and Liem, 2021). The equivalent mutant problem is the most common, along with test case reduction and higher-order mutation. However, for all of these, there is a lack of efficient solutions and more work is needed to clarify the principles and standardize the techniques for all types of mutation analysis.

- 1. Higher-Order Mutation:** First-order mutation research has been limited since there is no clear theoretical basis for the value of mutants. Higher-order mutation produces better mutants; however, most of these are still equivalent. Generation of more valuable mutants with higher-order mutations remains a goal of research.
- 2. Equivalent Mutant Problem:** One of the main problems of mutation analysis is detection and reduction of equivalent mutants. There are many techniques to deal with the problem; however, none can solve it efficiently, and the problem is still unresolved.

- 3. Test Case Reduction for Mutation Testing:** Test case reduction in mutation testing is a process aimed at minimizing the number of test cases while maintaining or even enhancing the effectiveness of the test suite. This concept is crucial in mutation testing due to the often excessive number of mutants generated, which can make the testing process resource-intensive and time-consuming. Reducing the number of test cases can lead to more efficient testing processes without compromising the quality of the software being tested.
- 4. AI Supported Mutation Testing:** Following recent trends and advances in AI areas such as artificial neural networks, machine learning, and deep learning, mutation testing problems tend to show more meaningful results, as problems can now be resolved with AI support.
- 5. Search-Based Mutation Testing:** Using meta-heuristic optimization techniques such as ant colony optimization, particle swarm optimization, and others in the context of mutation testing, defining fitness functions, test data generation, and introducing mutation operators create a potential research area to be further explored.
- 6. Mutation Tools Comparison and Development:** Comparative studies for the existing mutation testing tools are limited to a few programming languages. Additionally, many new programming languages lack their own mutation testing tools, and existing tools still need some adjustments, updates, or improvements, especially considering equivalent mutants.
- 7. Continuous Mutation Testing:** The use of continuous mutation testing can help ensure that software remains robust and reliable as it evolves over time. This involves automatically running mutation tests as part of the software development pipeline. This methodology is particularly relevant in the context of agile development and DevOps practices.
- 8. Model-Based Mutation Testing:** Model-based mutation that uses state machines or graph-based models has not been studied well over the last years.

Developing efficient and scalable tools in this area could improve testing strategies, particularly for complex, state-dependent systems.

Combining two or more approaches may also improve the overall outcome, such as combining search-based mutation testing with higher-order mutations or exploring equivalent mutant issues in the model-based mutation. The use of mutation testing can improve the effectiveness of software tests, but it is important to approach it with careful consideration and planning to overcome the challenges involved to produce more meaningful results.

#### ***1.4. Research Questions***

This research seeks to explore the domain of mutation testing in greater depth. Our investigation is driven by a selection of the problems or their combinations listed in Section 1.3. Therefore, we ask the following four key research questions:

- **RQ1:** What are the existing studies, prevailing trends, challenges, and advancements in the field of mutation testing as observed in recent academic and industry practices? (**All problems**)
- **RQ2:** How can existing mutation testing tools for C# be compared in terms of their features and effectiveness? (**Problem 6**)
- **RQ3:** How can the number of test cases in a test suite be reduced, and how can this process be enhanced using meta-heuristic methods in search-based mutation testing? (**Problem 3 and 5**)
- **RQ4:** How does the implementation of genetic algorithms as a search strategy in higher-order mutation testing impact the efficiency of generating high-quality mutants, particularly in reducing the production of equivalent high-order mutants? (**Problem 1, 2, and 5**)

These questions aim to investigate mutation testing tools and propose practical applications of search-based techniques for mutation testing in software engineering. By addressing these questions, we aim to contribute to the improvement of testing quality.



## 1.5. Contributions

This thesis makes several significant contributions to the field of mutation testing:

- **Addressing RQ1:** A comprehensive review of the literature is conducted. This review updates the existing literature by examining and categorizing the latest developments in AI related to mutation testing, alongside the associated challenges and interests. Such an approach lays a solid foundation for this thesis. The identification of existing research gaps contextualizes the present study and also contributes to the broader academic discourse by outlining potential areas for future investigation.
- **Addressing RQ2:** An extensive evaluation of the most popular mutation testing tools for C# has been carried out. Since the existing literature lacks such a study specifically for C#, this analysis stands out in its thoroughness, comparing features, capabilities, and limitations of these tools. The findings are further evaluated by a case study. This contribution holds significant value for practitioners and researchers seeking the most effective tools for mutation testing in C# environments.
- **Addressing RQ3:** A novel hybrid approach, called “EvoColony”, combining two search-based techniques for test case reduction, is proposed. This approach shows an innovative design and implementation. The comparative analysis of this methodology against four traditional approaches demonstrates its effectiveness to both the theoretical and practical aspects of mutation testing.
- **Addressing RQ4:** A comparative study for genetic algorithms is presented, designed to improve the efficiency and effectiveness of higher-order mutation testing. The empirical results were obtained using four different selection methods, and these provide new insights into the optimization of mutation testing processes. This aspect particularly contribute to advancing the use of meta-heuristics in search-based higher-order mutation testing.

## 1.6. Organization of the Thesis

This research is divided into six chapters. To provide a clear understanding of how we prepared this thesis, we have presented the process in the form of a system sequence diagram, as shown in Figure 2. The organization of the thesis is outlined below:

- **Chapter 1** provides an introduction of the research topic, its significance in the field of software engineering, describes the context for mutation testing, states contributions, presents the organization of the thesis, and lists the publications from this study.
- **Chapter 2** discusses the foundational concepts relevant to mutation testing, and reviews existing literature to situate the current research within the field including latest trends, areas of interests, in particular, the innovations using sub-fields of artificial intelligence such as evolutionary computation and machine learning.
- **Chapter 3** presents existing mutation testing tools, including a detailed analysis of mutation testing tools for C#, comparing their features, capabilities, and limitations, and presents a case study for evaluation.
- **Chapter 4** introduces and elaborates on “EvoColony”, a novel hybrid approach proposed in this research for test case reduction using search-based mutation testing. The focus is on the description of the approach, its design, implementation, and effectiveness in mutation testing through a comparative analysis with four traditional approaches.
- **Chapter 5** explores the application of genetic algorithms in enhancing the efficiency and effectiveness of higher-order mutation testing for equivalent mutant reduction, presenting empirical results using four different selection methods.
- **Chapter 6** summarizes the key findings of the current research, discusses the implications, highlights the contributions to the field, and suggests directions for future research.

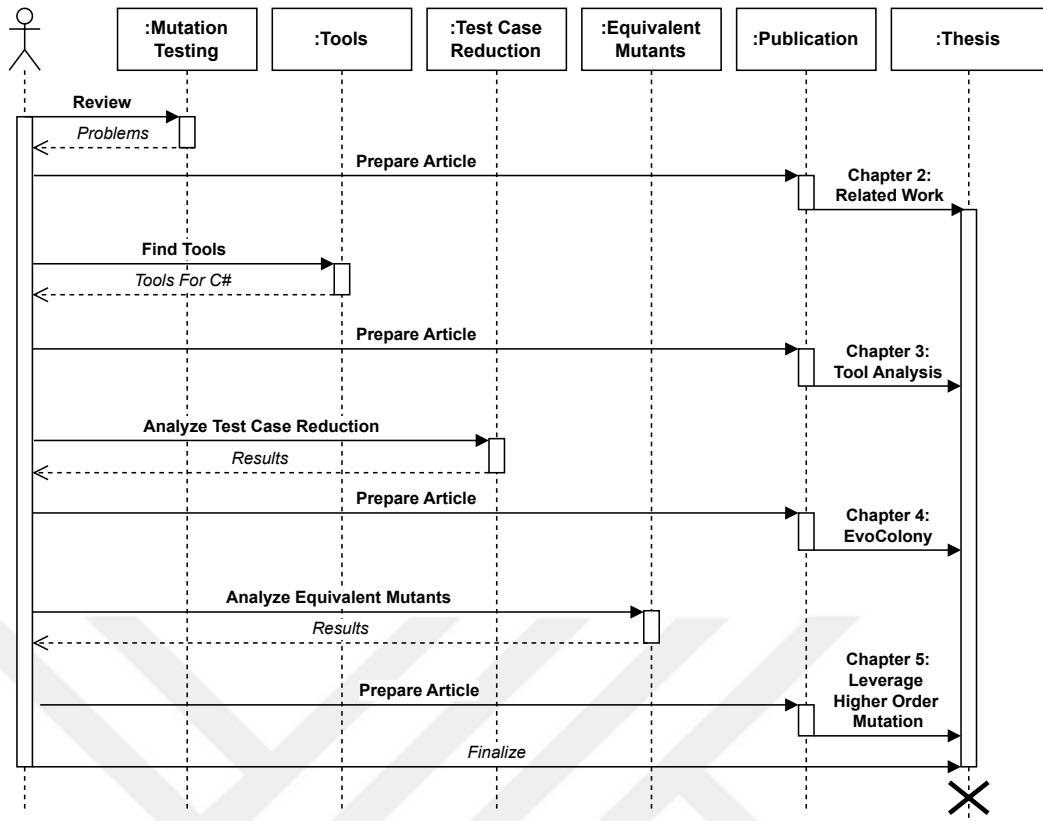


Figure 2. Thesis structure and preparation process.

### 1.7. List of Publications

The results presented in this thesis have been published, accepted, or are to be published in the conferences or journals specified in below:

1. Uzunbayir S. and Kurtel K.: An Analysis on Mutation Testing Tools for C# Programming Language, *2019 4th International Conference on Computer Science and Engineering*, **published** in September 11, 2019. (**Chapter 3**)
2. Uzunbayir S. and Kurtel K.: EvoColony: A Hybrid Approach to Search-Based Mutation Test Suite Reduction Using Genetic Algorithm and Ant Colony Optimization, *International Journal of Intelligent Systems and Applications in Engineering*, Vol. 12, No. 1, **published** in December 25, 2023. (**Chapter 4**)
3. Uzunbayir S. and Kurtel K.: Leveraging Genetic Algorithms for Efficient Search-Based Higher-Order Mutation Testing, *Computing and Informatics*, Vol. 43, No. 3, **accepted** in December 19, 2023. (**Chapter 5**)

4. Uzunbayir S. and Kurtel K.: Mutation Testing Reinvented: How Artificial Intelligence Complements Classic Testing Methods, **to be published** in 2024. **(Chapter 2)**

In addition to the above papers, four additional conference papers cited below were published during the preparation of this study. Studies 1 and 3, in particular, proved to be instrumental in guiding the choice of methods used in this thesis.

1. Uzunbayir S.: A Genetic Algorithm for the Winner Determination Problem in Combinatorial Auctions, *2018 3rd International Conference on Computer Science and Engineering*, **published** in September 20, 2018.
2. Uzunbayir S. and Kurtel K.: A Review of Source Code Management Tools for Continuous Software Development, *2018 3rd International Conference on Computer Science and Engineering*, **published** in September 20, 2018.
3. Uzunbayir S.: Reverse Ant Colony Optimization for the Winner Determination Problem in Combinatorial Auctions, *2022 7th International Conference on Computer Science and Engineering*, **published** in October 28, 2022.
4. Uzunbayir S.: Relational Database and NoSQL Inspections using MongoDB and Neo4j on a Big Data Application, *2022 7th International Conference on Computer Science and Engineering*, **published** in October 28, 2022.

## CHAPTER 2: PRELIMINARIES AND RELATED WORK

This chapter aims to answer **RQ1** outlined in Section 1.4: “*What are the existing studies, prevailing trends, challenges, and advancements in the field of mutation testing as observed in recent academic and industry practices?*”

In this chapter, we discuss the fundamental principles of mutation testing, preliminary concepts, and methodologies to establish the necessary context for understanding the area of mutation testing. Then, pivotal research and a detailed literature review that have significantly influenced the evolution of mutation testing are presented. This exploration includes innovative methodologies and emerging trends, assessing their impact on both academic research and practical applications in the industry.

### 2.1. *Fundamental Principles of Mutation Testing*

The huge number of potential errors in a software program makes it impractical to create mutants that mimic every possible fault. Mutation testing, therefore, focuses on a subset of faults that closely resemble the error-free version of the system, adequately representing the whole spectrum of possible errors. This approach is grounded in two fundamental hypotheses: *the Competent Programmer Hypothesis* and *the Coupling Effect*.

The Competent Programmer Hypothesis operates on the premise that programmers are skilled and thus produce software that nearly aligns with the correct version (DeMillo et al., 1978). This suggests that any errors made by such programmers are likely minor and only require a few syntactic modifications for correction. Mutation testing, under this hypothesis, seeks to replicate these minor errors through simple syntactic alterations, effectively simulating the types of mistakes a competent programmer might make.

The Coupling Effect shifts the focus from programmer behavior to the nature of the faults themselves (DeMillo et al., 1978). It assumes that test data capable of identifying programs with minor errors are inherently sensitive enough to also detect more complex errors. The concept, received further clarification from Offutt (1989).

According to this, a test suite capable of identifying all simple mutants in a program would also probably detect a substantial number of complex mutants. Consequently, mutation testing is generally confined to simple mutants, as noted by Jia and Harman (2011). The validity of the Coupling Effect and the Coupling Effect has been the subject of various research studies seeking to empirically support these hypotheses (Offutt, 1992; Kapoor, 2006).

## 2.2. *Mutation Analysis*

Formally, given an original program  $P$ , mutation testing uses mutation operators to generate a set of mutants  $M$  for  $P$ . Mutation operators are used to apply syntactical transformation rules to generate mutants. These operators usually correspond to regular programmer errors. Each mutant  $m \in M$  is the same as the original program  $P$ , except that a mutated program statement is changed by a mutation operator. Then, all mutants in the set of  $M$  are executed using the test suite  $T$  of  $P$ . At the end of this procedure, the effectiveness of  $T$  is evaluated using a mutation adequacy score, comparing the execution results between the mutants and the original program. A mutant  $m$  is killed by the test case  $t \in T$  in  $P$  if its results are different, otherwise  $m$  survives (Jia and Harman, 2011; Chen and Zhang, 2018).

Although mutation testing is a reliable method for identifying suitable test cases that can detect actual faults, due to its vast number, it is not feasible to generate mutations for all potential faults in a program. Consequently, mutation testing typically focuses on a subset of faults that are closest to the correct version of the program, under the assumption that this subset can provide an adequate representation of all faults, based on the principles of the Competent Programmer and the Coupling Effect.

### 2.2.1. *Mutation Process*

The traditional procedure for mutation analysis, in which mutants are created, executed, and evaluated using a test suite containing test cases, is shown in Figure 3.

3. The entire process involves six steps and can be described as follows:

**Step 1:** The original program  $P$  is altered by using mutation operators to create

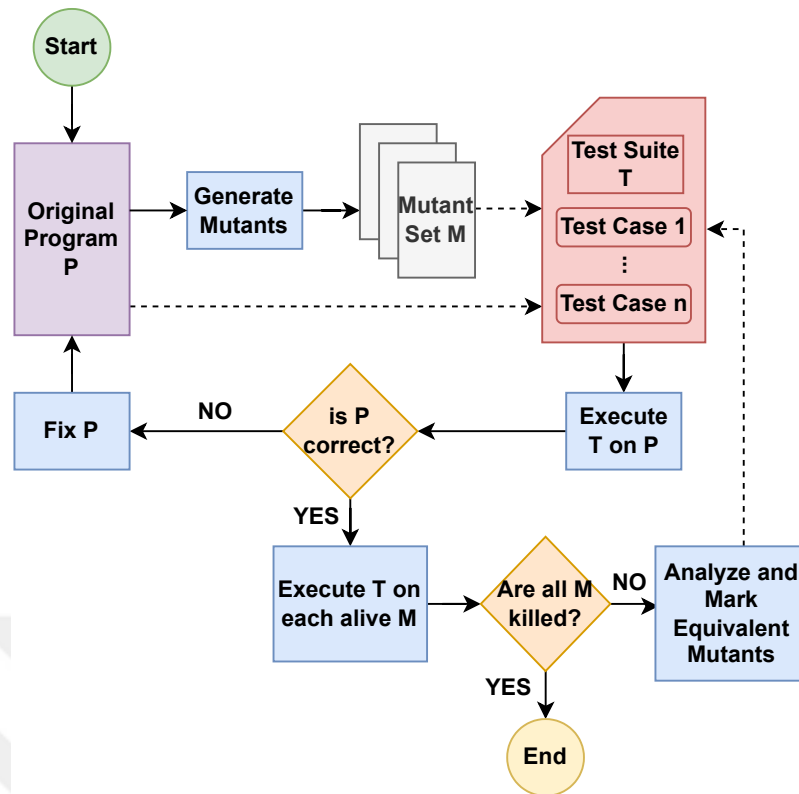


Figure 3. Traditional mutation process.

various mutants  $M$ . A mutant is a changed and faulty version of the original program. Table 1 shows an example of a simple mutation operation. Generated mutants can contain one or more faults, such as changed operators, changes in operand position, or deletion of statements. This step can be completely automated by using tools that apply specific mutation operators to  $P$ .

Table 1. An original program and its mutant.

Original Program P	Mutant M
<pre> READ n1 and n2   if (n1 &gt;50 &amp;&amp; n2 &gt;70)     PRINT n1   end if </pre>	<pre> READ n1 and n2   if(n1 &gt;50    n2 &gt;70)     PRINT n1   end if </pre>

**Step 2:**  $P$  and  $M$  are executed against the test suite  $T$ .

**Step 3:** In order to verify whether or not the output is correct,  $P$  is executed using  $T$ . If incorrect, then  $P$  must be fixed before continuing.

**Step 4:** When the output of the original program is correct,  $T$  is executed for each living mutant. Then, their output is compared with the output of  $P$  to identify which mutants should be killed. The mutant survives if  $P$  and the mutant give the same results; otherwise, it is killed and eliminated.

**Step 5:** Then, the mutation adequacy score is calculated using equation (1), which represents the proportion of mutants killed to the total number of mutants that can be killed. The procedure ends when all mutants have been killed.

**Step 6:** If any mutants survive, they must be inspected manually to decide if they are equivalent or if the test cases are not adequate to kill them. Equivalent mutants always produce the same results as the original program and can never be killed; i.e. they are syntactically different but functionally equivalent to  $P$ . If equivalent mutants are detected, they should be eliminated. If no mutants are equivalent but still alive, new test cases should be added to  $T$ , and the process continues with Step 4.

Mutation analysis measures the quality of the test suite with respect to the mutation adequacy score. In the end, the goal of the tester would be to increase the mutation score calculated by using equation (1) close to 1, so that  $T$  is sufficient to detect all the faults indicated by the mutants. Thus, this procedure provides a structured and effective way to measure test adequacy (Ma and Offutt, 2005).

$$\text{Mutation adequacy score} = \left( \frac{\text{Killed mutants}}{\text{All mutants} - \text{Equivalent mutants}} \right) \quad (1)$$

Let us illustrate with an example and provide a detailed explanation of the procedure. Consider the following code piece:

**Original program:**

```
1   Read a number
2   if (number > 50)
3       Print "RED"
4   else
5       Print "BLUE"
6   end if
```



To test this program, we supply values to the “number” variable, specifically 0, 49, 50, 51, and 90 by using equivalent partitioning and boundary value analysis. The objective is to determine whether these values are sufficient to identify all the defects.

Next, seven mutants are generated from the original program provided above, using various mutation operators:

**Mutant 1: Change operator > to <**

```
1  Read a number
2  if (number < 50)
3      Print "RED"
4  else
5      Print "BLUE"
6  end if
```

**Mutant 2: Change operator > to ==**

```
1  Read a number
2  if (number == 50)
3      Print "RED"
4  else
5      Print "BLUE"
6  end if
```

**Mutant 3: Change operator > to <=**


```
1  Read a number
2  if (number <= 50)
3      Print "RED"
4  else
5      Print "BLUE"
6  end if
```

**Mutant 4: Change operator > to >=**

```
1 Read a number
2 if (number >= 50)
3     Print "RED"
4 else
5     Print "BLUE"
6 end if
```

**Mutant 5: Deletion of Print "RED" statement**

```
1 Read a number
2 if (number > 50)
3 else
4     Print "BLUE"
5 end if
```



**Mutant 6: Deletion of Print "BLUE" statement**

```
1 Read a number
2 if (number > 50)
3     Print "RED"
4 end if
```

**Mutant 7: Wrong syntax number !% 50**

```
1 Read a number
2 if (number ! % 50)
3 else
4     Print "BLUE"
5 end if
```

Table 2 presents the results of the mutation testing for the example mentioned above. Mutants that produce the same results as the expected output are colored green, indicating their survival. In contrast, those that produce different results are marked in red, signifying their elimination. Specifically, the input value 0 leads to the failure of mutants 1 and 3, while having no impact on mutants 2, 4, and 5. The objective of mutation testing is to eradicate all mutants, thereby enhancing the quality of the test suite. As depicted in Table 2, mutants 1, 2, 3, and 4 are successfully eliminated. However, mutants 5 and 6 persist against some inputs and remain unaffected by others. Furthermore, mutant 7 triggers a compilation error. This implies that mutants 5, 6, and 7 are equivalent or redundant for testing this program and, ideally, should not be generated in the initial phase of the process.

Table 2. Mutation testing results of the example.

Input	Expected Output	Mutant 1	Mutant 2	Mutant 3	Mutant 4	Mutant 5	Mutant 6	Mutant 7
0	BLUE	RED	BLUE	RED	BLUE	BLUE	nothing	syntax error
49	BLUE	RED	BLUE	RED	BLUE	BLUE	nothing	syntax error
50	BLUE	BLUE	RED	RED	RED	BLUE	nothing	syntax error
51	RED	BLUE	BLUE	BLUE	RED	nothing	RED	syntax error
90	RED	BLUE	BLUE	BLUE	RED	nothing	RED	syntax error

### 2.2.2. Cost of Mutations

Mutation testing is an extremely costly process that requires substantial computational resources and human effort. It is time-consuming as a result of the need to compile and test each mutant separately. The cost of mutations can be explained with respect to the mutation analysis process expressed in Section 2.2.1:

- **In Step 1**, where numerous mutants are created using mutation operators, the number of mutations is proportional to the product of the number of data references and the data objects for a software component (Offutt et al., 1996).
- **In Step 2**, where the original program and the mutants are executed against the test suite, the cost can be neglected, since no computation is needed.

- **In Step 3**, the verification of the correctness of the original program output is checked, and therefore manual intervention is required, increasing the required effort. Although this task is not unique to mutation testing, it is still considered within the scope of cost.
- **In Step 4**, where the execution of the original program and all mutants with the test cases in the test suite causes significant computational costs.
- **In Step 5**, the adequacy of the mutation is calculated and the cost depends on the calculation operation.
- **In Step 6**, the equivalent mutants and the generation of additional test data are determined to kill the remaining mutants, and these challenging tasks require a complete inspection of the program mechanics.

Many solutions have been proposed to reduce computational resources of mutation testing and human effort; some focus on reducing mutations or executions for computational cost, and others, on reducing manual costs for human effort. However, none of these proved to be 100% effective. The next section introduces these techniques.

### **2.3. Cost Reduction Techniques Literature Review**

Mutation analysis requires a significant amount of time and resources for the following reasons: The process of mutation generation may be difficult considering larger programs, since each mutation should denote a potential error; a test suite should contain robust test cases to find and manage mutations; therefore, preparing a good test suite is time-consuming; designing mutations, preparing test cases, and evaluating results may be a demanding process, especially working with complex programs. To reduce these costs, various categories of cost reduction techniques have been proposed over the years. Offutt and Untch (2001) used the categories of *do fewer*, *do faster*, *do smarter*, while Mateo et al. (2010)' categories were *mutation generation*, and *test case generation and execution*. Due to the distinct nature of costs and resources they aim to optimize in the testing process, in this study, as depicted in Figure 4, the cost reduction

techniques are separated into two: **computational cost reduction**, and **manual cost reduction** techniques.

### *2.3.1. Computational Cost Reduction Techniques*

Computational cost reduction techniques can be classified into three; **mutant reduction**, **execution cost reduction**, and **run-time optimization**. The main source of computational costs comes from performing mutation analysis with a large number of mutants in the test suite. Some of the mutants may be unnecessary, if equivalent, or meaningless, and therefore should not be generated in the first place. Therefore, reducing the number of mutants while preserving the effectiveness of the testing process has become an important research topic. In other words, mutant reduction techniques can be defined as the problem of finding a subset of mutants for which the mutation adequacy score of testing with all mutants is equal to testing with the subset of mutants. Here, we discuss four mutant reduction techniques: **mutant sampling**, **selective mutation**, **higher-order mutation**, and **mutant clustering**.

**Mutant sampling** is a technique that involves the selection of a subset of mutants generated for a specific software program. The objective of mutant sampling is to reduce the number of mutants that need to be evaluated by selecting a smaller but representative subset. The purpose of this selection is to obtain a sample that accurately represents the full set of mutants, allowing for the extrapolation of the results of mutation testing to the entire set. There exist various methods for selecting a representative sample of mutants, including variation random sampling. The choice of method depends on the particular requirements of the mutation testing and the characteristics of the tested program. The ultimate goal of mutant sampling is to balance the accuracy of the results and the effort required to carry out the testing process.

Wong (1993) experimented with various sampling proportions between 10% and 40% in increments of 5%. As a result, sampling with 10% of the mutants is 16% less effective than processing with the entire set. This study concluded that mutant sampling is valid with a x% value higher than 10%, and was empirically studied by DeMillo et al. (1988), and King and Offutt (1991). Papadakis and Malevris

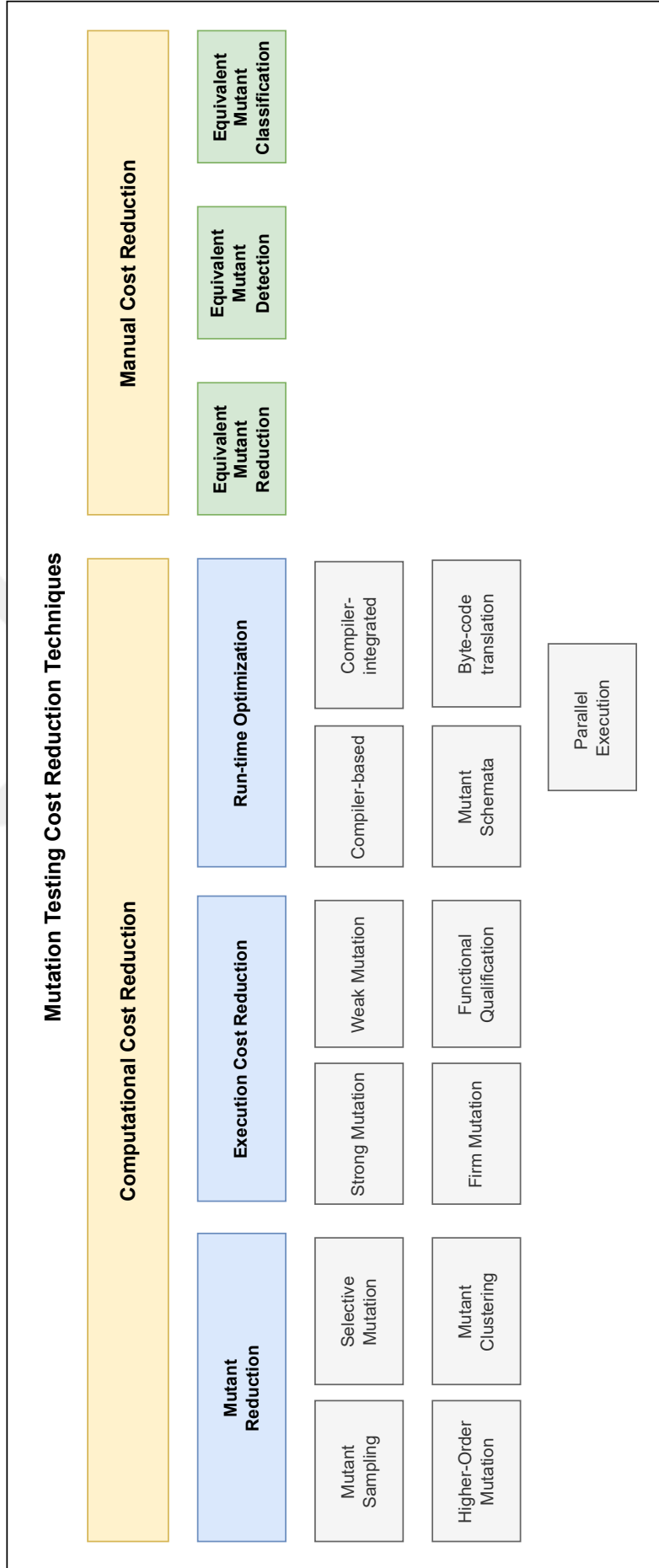


Figure 4. Mutation testing cost reduction techniques.

(2010) empirically investigated the effectiveness of the test of different random mutant sampling methods, ranging from 10% to 60% in steps of 10%. They concluded that the loss of effectiveness of recorded tests ranges between 26% and 6%. The authors also applied second-order mutation and demonstrated that second-order strategies with mutant sampling are effective in significantly reducing the number of produced and equivalent mutants. This led to additional savings in the number of required test cases while still maintaining a low level of fault detection loss. However, to validate the claims statistically, this study requires more experiments with second-order mutation to validate the claims statistically. Derezińska and Rudnik (2017) used random sampling and compared various sampling criteria such as fully random, class random, file random, method random, mutation operator random, and namespace random techniques in an object-oriented program, and suggested that the degree of sampling is about 40% for object-oriented operators, compared to 30-35% for standard ones. They concluded that sampling is helpful in reducing the number of mutants; however, it also reduces the mutation adequacy score. Therefore, the article suggests that in practical situations, the number of mutants is necessarily the most significant factor in terms of cost. The overall time required for mutation testing is also heavily influenced by the number of tests needed. Furthermore, Pitts (2021) argued that random mutant selection is almost as effective as selecting only a smaller set of mutants when a posteriori similar information is available and to highlight that observations are only valid in an environment that adequately allows mutations. As a result, the author concludes that random selection is still an efficient technique. However, a wider range of test subjects and programs can be used to experiment with random sampling in different environments.

**Selective mutation** is an approximation technique to reduce the number of mutants by identifying a small set of mutation operators whose mutants can simulate those generated by all available operators. In addition, this technique focuses on a subset of highly effective operators that result in significant savings. This means that selective mutation creates fewer mutations and that test suites that perform well for these mutations will also perform well for all mutations created by all mutation operators. In simpler terms, selective mutation focuses on a subset of mutations that yield the most

impact and cost-effectiveness. It is crucial to exercise caution when implementing selective mutation, as it has the potential to decrease overall trust in the results of mutation testing. If the portion of code chosen for mutation does not accurately represent the complete system, the results obtained may not give a fair assessment of the quality of the test suite.

Offutt et al. (1996) experimented with this idea in the Mothra software testing environment developed by DeMillo et al. (1988), and found that only five of the twenty-two mutation operators were adequate to perform an effective mutation analysis. This contradicts the results of previous mutation testing studies, as previous approaches focused on increasing the number of mutation operators to intensify mutations. However, in this study, fewer operators were found to increase efficiency and opens up more research opportunities in the area of mutation testing. Siami Namin et al. (2008) used statistical methods to find a subset of the comprehensive Proteum mutation operators that generates a significantly smaller number of mutants compared to the entire set. Despite this reduction, the subset still allows for an accurate prediction of the effectiveness of the test suite on the mutants produced by the full set of operators. The cross-validation results showed that the procedure used to identify the subset was appropriate and that the findings can be applied to other programs. With this model, researchers can more efficiently estimate the effectiveness of a test suite in a program without having to generate all possible mutants. Zhang et al. (2013) explored the application of two strategies, operator-based and random mutant selection. The result indicated that the combination of operator-based and random mutant selection results in more precise and comprehensive fault detection compared to the use of each strategy independently. The research evaluated various sampling strategies on 11 real-world Java programs and concluded that a sample of only 5% provided highly precise mutation scores. Gopinath et al. (2016, 2017) showed that selective mutation is actively harmful to mutation diversity and performs worse than random sampling.

**Higher-order mutation** aims to find rare but valuable mutants that can denote faults. In this mutation testing method, mutants are generated by applying mutations to already mutated programs, rather than directly to the original program. This results in the creation of a new set of more complex and realistic mutants that better reflect



real-world faults in the software. This approach is considered a more advanced form of mutation testing compared to traditional first-order mutation, which only modifies the original program by mutation one operator only (see Table 3).

Table 3. An example of higher-order mutation.

Original Program	First-Order Mutant	Higher-Order Mutant
<pre> READ x and y   if (x == 10    y &lt; 20)     PRINT "DONE"   end if </pre>	<pre> READ x and y   if (x == 10 &amp;&amp; y &lt; 20)     PRINT "DONE"   end if </pre>	<pre> READ x and y   if (x != 10 &amp;&amp; y &lt; 20)     PRINT "DONE"   end if </pre>

The advantages of higher-order mutation testing are: improving the accuracy of mutation score by generating more complex and realistic mutants, reducing the number of mutants, enhancing the reliability of mutation testing, improving the coverage of testing to uncover faults, and identifying complex faults to make testing applicable for interactions between different mutants and interactions between mutants and the test suite.

Polo et al. (2009) presented an approach to reduce the cost of mutation testing by combining first-order mutants using various combination algorithms. Depending on the algorithm used, the number of mutants can be halved, resulting in a substantial decrease in the time and effort required for mutant execution and analysis. Furthermore, the article shows that the second generation of mutants has significantly fewer equivalent mutants, as demonstrated by a decrease from 18.66% to approximately 5%. This study is important in showing that higher-order mutation can be used to reduce equivalent mutants. However, it needs to be validated with additional research using industrial applications, and Van Nho et al. (2019) showed that the combination of mutants using this technique brings more effective results. Harman et al. (2011) used higher-order mutation for test data generation. Their study presented SHOM, a new technique for generating test data through the use of mutations. It combines dynamic symbolic execution with search-based software testing methods to achieve high mutation adequacy and the ability to detect both simple and complex mutations. They conducted an empirical study using 17 programs, including industrial

company programs. The results showed that SHOM performed better than previous mutation-based test data generation techniques, successfully identifying between 8% and 38% more mutations than the previous best approach. Abuljadayel and Wedyan (2018) proposed a method to create higher-order mutants with a genetic algorithm and developed a tool using Java. To assess the effectiveness of the proposed idea, they performed experiments with mutants generated using HOMAJ on a 315 LOC program and created test cases using Randoop. Their findings showed that their algorithm uses a replacement crossover to produce new parent mutants, which are harder to kill than the previous population. These solutions show potential, but their study needs to be extended using larger programs and other programming languages to compare their effectiveness. Lima and Vergilio (2018) reported a systematic mapping study on search-based higher-order mutation testing. Their study looked at various aspects, such as the search-based algorithms used, the fitness of the evaluation applied, and the evaluation methods. They also analyzed the main publication venues and tracked the evolution of the field over time, including trends and opportunities for future research. The study found 17 different algorithms used in the context of higher-order mutation, with the genetic algorithm and NSGA-II being the most widely used. This points to the potential for exploring other objectives with multi-objective algorithms, which seems to be a growing trend in the field. Recently, Li et al. (2022) used deep learning mutation testing to assess whether a higher-order mutation is applicable in this context. They proposed a mutation testing framework that evaluates mutant classification tuples using first-order and higher-order mutants with four common datasets. The study reported that higher-order mutation effectively reduces the set of mutants using deep learning methods in mutation testing. Furthermore, extensive higher-order mutation tests have shown it to be more efficient than first-order mutation in the following studies; Papadakis and Malevris (2010), Kintis et al. (2010), Madeyski et al. (2014), Omar and Ghosh (2012), Mateo et al. (2010), and Parsai et al. (2016). More on higher-order mutation is discussed and experimented in Chapter 5 of this dissertation.

**Mutant clustering**, proposed by Hussain (2008), is a subset of mutants chosen using clustering algorithms rather than random selection, as in mutant sampling. The goal is to minimize the number of mutants that need to be evaluated and to present the

results in a more comprehensible manner. This is achieved by grouping mutants based on structural or behavioral similarities. First, all first-order mutants are generated. Then, a clustering algorithm is applied to distribute first-order mutants into various clusters that can be killable by the test suite. Each mutant in the same cluster is guaranteed to be killed by similar test cases. In the end, a small number of mutants will be selected from each cluster. As for clustering algorithms, it is possible to apply K-means, agglomerative hierarchical clustering, or mean-shift clustering after mutant generation is complete. Mutant clustering is a valuable method for improving the productivity and precision of mutation testing. By grouping similar mutants, it becomes easier to identify patterns and understand the strengths and weaknesses of the test suite, providing valuable information on the quality of the tests.

Ma and Kim (2016) applied mutant clustering to reduce the number of mutants executed. They showed that their approach is capable of clustering mutants by comparing the values of an innermost expression, which, as a result, is more effective than mutant sampling. The study introduced the idea of c-overlapped mutants, in which mutants are weakly killed in a test case, and involves testing only one mutant from each group of overlapping mutants with strong mutation. The research successfully reduces the number of mutants generated; however, the mutants were only grouped based on their expression, and this had the drawback of requiring at least two mutants to be generated by the mutation operator. The scope of clustering should be expanded to include mutants for a statement, which would show greater reduction in the cost by clustering a larger number of mutants. Yu and Ma (2019) worked on the clustering of c-overlapped mutants and focused on the clustering at three levels; the expression, statement, and block levels. The results of the study show that clustering mutants at the statement level led to a 24.44% reduction in mutant executions compared to the weakly live mutant filtering approach. The authors claimed that this reduction is higher than that achieved by the expression-level clustering method, which resulted in a reduction of only 10.51%. The block-level clustering resulted in a smaller reduction of 1.06% compared to the statement-level clustering. However, this small improvement is not practical, due to the increased cost of saving states required for a wider comparison scope. Liu and Song (2021) used second-order mutants with a self-organizing map

neural network. The first step involved using a more integrated combination strategy to generate second-order mutants with muJava. The next step was to analyze the reasons for killing mutants and used the similarity of intermediate values in the execution of the second-order mutants to construct an appropriate self-organizing neural network model. The final step was to cluster the second-order mutants based on this model to achieve a reduction in their number. The results of the experiments showed that this method was effective in reducing the number of second-order mutants that produced similar adequacy scores compared to the first-order mutation version.

Execution cost reduction techniques aim to reduce costs by optimizing the mutant execution process. In this context, there are four types of execution cost reduction techniques; **strong mutation**, **weak mutation**, **firm mutation**, and **functional qualification**.

**Strong mutation** is a type of mutation testing in which each mutant generated by the mutation operator is executed and its behavior is compared with that of the original program. The goal of a strong mutation is to determine whether the program's test suite is able to distinguish between the original program and each mutant, thus indicating that the test suite is effective in finding faults in the program. It is usually called traditional mutation testing or general mutation testing. Simply stated, a mutant generated from a given original program using mutant operators can be killed if and only if the mutant produces results different from the original program.

**Weak mutation** aims to avoid complete execution of the original program and its mutants. To do this, it compares the results of the original program and the mutant immediately after the execution of the mutant or mutated component, rather than waiting for the final step, as in a strong mutation process. It is in some ways more effective, allowing for checking of each mutant condition immediately after execution, so that there may not be a need to generate every single mutant. However, in other ways, this approach may be less effective than strong mutation in that it trades test effectiveness for effort costs.

Kintis et al. (2010) worked on weak mutation and revealed that it not only significantly decreases the number of equivalent mutants produced, but also provides a stronger test criterion than previously believed. Moreover, they offered an estimate

of how many first-order mutants need to be removed to eradicate the entire original set. The findings of their research revealed that targeting just a fraction of the mutant set for elimination is sufficient, as this leads to the removal of the remainder. Their experimental set could be extended to include more methods such as higher-order or selective mutation for further advances. Souza and Gheyi (2020) used weak mutation in method-level mutation operators and experimented with first-order and higher-order mutants. They claimed that higher-order mutants are not very efficient in method-level mutation. The study encoded 223 first-order mutants and 438 higher-order mutants, finding that, on average, 91% of all mutants could be discarded. This study reveals the usefulness of weak mutation and provides interesting details for further investigation by applying it to other mutation strategies or programming languages. Yao et al. (2020) suggested a method to generate test data for weak mutation testing based on the dominance levels of mutant branches. They converted all mutants in the original program into mutant branches to create a new program that incorporates all these branches. They evaluated the dominance relationship of the mutant branches in the transformed program to determine the non-dominant mutant branches and their dominance levels. The proposed method was applied to 15 programs and the experimental results showed that compared to other methods, it improves the quality of the test data and reduces the cost of mutation testing. The study required a separate execution of each mutant, increasing costs, but this approach increased the fault detection ability of the test suite.

**Firm mutation** aims to create an alternative to weak and strong mutations to overcome the disadvantages of both terms. In a firm mutation, there is no expectation of an assertion guaranteeing that the difference in behavior would be caught. The change needs to propagate further from its place of origin until the lexical boundaries of the source.

Jackson and Woodward (2001) proposed a parallel firm mutation technique for Java programs. At that time, they claimed that no clear systematic approach existed for choosing the parts of a program's code to use for firm mutation testing, since there were no object-oriented approaches. However, since then, the emergence of object-oriented languages has provided a potential solution. The study explored the application of firm

mutation to Java methods by leveraging Java threads to execute mutants and paved the way for the usage of object orientation along with parallelism used in multithreading for mutation testing. Singh and Srivastava (2017) proposed an extension to firm mutation in a small-scale application with aspect-oriented programming. Their aim was to assess the costs of mutation operators' application. The authors claimed that the system evaluated in this study reduces the number of live mutants to deal with. Despite this being an approximation, they argued that well-designed test suites can uncover most of the faults introduced by the mutation operator. However, a small but still significant increase in the size of the test set was required to produce reliable results in terms of the mutant analysis criteria.

**Functional qualification** applies a number of algorithms to mutation analysis with the aim of reducing execution costs. This approach can function in any test environment, thus the test bench used can be implemented with any language, as long as the results return pass or failure, which are considered for each test. Therefore, whenever a checker or assertion is missing from the functional verification environment, mutations will remain alive. Coverage metrics do not check the output behavior, which is captured instead by functional qualification, which is able to measure the bug detection.

Lin et al. (2012) presented a novel error propagation analysis method that effectively addresses the issue of error propagation in mutation analysis. The authors used a probabilistic analysis of HDL designs to add the appropriate observation points. Using the firm mutation approach, the mutant status was reported as determined by monitoring these selected observation points. As a result, the firm mutation approach could more effectively identify surviving mutants and eliminate redundant test cases, thus reducing the simulation cost for subsequent strong mutations. Empirical results demonstrated that the proposed method is more efficient than weak mutation, which neglects the problem of error propagation.

Run-time optimization use various sources, such as compilers, files, or parallel machines, in order to reduce costs. There are five approaches in this category; **compiler-based, compiler-integrated, mutant schemata, byte-code translation, and parallel execution.**

**Compiler-based** run-time optimization was introduced Delamaro (1993). It is a generic method in which mutants are compiled to form an executable program. After that, each compiled mutant is tested with a number of test cases, as shown in Figure 5. This is a fast technique; however, the speed might sometimes be limited when the run-time of a program takes longer than the compilation time. This problem is called compilation bottleneck in the literature (Byoungju and Mathur, 1993).

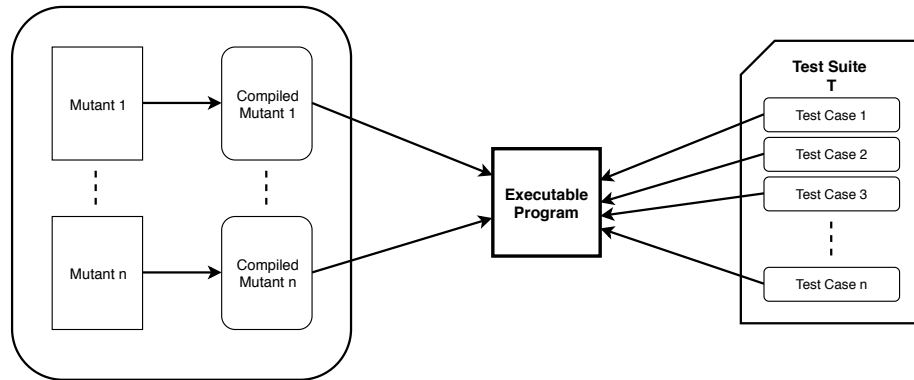


Figure 5. Compiler-based run-time optimization.

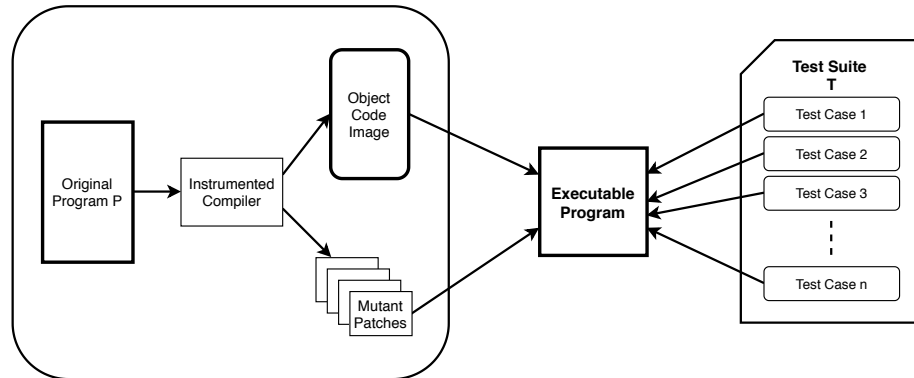


Figure 6. Compiler-integrated run-time optimization.

**Compiler-integrated** run-time optimization was proposed by DeMillo et al. (1991). It is aimed at reducing performance problems of compiler-based methods. Since the original program and the mutant differ with a single syntactic change, compilation of each mutant separately might cause redundant compilation costs. To remove this drawback, the compiler-integrated method contains an instrumented compiler that produces two outputs from the original program; an executable object code for the original program and a set of patches for mutants as shown in Figure 6.

These patches consist of information to help convert the object code to the executable code for the related mutant (Krauser, 1991).

**Mutant schema** run-time optimization was proposed by Untch et al. (1993). In this approach, compilation of the mutants is not separated. Instead, a meta-mutant, which represents all possible mutants, was generated (see Figure 7). This meta-mutant will be tested with the test suite and it will be compiled only once. Therefore, compilation costs will be reduced. More detailed explanations and experiments can be found in the works of Schuler and Zeller (2009) and Wright et al. (2013).

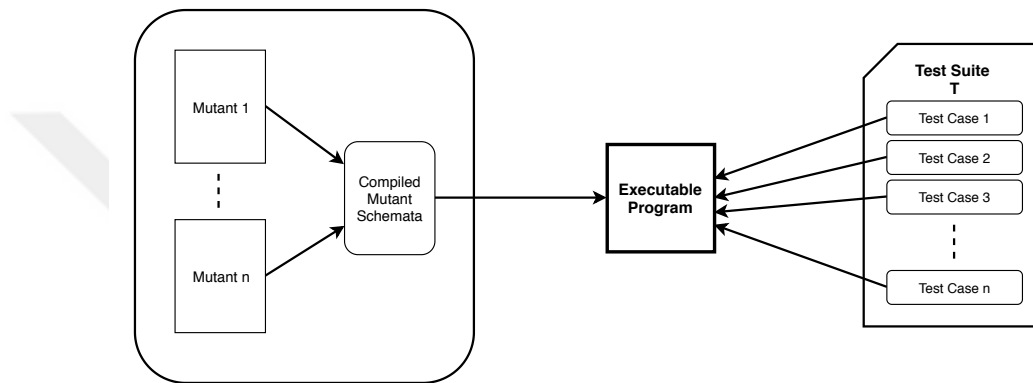


Figure 7. Mutant schemata run-time optimization.

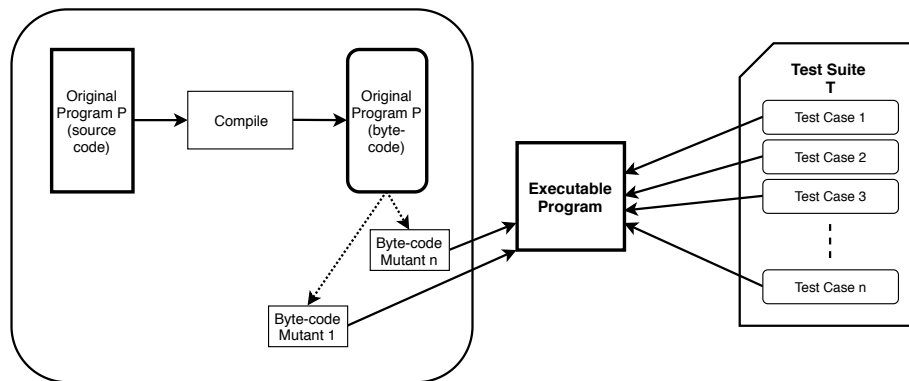


Figure 8. Byte-code translation run-time optimization.

**Byte-code translation** run-time optimization was proposed by Ma et al. (2005). Instead of generating mutants from the source code of the original program, this method generates them from the compiled object code (see Figure 8). Therefore, these byte-code mutants do not have to be executed by compiling them. This technique allows working on off-the-shelf programs when their source code is not available. The



drawback of the method is that some mutation operators cannot be represented with byte-codes (Schuler et al., 2009; Vallée-Rai et al., 2010).

**Parallel execution** idea was proposed by Mathur and Krauser (1988). In this study, the authors executed several mutants on a vector processor. Parallel execution aims to improve the efficiency of mutation testing by executing the original program and its mutants in parallel processors. This approach reduces the total time for mutation analysis. Offutt et al. (1992) developed HyperMothra tool that is capable of performing mutations using the Intel iPSC/2 hypercube machine with 16 processors. Mateo and Usaola (2013) adapted this idea to current technologies by using five algorithms under different network configurations and different number of processors.

### 2.3.2. *Manual Cost Reduction*

Manual cost reduction techniques are based on **equivalent mutant problem**. Identifying whether a mutant is equivalent or not is undecidable (Budd and Angluin, 1982), therefore, it is not possible to develop fully automated solutions and human intervention is required.

An equivalent mutant is generated when a mutation does not change the output of the original program, i.e. it is syntactically different but semantically identical. An example of an equivalent mutant, which is created by substituting the operator of the original program + with the operator \*, is illustrated in Table 4. Since the value of a does not change within the scope of the if statement, the original program *P* and the mutant *M* will produce identical results.

Table 4. An example of equivalence mutation.

<b>Original Program P</b>	<b>Mutant M</b>
<pre>int a = 2, b = 2, c = 3   if (b == 2)     PRINT b AND c     b = a + b   end if</pre>	<pre>int a = 2, b = 2, c = 3   if (b == 2)     PRINT b AND c     b = a * b   end if</pre>

Many different techniques have been proposed to at least partially automate the reduction of manual costs for equivalence mutations. The solutions to the equivalent mutant problem can be divided into three categories: **equivalent mutant reduction**, **equivalent mutant detection**, and **equivalent mutant classification**.

**Equivalent mutant reduction** is based on an attempt to reduce the number of equivalent mutants. The main approach is to use higher-order mutation methods, such as second-order mutation (Jia and Harman, 2009). Harman et al. (2010) claim that higher-order mutants can simulate real faults better than first-order mutants. Other approaches suggest co-evolutionary search techniques to prevent equivalent mutants (Adamopoulos et al., 2004).

Polo et al. (2009) created a set of second-order mutants through the combination of first-order ones. The second-order generation of mutants showed a significant reduction in the number of equivalent mutants, with the percentage decreasing from 18.66% to around 5% in one of their experiments. As a part of the testooj tool they implemented, three combination strategies were employed, namely LastToFirst, DifferentOperators, and RandomMix. All three methods notably decreased the number of equivalent mutants, with LastToFirst and RandomMix cutting the number of mutants by half, while DifferentOperators reduces the number by a different amount. The experiments were promising; however, they needed to validate the results using industrial projects. Papadakis and Malevris (2010) empirically evaluated several first- and second-order mutation testing strategies. The results suggested that, in general, first-order strategies were more successful in identifying faults compared to second-order strategies, although they came at a higher cost. On the other hand, second-order strategies significantly reduced the number of equivalent mutants and resulted in considerable savings in the number of mutants generated and required test cases. Statistical analysis and more experiments would direct the research to better results. Madeyski et al. (2014) proved that second-order mutants can significantly improve the efficiency of mutation testing, but at a cost in testing strength. They implemented four distinct second-order mutation strategies, in conjunction with the first-order mutation, and conducted a comparative analysis across multiple dimensions. The experimental investigation revealed that second-order mutation, specifically the

JudyDiffOp strategy, yielded the most favorable outcomes in terms of the total number of mutants generated, the correlation between the employed mutation strategy and equivalent mutant generation, the count of uneliminated mutants, the duration of mutation testing, and the time required for manual classification. The results were good, but not perfect. Higher-order mutants than second-order ones needed to be implemented and compared with these results. Kintis et al. (2010) stated that the use of second-order mutation can be advantageous in two ways: first, it can decrease the number of equivalent mutants generated, and second, it can allow high collateral coverage for strong mutation. With these advantages in mind, a new set of second-order mutation testing strategies based on the dominator was developed. Although the investigation of the study of the collateral behavior of second-order strategies was promising, more research was needed to determine which criterion should be prioritized to achieve collateral coverage and the advantages of this approach. A more recent research by Garg et al. (2023) proposed a technique called Cerebro that was able to learn to identify a subset of mutants that can kill all other mutants in a given set. They conducted experiments with 58 programs and found that Cerebro was able to identify these mutants with 0.85 precision and 0.33 recall in a scenario that is trained on different projects than the ones it was evaluated on. This information was useful and can be used by testers to design test cases that can kill more than twice the number of mutants that are harder to kill than they would be able to using randomly selected mutants or other machine learning-based mutant selection techniques. According to the experiments, Cerebro analyzed 66% fewer equivalent mutants and executed 90% fewer mutants, which significantly reduced the practical effort and cost of the approach.

**Equivalent mutant detection** approaches attempt to correctly identify a percentage of all equivalent mutants of the original program. These techniques reduce mutation costs, but not their effectiveness (Kintis, 2016). They function by eliminating equivalent mutants by developing useful heuristics, and the most effective heuristic methods depend on compiler optimizations (Mateo et al., 2013). The aim is to transform mutants to their optimized versions in such a way that semantically equivalent mutants will resemble the optimized version. The effectiveness of this

method has been empirically studied by Papadakis et al. (2015), and the results indicate that the approach can identify up to 30% of all equivalent mutants. Papadakis et al. (2015) examined the feasibility of utilizing the compiler optimization techniques of the GCC compiler to detect equivalent mutants. They stated that their approach was easily scalable and widely applicable, relying on the same technology as most compilers. Their method indicated that any mutant that produces code compiled identical to that of the original program is deemed equivalent. As a result, they were able to demonstrate that the method can effectively identify a significant number of equivalent mutants in the real world. Furthermore, they used an empirical study of 18 benchmark programs and discovered that the proposed method was capable of detecting 30% equivalent mutants. Offutt and Pan (1997) introduced a set of heuristic-based strategies with the aim of determining the infeasibility of constraint systems that model the conditions necessary for the killing of a mutant during mutation testing. They proposed a partial solution to the problem of equivalent mutant detection and the feasible path problem, and demonstrates the specific relationship between the two. Their findings demonstrated that the approach was an effective partial solution to the aforementioned issues. Furthermore, they showed that this technique was applicable to the feasible path problem and may produce better results than equivalent mutants. Future work for this general technique could be generalized to all instances of the feasible test problem and thus could support branch coverage techniques and data flow testing. Nica and Wotawa (2012) proposed an approach that involved the integration of constraint representations of programs with mutation testing to identify a unique test case that distinguishes a program from its mutant. However, due to the underlying problem being undecidable, their approach could not ensure a solution. They claimed that the effects of various parameters, such as nesting depth, are subject to further study. Additionally, their approach enabled the incorporation of new test cases into the test suite. The test cases they computed improved the mutation score. Hierons et al. (1999) suggested that amorphous slicing can be used to support manual analysis of particularly hard-to-kill mutants. The authors demonstrated that program slicing was able to make it easier to determine whether mutants are equivalent. When a mutant is not equivalent, program slicing simplified it, making it easier for the tester to find

test cases that can identify and eliminate it. When either firm or strong mutations are used, program slicing may also be used to identify certain equivalent mutants in advance, allowing the tester to generate fewer equivalent mutants. Their approach was not combinatorial explosion; however, they needed to perform more experiments with more real-world projects. Laurent et al. (2017) proposed a gamified system that can be used as a standalone tool for equivalent mutant detection. The authors presented a prototype of a gamified equivalent mutant detection platform, which allowed players to identify and eliminate mutants in software programs by writing a test or labeling them as equivalent. Players could earn points for successfully killing a mutant or agreeing on its equivalence. The study aimed to investigate the effects of different mutant and player attributes to optimize the game and achieve the best possible results for the tester. The idea sounds promising, but there is no indication that there exists an actual implementation as of today. Most recently, Jammalamadaka and Parveen (2022) suggested an interesting approach using the wavelet convolutional rain optimization technique to distinguish between equivalent and nonequivalent mutants based on code features. They added a wavelet function to the neural network to reduce the dimension of features in the convolutional layers. The simulation results demonstrated that this technique was better at identifying equivalent mutants in the source code than other existing methods with 85.17% precision. To make it more valid, the authors need to experiment with other classifiers for their approach. Moreover, Gong et al. (2022) presented an innovative method for automatic detection of equivalent mutants by tracing the program behavior. The research suggested that the detection of equivalent mutants involves the use of a weighted software behavior graph, which was not previously used. The proposed method was capable of detecting different execution paths and was sensitive to the frequency of execution. By comparing the weighted software behavior graphs of an alive mutant and its original program, it was possible to precisely examine whether the alive mutant is the same as the original program in terms of infection state or propagation. The authors evaluated the performance of their approach using an open dataset of equivalent mutants manually evaluated. The results showed that the approach was able to detect 77.5% of all equivalent mutants, which was a significant improvement over the existing methods.

**Equivalent mutant classification** aims at categorizing mutants as either possible to kill or possible to make equivalent by using the program characteristics rather than detecting the existence of equivalent mutants. These techniques suggest that if a mutant does not conform to the characteristics of the program, it is probably killable.

An empirical study conducted by Papadakis et al. (2014) revealed that equivalent mutant classification is effective only when low-quality test suites are used for the program under test. However, a study by Zhang et al. (2018) claimed that their proposed methods, namely predictive mutation testing, can improve the precision of mutant execution results with only a small overhead. Schuler and Zeller (2013) investigated the use of coverage changes to detect nonequivalent mutants. They claimed the hypothesis that if a mutant alters the coverage of a run, it is more likely to be non-equivalent. The study involved 140 manually classified mutations of seven Java programs. The findings indicated that the problem of undetected mutants was widespread, with about 45% of these mutants being equivalent. Furthermore, manual classification of mutants was a time-consuming task, with approximately 15 minutes required per mutation. The study showed that coverage is a simple, efficient, and effective method of identifying equivalent mutants. The authors should focus on finding the most powerful mutants and evaluate improvements in test suites using the proposed approach in future studies. Naeem et al. (2020) developed a method to predict equivalent mutants without the need for manual human intervention by training classification models. To achieve this, the authors utilized three different classifiers; Random Forest, Gradient Boosted Trees, and Support Vector Machines. Through experiments carried out on ten subject programs ranging from a few hundred to a few thousand lines of code, the study demonstrated that the proposed approach can achieve high levels of accuracy in predicting classification results in two application scenarios. Since machine learning is a current trend for mutation testing, the idea should be further investigated using various mutant reduction techniques, such as selective mutation. A more recent approach by Kusharki et al. (2022) presented an automated method to classify equivalent mutants in the mutation testing of Android applications using tree-based convolutional neural networks. The authors used a tool to generate a standardized dataset, transformed the data into vector representation,

and generated training, development, and testing datasets. The results indicated that automating the classification of equivalent mutants still requires more work and should be explored further using more operators and other tools.

#### ***2.4. Challenges and Current Trends for Mutation Testing***

In this section, challenges and current trends in mutation testing that involve newly emerging ideas and approaches, especially considering the latest advances in artificial intelligence, are discussed to enrich our understanding of **RQ1**.

##### ***2.4.1. Challenges and Interests of Mutation Testing***

Although mutation testing has been around for several decades, it still faces a number of challenges, and new trends and interests are emerging in the field. These challenges and interests for mutation testing can be listed as follows:

- One of the biggest challenges in mutation testing is the high computational cost. Mutation testing involves the creation of a large number of mutant programs, each of which must be executed using the test suite. This may be time-consuming and computationally expensive, particularly for large codebases. As a result, researchers and practitioners are exploring new techniques to reduce the cost of mutation testing.
- Another challenge in mutation testing is the tool support demands. While there are several mutation testing tools available, they often have limitations in terms of the languages and frameworks that they support. Additionally, some of these tools can be difficult to use, particularly for those new to mutation testing. As a result, researchers and practitioners are working on developing new mutation testing tools that are more user-friendly and support a wider range of programming languages and frameworks.
- Mutation analysis for model-based testing using formal models and generated variations from those models is another area which requires constructing a model of the software under test. This model can be a formal model, such as

a finite-state machine or a push-down automata, or an informal model, such as a high-level design document. Then, mutants are generated for the model followed by test case execution and adequacy calculation. Model-based testing can be challenging to implement, especially for complex software systems. It requires expertise in model construction and mutation generation, and it can be computationally expensive to execute a large number of mutants. However, it is a valuable approach to identify weaknesses in a test suite and guide future testing efforts.

- In recent years, there has been growing interest in the application of mutation testing to machine learning models. Machine learning models are increasingly being used in a wide range of applications, from healthcare to finance. However, these models can be difficult to test, particularly as they become more complex. Mutation testing offers a way to evaluate the quality and effectiveness of machine learning models by creating small changes to the model and evaluating its performance. This is an exciting new area of research in mutation testing and one that is likely to see significant growth in the coming years.
- There is a trend towards the integration of mutation testing into continuous integration and continuous delivery (CI/CD) pipelines. CI/CD pipelines are used to automate software testing and deployment, and mutation testing can be a valuable addition to these pipelines. By integrating mutation testing into the pipeline, developers can automatically evaluate the quality and effectiveness of their test suites, and identify potential bugs and errors before they are deployed to production.
- Automated test case generation for mutation testing is another interest which refers to the process of automatically creating test cases that can detect faults in the software being tested. It can be useful in overcoming challenges that come with manual test case creation. Techniques such as search-based testing using heuristic approaches, such as genetic algorithms, symbolic execution, and random testing, can be employed to generate test cases. Automated test case generation helps to reduce the effort and cost required for creating test



cases. However, it also presents challenges, such as ensuring the effectiveness of the generated test cases and achieving coverage of the possible inputs. With advanced techniques and tools, automated test case generation for mutation testing can be an efficient and effective way to test software.

- Mutant generation is important because it can help improve the quality and reliability of software. However, it is equally important to generate mutants in a way that balances mutation density, maintains equivalence, and ensures coverage for meaningful results. Balancing mutation density means creating a reasonable number of mutants that are sufficient to test the program, but not too many to overwhelm the testing process. Too many mutants can be time-consuming and resource-intensive and lead to diminishing returns. By balancing the mutation density, the testing process can be made more efficient and effective.
- Dealing with equivalent mutants is crucial to ensure that the mutations do not change the program's functionality. Equivalent mutants do not contribute to the testing process and can waste resources. Therefore, it is essential to identify and remove equivalent mutants to avoid skewing the results.

Mutation testing research in computer science is constantly growing to address the above challenges and new developments. New practitioners and researchers in the field are expected to facilitate the application of this area to industrial projects in the future.

#### ***2.4.2. A Hot Topic: Artificial Intelligence Supported Mutation Testing***

It is almost impossible to guarantee that no bugs remain in a piece of software. However, it is still necessary to have adequate test sets in accordance with the test policy during validation activities. At this point, artificial intelligence (AI) methods can be used to learn from errors and produce mutants that mimic possible errors. This improves the quality of the test cases and helps to decrease the time and budget needed. It also supports test automation, enabling rapid generation of new white-box test cases when software changes are made (Cai et al., 2021), and defect prediction, reducing costs while providing high-quality software products (Pachouly et al., 2022).

The application of AI techniques in mutation testing has been studied by various researchers and has recently become an important trend.

Sleuth was one of the early automated test generation tools developed by a team at Colorado State University to determine whether AI planning provided an easy and natural way to generate test sequences (Mraz et al., 1995). The same team members have also shown how the concepts of AI and mutation testing can be combined (von Mayrhauser et al., 2000). As a result, they obtained useful mutants using four best mutation operators. As the subject progressed, they pointed to understanding the acceptance criteria for the adequacy of the mutation and the investigation of what kind of mutations can produce useful error correction test scenarios, together with the knowledge of the application.

AI subfields, such as machine learning (ML) and evolutionary computation (EC), also considered and experimented within the concept of mutation testing and are increasingly being used to enhance the mutation testing process. Artificial neural networks (ANNs), which is a huge part of ML can automate the creation of mutations, other techniques can improve the selection of mutations and can more accurately analyze the results of tests, and EC can help create a smaller group of mutants while still retaining important information. Figure 9 shows the utilization of AI subfields in mutation testing applications.

#### ***2.4.2.1. Machine Learning***

Machine learning techniques can be divided into two parts: Artificial Neural Networks and other techniques.

**Artificial Neural Networks:** An ANN refers to a category of machine learning algorithms that simulate the structure and function of the human brain. These algorithms are designed to perform complex tasks, such as image recognition, speech recognition, and natural language processing, by identifying patterns in large amounts of data. In an ANN, artificial neurons, which are individual processing nodes, are connected and process information by transmitting signals through the connections. The strength of these connections can be modified during training, allowing the network to learn from its experiences.

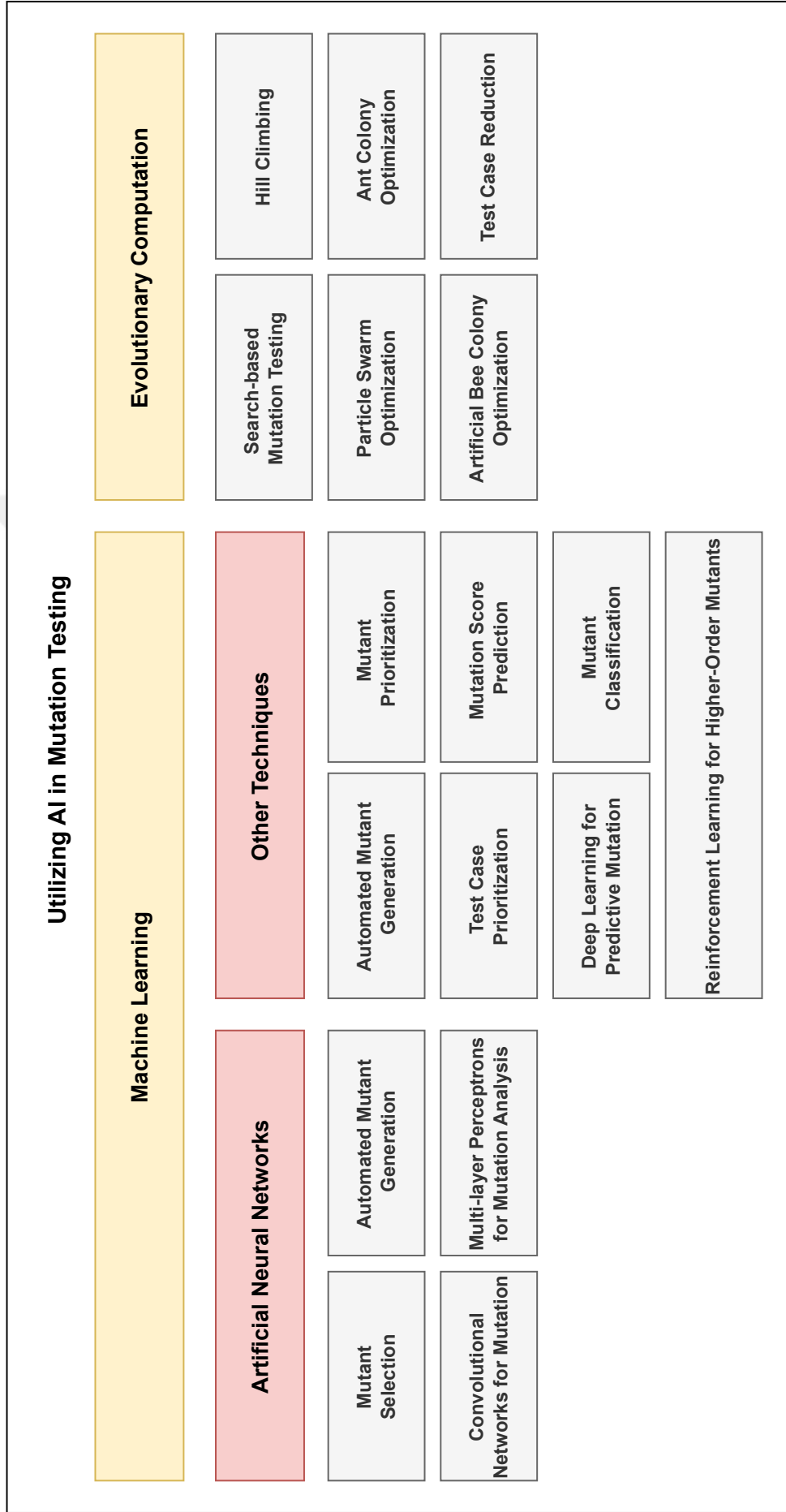


Figure 9. Utilizing AI in mutation testing.

One use of ANNs in mutation testing is to analyze the results of mutation testing and identify patterns in the way that different mutations are detected by the tests. This information could then be used to guide the selection of mutants for further testing, helping to make mutation testing more efficient and effective. Another application of ANNs in mutation testing is to automatically generate new mutants. ANNs could be trained on a source code and then used to generate new mutants that are likely to be effective in finding bugs and weaknesses in the code.

Shen et al. (2018) focused on the impact of mutation on neural networks and the influence of neural depth on mutation analysis. The authors proposed the MuNN method involving five mutation operators, which can calculate the mutation adequacy score. They stated that theirs was the first study to analyze ANNs in the context of mutation testing. The results of the experiments demonstrated that the analysis of mutations in neural networks has distinct characteristics within the domain. This suggests that domain-specific mutation operators are necessary to improve mutation analysis. The results also reveal that the effects of mutation are gradually reduced as the depth of the neurons increases.

Yao et al. (2019) argued that it is challenging to assess the adequacy of Convolutional Neural Network (CNN) applications using traditional testing criteria and presented a new model coverage approach based on mutation testing for CNN by applying it to a classification model called LeNet-5 to evaluate testing accuracy. Their results demonstrated that the Add Fully Connected Layers (AFCL) model is the best local model considering test adequacy. This study seems promising; however, it needs to be further evaluated and validated using a wider range of test subjects.

Klampfl et al. (2020) used a mutated neural network and attempted to distinguish it from its original form using test evaluation techniques. The results of their experiments, which involved the use of Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs), indicated that relying solely on the mutation score is not adequate to identify an adequate quantity of mutants. Their study showed that it works at the configuration level, implying that the test data alone are insufficient for thoroughly testing neural networks. They obtained 99% adequacy for MLP Classifier Saturn and 97% adequacy for MLP Classifier X. As a result, it becomes apparent

that specialized testing suites for neural network libraries are required to significantly improve the mutation score.

The application of ANNs in mutation testing is still in its early stages, and this area has the potential to grow and evolve with technological advancement. ANNs could become a valuable tool for software developers in the future, helping to improve the quality and reliability of software systems.

**Other Techniques:** Other techniques deal with the development of algorithms and models capable of learning from data and making predictions or performing actions without explicit programming. The overall process involves training a model using a large dataset, allowing it to identify patterns and relationships in the data. The model then utilizes these patterns to predict the outcomes for new, previously unseen data.

There are three primary learning categories here: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. In supervised learning, the algorithm is trained using a labeled dataset, where the correct output is known for each input. The algorithm learns how to map between inputs and outputs and can then be used to predict new unseen data. Unsupervised learning involves training an algorithm on an unlabeled dataset, which requires the algorithm to identify patterns and relationships in the data. This type of learning is utilized when the desired output is unknown and is used to uncover hidden structures in the data. Reinforcement learning trains an algorithm through trial-and-error, where the algorithm receives rewards or penalties based on its actions. Over time, the algorithm learns the actions that are most likely to result in the highest rewards.

Currently, discussions of these concepts inevitably include deep learning (DL). DL is a specialized area within the field of ML that is based on the concept of ANNs. The objective of DL is to create complex models that can learn and identify patterns in large amounts of data, such as audio, images, and text. The term “deep” in deep learning refers to the number of hidden layers in the network. This technique involves training a model using a vast dataset that allows it to make predictions based on the input data without the need for explicit programming. In recent times, deep learning has been utilized in various applications such as speech and image recognition, natural language processing, and autonomous vehicles and has produced outstanding results

in these areas. DL represents a high potential area within ML, and its impact on a wide range of industries and applications continues to grow.

There are several ways in which other ML techniques can be utilized in mutation testing. For instance:

- Automation of the mutation generation process, reducing the manual effort required.
- Determination of which mutations are likely to have a greater impact on the code and prioritize them for testing, reducing the manual testing effort.
- Prioritization of test cases based on their ability to detect mutations, allowing testers to focus on the most crucial test cases first.
- Prediction of the mutation score of a program, which measures the efficacy of the test suite in detecting mutations. This can help testers assess the completeness of the test suite and identify areas that require additional testing.

Integration of other ML techniques in mutation testing has the potential to improve the efficiency and effectiveness of the testing process, making it easier for developers to identify and resolve code defects.

Strug and Strug (2012) used a classification algorithm to predict the success of the mutation and assess errors encountered during the experiments. The approach presented in the study was centered on the similarity between mutants. The goal was to minimize the number of mutants that need to be tested by taking advantage of this similarity. The structure of each mutant is analyzed by constructing a hierarchical control flow graph that represents the program's flow, variables, and conditions. Based on this graph, a similarity score is calculated between the mutants. This score is then used to predict the ability of a given test to detect a mutant by applying a classification algorithm. The authors claimed that their results were promising; however, random selection was problematic considering test suites. Therefore, more future work and experimentation is required to validate the approach.

Zhang et al. (2018) proposed predictive mutation term which suggests the prediction of testing results that can be identified without executing mutants. They

built a predictive model using classification algorithms with coverage and mutation operators collected through mutants executed on previous versions of the projects. Their experiments demonstrate that this method improves the efficiency of mutation testing by up to 151.4X. Furthermore, in a study by Mao et al. (2019), additional features and deep learning models were used to show a prediction accuracy of more than 0.85 in 654 projects, and it was shown that it is consistent with the previously proposed study Zhang et al. (2018). These advances seem to produce promising solutions, although it is important to develop more models and compare them with the results of this research.

Ma et al. (2018) proposed a new mutation testing framework specific to DL systems using both source-level and model-level mutation operators in two datasets: MNIST and CIFAR-10. The results show that the source-level mutation testing obtains lower mutation score on model A, but obtains higher mutation score on model B. Hence they concluded that, mutation testing increases the test quality of DL systems and can be further extended to cover more diverse areas of deep learning. All of the discussed applications can still be explored further by combining other subfields of AI, such as vector support machines.

Naeem et al. (2019) used a TensorFlow-based deep learning Keras model to predict whether mutants will survive or die. They applied predictive analysis using program dependency graphs on five open source projects consisting of more than 10 KLOC to minimize accuracy loss. Consequently, both RNN and MLP demonstrated strong effectiveness in scalable mutation testing for mutant prediction.

Durelli et al. (2019) conducted a study to examine the application of ML in software testing. Through a systematic mapping of the relevant literature, the study classified the research based on various aspects such as the testing phase, the type of ML technique utilized, and the type of software being tested. The results indicated a noticeable increase in the use of ML in software testing and its application in different testing phases, including requirement analysis, testing design, test case generation, test case selection, and test execution. The research also highlighted that the most prevalent type of ML applied in software testing is supervised learning, followed by unsupervised learning and reinforcement learning. The study provides

a comprehensive examination of the current state of research in this field and identifies potential avenues for future research.

Panichella and Liem (2021) discussed the usage of ML techniques in mutation testing, due to the similarity of the ML model development process to the test-driven development (TDD) process, where a training algorithm generates a model that fits the data points to labels within a certain degree of precision. However, when considering mutation testing techniques for ML systems through TDD, the distinction between production and test code is unclear, and the authenticity of mutation operators can be called into question. Based on these observations, the authors suggested several steps to better align mutation testing techniques for ML with the paradigms and terminology of classical mutation testing, such as competent programmer hypotheses and the coupling effect. The study noted that there is no clear indication of how ML approaches should be applied in mutation testing concepts. They also suggested that, when ML is applied to mutation testing, important considerations are mutation operators for ML, the system being tested and small mutant term.

Tambon et al. (2023) claimed that adapting mutation testing to DL systems would make systems more verifiable than ever. DL testing is difficult due to its random behavior. Although some progress has been made in using mutation testing for supervised learning in DL, little has been done for reinforcement learning, which is a crucial part of DL but operates differently from supervised learning. The authors proposed the RLMutation framework for using mutation testing in reinforcement learning, which identifies a set of mutation operators relevant to reinforcement learning and produces test cases. This allowed for a comparison of different definitions of the behavior of mutation operators and higher-order mutations. The results revealed that the selection of the mutation killing definition can determine whether a mutation is killed and how the test cases are generated. The framework successfully generated higher-order mutations using DQN, A2C, and PPO models on LunarLander and CartPol datasets with unique features that can improve testing in reinforcement learning systems.

A more recently study by Khanfir et al. (2023) argued that pre-trained language models can be used to automatically generate meaningful mutations with a better



chance of uncovering bugs in a program. They developed a predefined language model called  $\mu$ BERT, based on fault injection. The proposed method uses these models to prioritize the execution of test cases, so that the most important tests are run first, thus reducing the total time required to perform a full mutation test. The results of the experiments show that the effectiveness of  $\mu$ BERT was 82.92% compared to other models: Pit-all, Pit-default, and Pit-rv-all. And, it can significantly reduce the time required to perform mutation testing, while still effectively uncovering bugs in the program. The authors stated that there is no research group currently working with ML approaches in software testing and that their study sheds light on the usefulness of these concepts in mutation testing.

#### **2.4.2.2. Evolutionary Computation**

EC is a subfield of AI that utilizes the concepts of natural selection and genetics to optimize and resolve complex issues. This approach models problem-solving as a natural selection process, where a population of potential solutions evolves over time through selection and recombination of the best solutions, ultimately leading to the discovery of a satisfactory solution.

Various techniques fall under the umbrella of EC including genetic algorithms, genetic programming, evolution strategies, and particle swarm optimization, to name a few. These techniques have been employed in diverse areas such as optimization, machine learning, robotics, and control systems. EC offers a biologically inspired solution to problem-solving in AI and has been applied to a wide range of applications with notable success.

EC can play a role in mutation testing by automating the process of generating and evaluating new mutations. For example, a genetic algorithm can be utilized to evolve the set of mutations applied to the program, with the objective of maximizing the detection of these mutants by the test suite. This approach can lead to the discovery of more effective mutations and a more complete test suite. Thus, it is also known as "*search-based mutation testing*".

In 2017, Jatana et al. (2017) conducted a systematic review to assess trends in search-based mutation testing. Their analysis covered 43 papers, diving deeper into

18 of them. Their findings indicated that search-based methods are primarily effective for both generating mutants and optimizing test cases. Another comprehensive review was carried out by Silva et al. (2017). Their focus was on the application of meta-heuristic methods in search-based mutation testing. They examined 49 papers related to the creation of test data and another 15 concerning mutant generation. The main techniques examined were genetic algorithms, hill climbing, and NSGA-II.

Souza et al. (2016) proposed a hill climbing algorithm to optimize the mutation score, and experimented with C programs that include dynamic data structures and pointers. To overcome the difficulties, the authors implemented constraints. The results showed that the hill climbing algorithm outperforms other algorithms in generating test data for mutation testing and is able to produce high mutation scores, and it was concluded that this approach for test data generation in mutation testing is worth exploring, but, however, it needs further experimentation and different datasets.

Jatana and Suri (2020) compared the performance of particle swarm optimization and genetic algorithm in generating test data for mutation testing. The objective of the evaluation was to evaluate the effectiveness of particle swarm optimization and GA in optimizing the mutation score. The results indicated that both particle swarm optimization and the genetic algorithm can be applied effectively to mutation testing, with particle swarm optimization slightly outperforming the genetic algorithm in terms of optimization of the mutation score.

Mishra et al. (2022) proposed a method to minimize the size of the test dataset while maximizing the mutation score by removing duplicate test cases that cover the same mutants. The method was applied to the same software under the test used for path testing and has been shown to effectively cover a maximum number of mutants with a minimum number of test cases and also to be capable of generating a test suite that provides maximum path coverage while requiring fewer test cases than other algorithms. As a result, the authors created an optimal test suite with the highest adequacy, achieved by using a genetic algorithm to find the maximum mutation coverage and eliminate redundant test cases. In the future, more tools should be added to increase the effectiveness and verifiability of these results.

A recent study by Arasteh et al. (2022) used an artificial bee colony optimization

algorithm to identify the paths most susceptible to faults. Following this, they applied mutation operators to the identified paths for analysis. Their methodology led to a reduction of approximately 28% in the total number of mutants. The authors proposed that widely used Java mutation testing tools such as MuJava and Jester could leverage this strategy to generate mutants at a lower computational cost.

AI has been increasingly applied in mutation testing to improve its efficiency and accuracy. The use of AI can help to automate the process of detecting nonequivalent mutants, which reduces the need for manual effort and makes the testing process more efficient. Moreover, AI techniques, such as ANNs, ML, and EC, can be used to develop models that can accurately classify mutants as equivalent or non-equivalent. These models can be trained on large mutant datasets, which can improve the accuracy of the classification results. In general, the application of AI in mutation testing has the potential to greatly enhance the effectiveness and efficiency of the testing process.

#### ***2.4.3. State of the Art Models for Mutation Testing Using AI***

In recent years, significant progress has been made in developing state-of-the-art (SOTA) models that take advantage of AI in mutation testing. These models aim to automate and optimize the mutation testing process, leading to improved fault detection and more efficient testing methodologies. By utilizing AI algorithms, these SOTA models can effectively generate, evaluate, and prioritize mutants, enabling software developers and testers to identify critical faults and vulnerabilities in their code.

One such approach is **DeepMutation**, which uses neural networks to automatically generate and evaluate mutants. By combining mutation operators and neural networks, DeepMutation surpasses traditional methods in its ability to identify faults.

Another approach, known as **MutateGCN**, focuses on graph-structured code and utilizes graph convolutional networks (GCNs). By representing the code as a graph and employing GCNs, MutateGCN enhances mutation testing, particularly for programs with graph-based structures.

**MuDiff** takes a machine learning-based approach, using diff analysis to detect semantic discrepancies between mutants and the original code. This technique

prioritizes mutants based on their changes, improving the efficiency and effectiveness of mutation testing.

**DeepMutation++** builds upon the success of DeepMutation by incorporating additional mutation operators and leveraging transfer learning. This extension enables DeepMutation++ to enhance fault detection capabilities and exhibit good generalization in various software projects.

## 2.5. Conclusion

Although the area of mutation testing appears to have reached its mature state, the number of publications and studies on the topic is increasing regardless. We have discussed equivalent mutants, the many computational cost reduction techniques that have been proposed to reduce them, explained which AI techniques can be used to enhance mutation testing, and examined current studies. However, these are still ongoing issues that need attention. Recent work in the area also tends to focus on developing source code applications that can use mutation testing in industry supported by AI, increase the effectiveness of mutation analysis, and widen its areas of application in software engineering.

The next focus in this thesis is on mutation testing tools for evaluation, search-based mutation testing using evolutionary computation, and higher-order mutation testing applications. In the next sections, we discuss, analyze, and experiment in these areas of mutation testing.

As a result, **RQ1** outlined in Section 1.4 is answered in this chapter: “*What are the existing studies, prevailing trends, challenges, and advancements in the field of mutation testing as observed in recent academic and industry practices?*” All aspects and challenges of mutation testing are explored together with the recent interests are identified. There is a current trend for using mutation testing with AI practices. The findings of this chapter have been synthesized into a survey article (Uzunbayir and Kurtel, 2024).

## CHAPTER 3: AN ANALYSIS ON MUTATION TESTING TOOLS

This chapter aims to answer **RQ2** outlined in Section 1.4: “*How can existing mutation testing tools for C# be compared in terms of their features and effectiveness?*”

Mutation testing is resource intensive, requires a large number of mutants, making it necessary to automate the mutant creation process. Many tools for mutation testing exist both in the literature and in the field, each supporting a different programming language. Most of the tools available were implemented using C, C++, and Java programming languages, which support strong mutation, weak mutation, higher-order mutation, as well as object-oriented operators specific to object-oriented programming languages such as Java and C# (Papadakis et al., 2017). Since comparative studies for C# is rather limited in the literature compared to other object-oriented languages, our focus is on it. We list the most popular tools, discuss object-oriented operators, investigate tools for C#, by examining their unique attributes to help testers select the most suitable tool through a comparative evaluation, and present a case study to select one of the available tools for our experiments used this thesis.

### 3.1. *Mutation Testing Tools for Different Programming Languages*

Mutation testing tools offer several advantages to software developers. These benefits include improved test suite effectiveness, early bug detection, reduced cost, faster software development, and improved code quality. Additionally, they can improve software development efficiency by automating the testing process and identifying defects more quickly. They vary in features, capabilities, and supported languages. Some are open-source and freely available, while others are commercial products. Choosing the right tool for a project depends on the programming language used, the desired level of test coverage, and the available resources.

In this section, we cover 56 mutation testing tools. Note that there is a wide range of tools in the literature, and from these, we selected the most popular ones proposed or updated between 2013 and 2023. Moreover, we provide at least one tool for a specific language as an example. Table 5 lists popular mutation testing tools and their

Table 5. Mutation testing tools.

Programming Language	Tool
Alloy	Mualloy
C	Mutgen, ESTP, SMT-C, Milu
C++	MuCPP, Mutate_CPP
C and C++	Mull, Proteum, PlexTest, Certitude, CCMutator
C#	Nester, Stryker, NinjaTurtles, VisualMutator, PexMutator, CREAM
Fortran	Mothra
Go	Gremlins, Gomutate
Haskell	Mucheck, Fitspec
HTML	REDECHECK, WebMuJava
Java	PIT, Jester, ByteME, MutMut, Bacterio, Jumble, muJava, MuClipse, LittleDarwin, Judy, JavaLanche
JavaScript and Node.js	Mutode, Mutandis
PHP	MutateMe, Hambug, InfectionPHP
PL/SQL	Muplsql
Ruby	Mutant, Heckle
R	Mutant
Rust	Cargo Mutants
Python	Mutmut, MutPy, Mutatest, CosmicRay
Scala	Stryker4s, Scalamu
SQL	SQLMutation, SchemaAnalyst, JDAMA
Swift	Muter

supported programming languages. According to the table, Java and C# have many different tools for mutation testing. Python supports four different tools, as it is also a popular language. Many old and new programming languages have their own tool support. Whenever a new programming language is proposed, its associated mutation testing tool is also proposed, indicating that researchers are working on implementing mutation support to the new trends.

Mutation testing tools automatically insert changes into code by mutation operators and then check how well the test suite can find these changes. These tools give useful information about how effective the test suite is and point out parts of the code that might have hidden bugs by automating the process.

Instead of trying to review all mutation testing tools, which is not practical, we focus on C# because it is popular, being widely used in many different applications, there are a lot of tools available for it, and not much comparative research has been done in this area yet.

### 3.2. Mutation Operators for C#

Mutation operators modify the original program to generate mutants. Common mutation operators are applicable to multiple programming languages (Boubeta-Puig et al., 2011). Table 6 details some of these operators specifically used for Fortran in conjunction with the Mothra tool.

Table 6. Some traditional mutation operators (Jia and Harman, 2010).

<b>Mutation Operator</b>	<b>Description</b>
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for scalar variable replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
SVR	Scalar variable replacement
UOI	Unary operator insertion

Conventional mutation operators, typically used in procedural programming languages, are quite broad in nature. In contrast, object-oriented programming languages need a more extensive set of mutation operators to adequately test unique features such as encapsulation, inheritance, and polymorphism (Kim et al., 2000).

Early studies on mutation operators for object-oriented languages centered around Java. Following the successful application in Java, these operators were also adapted for use in C# (as detailed in Table 7).

Table 7. Some mutation operators for C# (Derezińska, 2006).

<b>Mutation Operator</b>	<b>Description</b>
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	Base keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	New method call with child class type
JTD	This keyword deletion
JSC	Static modifier change
EAM	Accessor method change
EMM	Modifier method change
MNC	Method name change
MCO	Member call from another object
ORO	Operand replacement operator
EMO	Expression modification operator
RSR	Return statement replacement
SAN	Statement analysis
RFI	Referencing fault insertion
EHR	Exception handler removal
EHC	Exception handling change
EXS	Exception swallowing
DMO	Delegated method order change
OID	Overriding indexer deletion
NDC	Namespace declaration change
PRM	Property replacement with member field



Certain operators create easily killable mutants, which makes them not useful, while others produce mutants that remain unkillable even after applying the entire test suite. These indestructible mutants, known as equivalent mutants, yield the same output as the original program. There are various strategies to address this problem, including detecting, reducing, and classifying equivalent mutants (Orzeszyna, 2011). A positive aspect of object-oriented operators is that they typically generate a smaller number of mutants compared to traditional operators (Ma et al., 2005).

### **3.3. Mutation Testing Tools for C#**

#### **3.3.1. Nester**

Nester (2002), the pioneering mutation testing tool for C#, requires the .NET Framework 2.0 and is compatible only with the NUnit framework. Its latest version features an XML-based grammar and a C# parser, enhancing its ability to dynamically read grammar tokens for faster mutation analysis. It includes a GUI, influenced by the SharpDevelop IDE, with a text editor.

In its project view mode, Nester displays project files with a clear color-coding system: mutants killed in green, mutants surviving in red, and code sections not covered by test cases in blue. This visual approach effectively demonstrates code evaluation post-testing.

Mutation Information displays test results in raw XML format, with HTML and Excel also supported. The results are presented in a square table format, making it easier to see how many mutants each test case eliminates. Nester supports various mutation operators, including addition, multiplication, shift, and or, relational, inclusive or, exclusive or, conditional and, and conditional or operators.

#### **3.3.2. Stryker**

Stryker (2014), also known as Stryker.NET, is another C# mutation testing tool that can be installed via the NuGet package manager. It works with .NET CoreApp 1.1+ and .NET Framework 4.5+, requiring .NET core runtime 2.2+ for seamless operation. Stryker supports a range of operators,

- Arithmetic operators (+, -, \*, /, %)
- Assignment statements (+ =, - =, \* =, / =, % =, <<=)
- Boolean literals (*true*, *false*)
- Checked statements (*checked*(1 + 2))
- Equality operators (>, <, >=, <=, ==, !=)
- Logical operators (&&, ||)
- LINQ methods (*first*(), *last*(), *skip*())
- String literals (“”)
- Unary operators (+*var*, -*var*, *var*)
- Update operators (*var* ++, *var* --, ++ *var*, -- *var*)

Stryker features five different reporters for result display which can be listed as follows:

- **HTML Reporter:** Outputs an HTML file for a visual representation of the project and mutations, suitable for large projects with numerous mutants.
- **Console Reporter:** Visually displays mutations post-test run, without creating new files, ideal for smaller projects needing quick runs.
- **Progress Reporter:** Shows the real-time status of the mutation test, including a time counter for the remaining test time, making it appropriate for larger projects.
- **Console Dots Reporter:** Indicates the number of mutants that have completed testing without impacting system performance, useful in build servers.
- **JSON Reporter:** Outputs a JSON file for a visual representation of mutations from the last test run.

### 3.3.3. *NinjaTurtles*

NinjaTurtles (2014), an open-source and user-friendly C# mutation testing framework, supports method-by-method testing to improve test suite quality and is conducive to test-driven development. It seamlessly integrates with existing C# unit test frameworks.

The installation options for NinjaTurtles include using the NuGet package manager or downloading binary files. Its source code is available on CodePlex for those who need it. The framework is equipped with a variety of operators, such as arithmetic and bit-wise operators, branch substitution, conditional boundary, sequence point deletion, and substitutions for reading and writing variables, parameters, and fields of the same type.

NinjaTurtles delivers test results in an XML file format and is compatible with numerous unit testing frameworks, including NUnit, xUnit, and MSTest.

### 3.3.4. *VisualMutator*

VisualMutator (2013) is another mutation testing tool for C# that integrates as an extension within the Visual Studio IDE. The tool mutates code during post-compilation and is capable of second-order mutation using its built-in operators. It is compatible with NUnit and XUnit frameworks and can output results to an XML document if required. Its integration as a Visual Studio tool window makes it user-friendly, and includes functionality to mark equivalent mutants.

The tool is well documented on its website and supports a range of traditional and object-oriented mutation operators:

- Super/base keyword deletion
- Delegated method change
- Method delegated for event handling change
- Accessor modifier method change
- Exception handler removal

- Exception handling change
- Exception swallowing
- This keyword insertion
- Member variable initialization deletion
- Member call from another inherited class
- Reference assignment with other compatible type

Additionally, VisualMutator offers a command prompt feature for situations where interactivity is not necessary.

### **3.3.5. *PexMutator***

PexMutator (2010) is the fifth mutation testing tool that uses dynamic symbolic execution for test generation. Initially, it transforms the original program into a meta-program featuring weak mutant killing constraints. Subsequently, PexMutator leverages its engine to generate test cases, which are then used in conjunction with the weak constraints to eliminate the mutants.

This tool utilizes five effective mutation operators: relational, arithmetic, logical, binary, and unary operators.

### **3.3.6. *CREAM***

The Creator of Mutants CREAM (2008) is the last mutation testing tool discussed in this study for C# programs. To operate effectively, it requires the .NET Framework 4.0 and Visual Studio. Additionally, it is compatible with NUnit, NCover, SVN Client, and Excel.

CREAM's most recent version encompasses structural operators as well as a variety of object-oriented operators. Detailed information about these features is available in its user manual. To address the issue of equivalent mutants, CREAM offers optional cost-saving techniques such as selective mutation and mutant clustering.

### ***3.4. An Analysis of Mutation Tools for C# Based On Tool Characteristics***

In this section, we conduct an analysis based on tool characteristics. For this reason, we perform an analysis and comparison between C# tools discussed in the previous section, we define four tool characteristics: 1) open-source, 2) object-oriented operators, 3) mutation generation level, and 4) mutant format based on a study by Halabi and Shaout (2016). Open-source implies that the source code of the tool is publicly accessible and modifiable. Object-oriented operators determine if the tool supports specific operators for object-oriented programming, as discussed in Section 3.2. Mutation generation level refers to optimization techniques during runtime, including source code, byte-code, or compiler-based methods. Mutant format addresses the efficiency aspect, questioning whether mutants are stored in separate files or in memory.

We have introduced seven additional characteristics common among the tools: 1) traditional operators, indicating support for standard mutation operators; 2) unit testing frameworks, specifying compatible frameworks; 3) GUI, denoting the presence of a graphical user interface for user convenience; 4) detailed user manual, assesses the availability of an online guide for users; 5) LINQ support, checks if tools facilitate LINQ queries, a C# syntax for data retrieval; 6) result display format, examines the types of formats supported for displaying outcomes; 7) equivalent mutant reduction, which is essential for mutation testing efficiency, to find if any tools offer features to handle equivalent mutants.

The results of our analysis are presented in Table 8. This analysis revealed that all the evaluated tools support traditional mutation operators. However, only NinjaTurtles, VisualMutator, and CREAM offer object-oriented operators for managing object-oriented aspects of the language. All tools are open-source, with the exception of VisualMutator, which is being developed by the Institute of Computer Science at Warsaw University of Technology. Multiple unit testing frameworks are supported by some tools; for instance, NUnit is compatible with Nester, NinjaTurtles, VisualMutator, and CREAM, while Stryker works with VSTest. There was no information available regarding CREAM's support.

Table 8. Feature comparison of mutation testing tools for C#.

<b>Characteristics</b>	<b>Nester</b>	<b>Stryker</b>	<b>NinjaTurtles</b>	<b>VisualMutator</b>	<b>PexMutator</b>	<b>CREAM</b>
Traditional operators	YES	YES	YES	YES	YES	YES
Object-oriented operators	NO	NO	YES	YES	NO	YES
Open-source	YES	YES	YES	NO	YES	YES
Unit testing frameworks	NUnit	VSTest	NUnit, XUnit, MSTest	NUnit, XUnit	N/A	NUnit, NCover
Mutant generation level	Source code	Byte-code	Compiler-based	Compiler-based	Compiler-based	Compiler-based
GUI	YES	NO	NO	YES	NO	YES
Detailed user manual	NO	YES	YES	YES	NO	YES
Mutant format	Separate file	In memory	Separate file	Separate file	In memory	Separate file
LINQ support	NO	YES	NO	NO	NO	NO
Result display options	XML, HTML, Excel	HTML, JSON	XML	XML	N/A	Excel
Equivalent mutant reduction	NO	NO	NO	NO	NO	YES

In terms of mutant generation, Nester operates at the source code level, Stryker at the byte-code level, and others at a compiler-based level. Having a graphical user interface (GUI) simplifies usage, eliminating the need for command prompt memorization. Nester, VisualMutator, and CREAM are equipped with GUIs, with some such as Nester offering both GUI and non-GUI options.

The usability of tools is greatly influenced by the availability of comprehensive documentation. Stryker, NinjaTurtles, VisualMutator, and CREAM provide detailed guides on their websites. Stryker is the only tool that supports LINQ. Regarding result display formats, many tools are optional for XML, while CREAM uses Excel, and Stryker uses HTML and JSON. No information was found about PexMutator's capabilities in this area. Finally, CREAM stands out for its support of equivalent mutant reduction, incorporating features such as strong and weak mutation interfaces, selective mutation, and mutant classification. However, there is no further support for CREAM, therefore, VisualMutator becomes the most useful and robust tool among all.

### ***3.5. A Case Study: Cross-Evaluation of the Tools***

In the following two chapters of this research, a particular mutation testing tool for C# has been selected for mutant generation. This choice was the result of a case study aimed at evaluating various tools to determine the most suitable one for our research goals. We identified five tools analyzed in the previous Section 3.4 excluding CREAM. The decision not to include the CREAM tool in our C# mutation testing toolkit was based on several reasons. The crucial one for its exclusion is its discontinued support. In the dynamic domain of software engineering, especially with a prevalent language such as C#, using tools that no longer receive updates or support can introduce significant drawbacks. Tools without ongoing support might not work well with the latest C# versions or frameworks, leading to potential integration issues. They may also miss essential updates or fixes for known bugs, affecting the test's effectiveness and accuracy. Therefore, selecting a tool that benefits from active maintenance ensures that our research remains current, efficient, and aligned with the latest standards and methodologies in software testing.

### 3.5.1. Methodology

This case study included a detailed comparison of five selected mutation testing tools. The primary objective of this cross-evaluation is to determine the extent to which a test set that can be killed by one tool can also be killed by other tools. This comparison is crucial as it reveals the strengths and weaknesses of each tool, based on their mutation operators, fault detection capabilities, and overall compatibility with different codebases. This rigorous evaluation, based on quantitative data guided us to choose the tool that best meets the specific needs and objectives of our study.

The secondary goal of this case study is to ensure a balanced comparison among the tools. Test cases in the test suite are designed to kill all killable mutants. A special mutant set called a *disjoint mutant set* is required for these experiments (Kintis et al., 2018). A disjoint mutant set is the minimum subset of mutants that need to be killed in order to kill the original set. In other words, they do not contain any equivalent mutants. Equivalent mutants produced by the tools have been manually identified and subtracted from all mutants. Mutation testing is applied independently using each tool. Therefore, one test set for each subject program, a total of eight test sets are generated manually. Each test case was removed one by one and checked if the mutation score decreased or not. To produce an accurate disjoint mutant set, it was necessary to remove redundant test cases. At the end, a reference mutant set has been constructed from the disjoint mutants sets considering all tools.

### 3.5.2. Research Questions

This case study aims to achieve an understanding of the effectiveness of mutation testing tools for C#. Effectiveness in this section is defined as the ability of a mutation testing tool to kill more mutants. The tool that demonstrates a higher mutant kill rate is considered more effective. We delve deeper by dividing **RQ2** into two distinct but interrelated sub-research questions:

- **RQ2a:** Is there a C# mutation testing tool that is more effective than the others?
- **RQ2b:** How can these tools be fairly compared?



### 3.5.3. *Subject Programs*

To evaluate the effectiveness of the chosen tools, eight widely-used mutation testing subject programs were selected, each differing in size and complexity. Some of these programs were initially employed in Java-based experiments in a study by Rani et al. (2015) and have been adapted into C# for this research, complemented by additional similar common mutation testing programs. Table 9 presents a list of these programs, along with their descriptions and code sizes.

Table 9. Subject programs for the case study.

<b>Subject Program</b>	<b>Description</b>	<b>Lines of Code</b>
FindMin	Returns the minimum number element from a list.	15
Palindrome	Finds if the word is a palindrome.	17
QuickSort	A sorting routine.	25
QuadraticSolver	Decides whether a given equation is quadratic or not.	73
TriangleType	Finds the type of a triangle.	123
PrintPrimes	Finds and prints n prime integers.	49
CalculateDays	Finds the number of days between dates.	51
PatternIndex	Finds index of pattern in a subject string.	85

### 3.5.4. *Results*

Our detailed study includes five separate tables, each dedicated to contrasting a particular mutation testing tool with its competitors. Table 10 offers a comparison of NinjaTurtles with various other tools, highlighting performance indicators. Table 11 puts Stryker in focus, comparing its functionality and performance efficiency with other available tools. Table 12 is devoted to the analysis of Nester, comparing its effectiveness and ease of use with others. In Table 13, we examine visualMutator, analyzing how it compares and contrasts with other mutation testing tools. Lastly, Table 14 examines PexMutator. Taken together, these tables provide a detailed perspective, allowing a better understanding of the landscape of mutation testing tools in C# in Table 15.

Table 10. NinjaTurtles vs. other tools.

Subject Program	NinjaTurtles							
	Stryker		Nester		VisualMutator		PexMutator	
	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants
FindMin	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Palindrome	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	97.00%	84.33%
QuickSort	98.66%	76.86%	96.82%	87.24%	94.40%	90.01%	96.92%	87.19%
Quadratic	90.71%	84.01%	100.00%	100.00%	94.58%	85.12%	100.00%	100.00%
TriangleType	100.00%	100.00%	100.00%	100.00%	100.00%	90.39%	100.00%	73.22%
CalculateDays	99.25%	98.22%	100.00%	100.00%	95.55%	100.00%	100.00%	94.53%
PrintPrimes	100.00%	100.00%	100.00%	100.00%	97.18%	91.25%	92.00%	82.33%
PatternIndex	93.82%	77.20%	92.46%	88.53%	96.03%	92.78%	94.81%	90.25%
<b>Average</b>	<b>97.81%</b>	<b>92.04%</b>	<b>98.66%</b>	<b>96.97%</b>	<b>97.22%</b>	<b>92.78%</b>	<b>97.59%</b>	<b>88.98%</b>

Table 11. Stryker vs. other tools.

Subject Program	Stryker							
	NinjaTurtles		Nester		VisualMutator		PexMutator	
	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants
FindMin	95.66%	90.21%	88.63%	80.44%	100.00%	95.55%	96.22%	92.15%
Palindrome	100.00%	100.00%	100.00%	100.00%	88.22%	85.46%	99.00%	92.25%
QuickSort	92.44%	90.48%	96.82%	87.24%	94.40%	90.39%	96.92%	87.19%
Quadratic	94.88%	84.01%	88.59%	78.56%	94.58%	85.12%	100.00%	100.00%
TriangleType	89.25%	82.15%	88.45%	80.77%	100.00%	100.00%	100.00%	77.66%
CalculateDays	91.55%	85.22%	92.55%	78.58%	100.00%	100.00%	100.00%	88.96%
PrintPrimes	100.00%	100.00%	100.00%	100.00%	97.18%	91.25%	92.00%	65.95%
PatternIndex	91.25%	78.98%	89.27%	76.25%	96.03%	85.44%	88.98%	78.88%
<b>Average</b>	<b>94.38%</b>	<b>88.88%</b>	<b>93.04%</b>	<b>85.23%</b>	<b>96.30%</b>	<b>91.65%</b>	<b>96.63%</b>	<b>85.38%</b>

Table 12. Nester vs. other tools.

Subject Program	Nester							
	NinjaTurtles		Stryker		VisualMutator		PexMutator	
	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants
FindMin	93.58%	90.65%	92.63%	90.02%	99.65%	99.05%	100.00%	100.00%
Palindrome	88.69%	75.98%	89.36%	81.56%	92.63%	88.63%	100.00%	84.33%
QuickSort	96.54%	89.55%	96.55%	77.52%	100.00%	100.00%	95.63%	88.77%
Quadratic	91.58%	88.65%	89.56%	82.22%	95.66%	87.64%	100.00%	100.00%
TriangleType	100.00%	100.00%	78.63%	65.26%	89.63%	85.50%	100.00%	73.22%
CalculateDays	99.45%	88.62%	100.00%	100.00%	100.00%	100.00%	88.63%	78.63%
PrintPrimes	93.66%	91.88%	100.00%	100.00%	100.00%	100.00%	90.65%	86.63%
PatternIndex	95.69%	79.56%	80.56%	73.65%	93.63%	89.26%	94.66%	91.05%
<b>Average</b>	<b>94.90%</b>	<b>88.11%</b>	<b>90.91%</b>	<b>83.78%</b>	<b>96.40%</b>	<b>93.76%</b>	<b>96.63%</b>	<b>87.83%</b>

Table 13. VisualMutator vs. other tools.

Subject Program	VisualMutator							
	NinjaTurtles		Stryker		Nester		PexMutator	
	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants
FindMin	96.63%	91.66%	85.69%	88.95%	90.85%	85.74%	95.63%	91.56%
Palindrome	100.00%	100.00%	91.52%	85.96%	100.00%	100.00%	95.00%	94.63%
QuickSort	88.65%	78.69%	100.00%	100.00%	92.65%	86.52%	85.64%	80.56%
Quadratic	92.88%	89.68%	100.00%	100.00%	90.56%	84.58%	78.95%	71.57%
TriangleType	89.66%	81.97%	88.56%	80.47%	100.00%	100.00%	100.00%	84.65%
CalculateDays	96.78%	92.85%	78.50%	69.56%	100.00%	100.00%	100.00%	93.59%
PrintPrimes	100.00%	100.00%	87.99%	76.52%	91.56%	82.69%	93.65%	87.44%
PatternIndex	89.65%	78.95%	85.45%	80.56%	85.45%	75.98%	92.63%	88.56%
<b>Average</b>	<b>94.28%</b>	<b>89.23%</b>	<b>89.71%</b>	<b>85.25%</b>	<b>93.88%</b>	<b>89.44%</b>	<b>92.64%</b>	<b>86.57%</b>

Table 14. PexMutator vs. other tools.

Subject Program	PexMutator							
	NinjaTurtles		Stryker		Nester		VisualMutator	
	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants	All Mutants	Disjoint Mutants
FindMin	100.00%	100.00%	100.00%	100.00%	95.63%	92.36%	88.52%	81.64%
Palindrome	100.00%	100.00%	100.00%	100.00%	94.78%	90.06%	91.00%	84.12%
QuickSort	89.57%	85.63%	96.82%	87.24%	100.00%	100.00%	95.79%	92.63%
Quadratic	94.56%	89.74%	76.52%	75.26%	95.96%	87.49%	84.96%	82.25%
TriangleType	78.56%	69.96%	100.00%	100.00%	78.63%	74.62%	100.00%	89.63%
CalculateDays	89.65%	84.45%	100.00%	100.00%	79.63%	68.92%	100.00%	88.58%
PrintPrimes	90.63%	87.85%	100.00%	100.00%	91.62%	90.01%	100.00%	81.69%
PatternIndex	89.52%	84.44%	90.63%	85.46%	96.26%	89.52%	94.63%	91.45%
<b>Average</b>	<b>91.56%</b>	<b>87.76%</b>	<b>95.50%</b>	<b>93.50%</b>	<b>91.56%</b>	<b>86.62%</b>	<b>94.36%</b>	<b>86.50%</b>

Table 15. Relative test suite evaluation.

<b>NinjaTurtles Test Suite</b>	<b>Stryker</b>	<b>Nester</b>	<b>VisualMutator</b>	<b>PexMutator</b>
All Mutants	94.38%	94.90%	94.28%	91.56%
Disjoint Mutants	88.88%	88.11%	89.23%	87.76%
<b>Stryker Test Suite</b>	<b>NinjaTurtles</b>	<b>Nester</b>	<b>VisualMutator</b>	<b>PexMutator</b>
All Mutants	97.81%	90.91%	89.71%	95.50%
Disjoint Mutants	92.04%	83.78%	85.25%	93.56%
<b>Nester Test Suite</b>	<b>NinjaTurtles</b>	<b>Stryker</b>	<b>VisualMutator</b>	<b>PexMutator</b>
All Mutants	98.66%	93.04%	93.88%	91.56%
Disjoint Mutants	96.97%	85.23%	89.44%	86.62%
<b>VisualMutator Test Suite</b>	<b>NinjaTurtles</b>	<b>Stryker</b>	<b>Nester</b>	<b>PexMutator</b>
All Mutants	97.22%	96.30%	96.40%	92.64%
Disjoint Mutants	92.78%	91.65%	93.76%	86.57%
<b>PexMutator Test Suite</b>	<b>NinjaTurtles</b>	<b>Stryker</b>	<b>Nester</b>	<b>VisualMutator</b>
All Mutants	94.59%	96.63%	92.64%	96.15%
Disjoint Mutants	88.98%	85.38%	86.57%	87.83%

Table 16. Average results for mutation score

<b>Mutation Adequate Test Suites</b>	<b>Average</b>
VisualMutator	95.64%
PexMutator	95.00%
Nester	94.29%
NinjaTurtles	93.78%
Stryker	93.48%

Table 17. Average results for disjoint mutant sets.

<b>Mutation Adequate Test Suites</b>	<b>Average</b>
VisualMutator	91.19%
Nester	89.57%
Stryker	88.66%
NinjaTurtles	88.50%
PexMutator	87.19%

**RQ2a:** *Is there a C# mutation testing tool that is more effective than the others? To answer RQ2a, we can ask if there is a tool that subsumes the others? Or, is there a tool that kills all killable mutants of other tools? There can be two answers; yes, there is a single tool that is superior to others. Or, no, there is not a single tool that is superior to others, which means the tools are not comparable.*

According to the results in Table 16, and Table 17, there is no tool that subsumes the rest. All generated test suites have reduced effectiveness when compared to mutants with other tools. Stryker test suite performs the worst compared to the others on average. VisualMutator test suite performs the best when compared to the others on average.

**RQ2b:** *How can these tools be fairly compared?*

To answer RQ2b, a special mutant set called a reference mutant set based on disjoint mutant sets is required. It helps to classify the tools in terms of their effectiveness. Table 18 shows the number of mutants in the reference mutant sets for each subject program. As a result, a total of 184 mutants are selected in the reference mutant sets. The performance is different for all methods; however, on the average results can be discussed. Based on the results in Table 19, VisualMutator seems to perform the best among the listed tools with a high average of 93.47%. PexMutator also has a respectable average score of 91.30%. Although not as high as VisualMutator,

Table 18. Reference mutant set.

Subject Program	Number of Disjoint Mutants					Number of Mutants
	NinjaTurtles	PexMutator	Nester	Stryker	VisualMutator	Reference Set
FindMin	4	3	4	3	4	4
Palindrome	6	9	11	7	9	11
QuickSort	14	13	14	13	12	14
Quadratic	16	19	14	14	15	19
TriangleType	19	22	21	18	24	24
CalculateDays	26	29	25	25	33	33
PrintPrimes	33	35	30	29	31	35
PatternIndex	41	38	40	41	44	44
<b>TOTAL</b>	<b>159</b>	<b>168</b>	<b>160</b>	<b>150</b>	<b>172</b>	<b>184</b>

Table 19. Tool rankings.

Mutation Adequate Test Suites	Average
VisualMutator	93.47%
PexMutator	91.30%
Nester	86.95%
NinjaTurtles	86.41%
Stryker	81.52%

it still indicates that PexMutator is effective in mutation testing. Nester falls in the middle with a score of 86.95%. It is not the highest, but it is certainly not the lowest either. This suggests that it is effective in detecting faults or vulnerabilities in the software being tested. Stryker has the lowest score among the listed tools with an average of 81.52%. This suggests that it might not be as effective compared to the others. Therefore, we selected VisualMutator to generate mutants for our study.

### 3.6. Conclusion

This chapter proved to be crucial for the subsequent chapters of this study, as it facilitated the identification of the most effective and practical mutation testing tool for C# through our experimental findings. Following this, our research progressed with a case study that involved a comparative evaluation of the performance of selected tools, aiming to identify and show the top performer. This allowed us to establish a comprehensive comparison of these tools for further analysis. This comparative analysis will be beneficial for testers in choosing the optimal mutation testing tool for C# applications. Future expansions of this study could include exploring additional tool characteristics and subject programs. Additionally, similar studies could be

conducted for different programming languages and their tools, broadening the scope and applicability of the research.

As a result, **RQ2** outlined in Section 1.4 is answered: “*How can existing mutation testing tools for C# be compared in terms of their features and effectiveness?*”

Available mutation testing tools for C# were reviewed and analyzed. A case study was conducted to determine the most suitable tool for the experiments conducted in this thesis which was determined as VisualMutator. The findings of this chapter have been synthesized into a conference article except the case study presented in Section 3.5 (Uzunbayir and Kurtel, 2019).



## CHAPTER 4: EVOCOLONY: A HYBRID APPROACH

This section aims to answer **RQ3** outlined in Section 1.4: “*How can the number of test cases in a test suite be reduced, and how can this process be enhanced using meta-heuristic methods in search-based mutation testing?*”

Test case reduction is one of the key problems in mutation testing, primarily due to its role in enhancing process efficiency and reducing computational demands. By simplifying the number of test cases, this process can cut down on resource usage, associated costs, and sharpens the focus on the most effective tests, leading to improved fault detection. Furthermore, it contributes to better test suite quality by eliminating redundant tests and ensures that mutation testing remains scalable. Search-based mutation testing uses meta-heuristic approaches to enhance the mutation testing process. These methods excel in rapidly exploring numerous potential solutions, especially useful in mutation testing due to the large search space created by numerous code mutations. However, one challenge lies in managing the test suite size, which can slow down the process. To address this, this chapter has focused on developing EvoColony, a novel approach combining genetic algorithms and ant colony optimization to reduce test cases in a test suite. This hybrid method leverages the strengths of both techniques: genetic algorithms provide an effective initial solution, while ant colony optimization refines it and avoids local optima, ensuring a balance between exploration and exploitation. The effectiveness of EvoColony is validated against traditional methods, including random search and variants of genetic algorithms, showing superior performance in achieving optimal solutions.

### ***4.1. Search-Based Mutation Testing***

Search-based mutation testing merges mutation testing principles with search-based optimization methods. This technique employs meta-heuristic algorithms such as genetic algorithms, ant colony optimization, or particle swarm optimization. Meta-heuristics are capable of solving complex problems with the aim of finding an optimal solution from a large search space (Silva et al., 2017). In the context of mutation

testing, these algorithms can be independently used to generate optimal mutants or reduce test cases of the test suite. Therefore, search-based mutation testing can increase testing efficiency by lowering time and computational demands for creating and maintaining a robust test suite, and automating test case generation to decrease manual intervention.

Figure 10 shows the categorization of meta-heuristics with examples. Different meta-heuristic algorithms can be applied in search-based mutation testing, each with specific limitations. Some of these can be listed as follows:

- Ant colony optimization is theoretically complex and has unpredictable convergence times.
- Genetic algorithms may be slow to converge to a solution, require long iterations, and can be computationally expensive.
- Particle swarm optimization risks getting trapped in local minimums, and its performance is highly parameter-sensitive.
- Simulated annealing can take a long time to converge in complex scenarios, and selecting an appropriate cooling schedule is crucial.
- Tabu search can be computationally demanding and needs a memory structure.

To overcome these drawbacks, a multipurpose strategy is often necessary. Hybrid models combining different algorithms' strengths are common, improving both reliability and efficiency. Parameter tuning is critical, with automated methods such as grid search or dynamic adaptation during the run aiding in finding optimal settings.

In this chapter, the aim is to use search-based mutation testing in test case reduction. The challenge of test case reduction in software testing is a critical issue that focuses on reducing the size of test suites while ensuring that they maintain their effectiveness and comprehensive coverage. This problem is particularly relevant because of the increasing complexity of software systems, leading to larger and more unwieldy test suites. As the number of test cases grows, so does the demand for time and resources for testing, making efficiency a key concern. In the following sections, we discuss



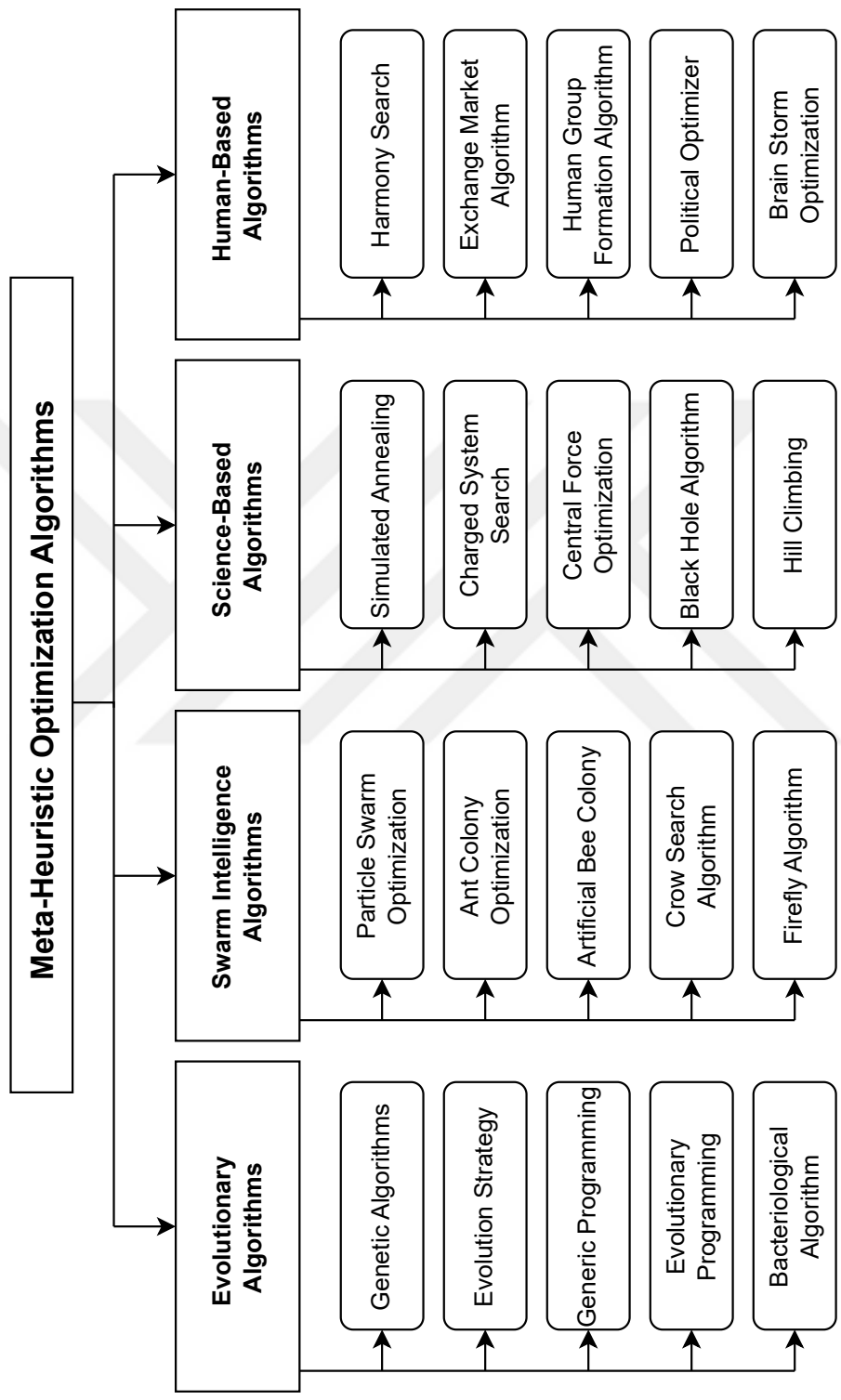


Figure 10. Meta-heuristic optimization algorithms categorization with examples.

test case reduction problem, genetic algorithms, and ant colony optimization in detail. Then, we propose and experiment with our proposed approach EvoColony which uses both meta-heuristics to reduce test cases in search-based mutation testing.

#### 4.2. Test Case Reduction Problem

A major aspect of test case reduction is identifying and eliminating redundant test cases (see Figure 11). These are tests that do not add extra value in terms of coverage or defect identification. Optimizing the test suite by eliminating such redundancies is essential, but it must be done without compromising the quality of the testing process. The goal is to generate a smaller set of test cases that are equal to its larger form, and still effective.

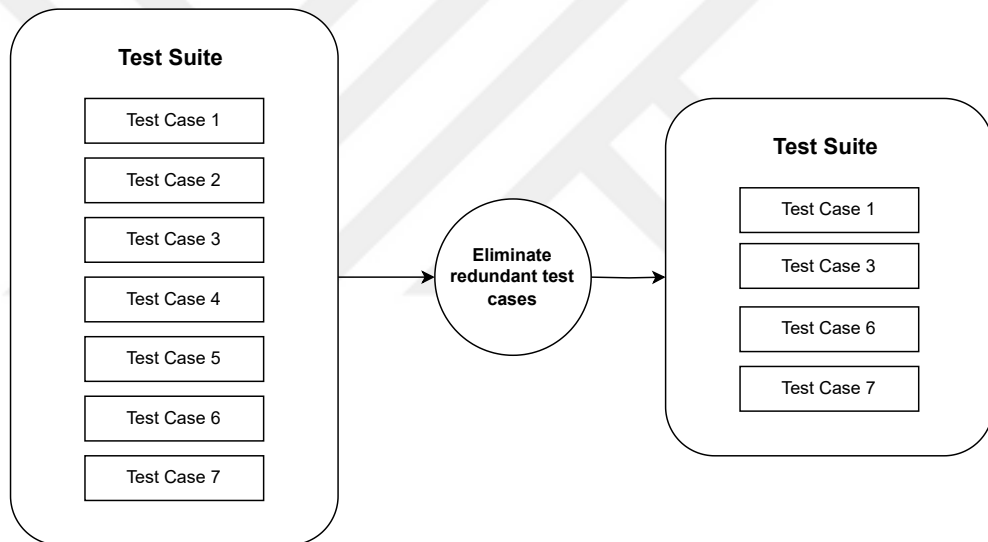


Figure 11. Test suite reduction process.

Balancing efficiency with effectiveness is at the heart of the test case reduction problem. The objective is to reduce the number of test cases without reducing their ability to detect defects. This balance is crucial as it ensures that the testing process remains thorough and becomes more efficient. Additionally, there are often limitations in terms of the cost and resources available for testing. With limited time and computational power, optimizing the testing process to make the most of these resources is a key consideration. Another layer of complexity is added by the dynamic nature of the software. Frequent changes and updates in software require continuous

revisions and optimizations of the test suite. This dynamic environment makes the test case reduction problem even more challenging, as the test suite must be regularly evaluated and adjusted to remain effective.

Haga and Suehiro (2012) introduced a method for test case reduction using a reduction matrix. This method is illustrated in Figures 12 and 13, which depict an example based on this study. Initially, the scenario involves 9 test cases and 6 mutants. Upon analysis, it is observed that test cases 1, 2, and 3 are redundant, since test case 4 alone is capable of eliminating all the mutants that these tests collectively address. Consequently, test cases 1, 2, and 3 are eliminated. Similarly, test cases 6, 7, and 8 are found to be also redundant because test case 5 can target all mutants that these tests could. As a result, these test cases are also removed. Ultimately, this approach effectively reduces the number of test cases from 9 to just 3, while still ensuring that all mutants are addressed during testing.

Ensuring that the reduced test suite still covers all critical aspects of the program, including both functional and nonfunctional requirements, is also vital. The ultimate aim is to utilize the testing process, making it more efficient and cost-effective while maintaining the reliability of the software. To achieve this, various meta-heuristics are employed. These strategies focus on finding the optimal balance between reducing the size of the test suite and maintaining robust coverage and defect detection capabilities.

In this study, we selected two meta-heuristics to further investigate the challenges of test case reduction. According to Jatana et al. (2017) genetic algorithms are the most frequently used technique for search-based mutation testing. These algorithms operate as they are exploring different parts of an unknown territory, ensuring that the important areas of the search space will not be missed. This approach helps finding a good, but not the best, solution. On the other hand, ant colony optimization uses a team of ants efficiently searching for food. This approach would effectively intensify the search around the promising areas identified by genetic algorithm. It fine-tunes and enhances the solutions, focusing on the most promising paths identified in the initial exploration phase. Therefore, we combined these two approaches.

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
M1	1	1	0	1	1	1	0	1	1
M2	0	0	0	0	0	0	0	0	1
M3	0	0	0	0	1	0	1	1	0
M4	0	0	0	0	1	1	0	0	0
M5	1	0	1	1	0	0	0	0	0
M6	0	1	0	1	1	0	0	1	0

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
M1	1	1	0	1	1	1	0	1	1
M2	0	0	0	0	0	0	0	0	1
M3	0	0	0	0	1	0	1	1	0
M4	0	0	0	0	1	1	0	0	0
M5	1	0	1	1	0	0	0	0	0
M6	0	1	0	1	1	0	0	1	0

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
M1	<del>1</del>	<del>1</del>	<del>0</del>	1	1	1	0	1	1
M2	<del>0</del>	<del>0</del>	<del>0</del>	0	0	0	0	0	1
M3	<del>0</del>	<del>0</del>	<del>0</del>	0	1	0	1	1	0
M4	<del>0</del>	<del>0</del>	<del>0</del>	0	1	1	0	0	0
M5	<del>1</del>	<del>0</del>	<del>1</del>	1	0	0	0	0	0
M6	<del>0</del>	<del>1</del>	<del>0</del>	1	1	0	0	1	0

Figure 12. Test case reduction using detection matrix.

	TC4	TC5	TC6	TC7	TC8	TC9
M1	1	1	1	0	1	1
M2	0	0	0	0	0	1
M3	0	1	0	1	1	0
M4	0	1	1	0	0	0
M5	1	0	0	0	0	0
M6	1	1	0	0	1	0

	TC4	TC5	TC6	TC7	TC8	TC9
M1	1	1	<del>1</del>	0	<del>1</del>	1
M2	0	0	0	0	0	1
M3	0	1	0	1	1	0
M4	0	1	1	0	0	0
M5	1	0	0	0	0	0
M6	1	1	<del>0</del>	0	<del>1</del>	0

	TC4	TC5	TC9
M1	1	1	1
M2	0	0	1
M3	0	1	0
M4	0	1	0
M5	1	0	0
M6	1	1	0

Figure 13. Test case reduction using detection matrix (cont'd).

### 4.3. Genetic Algorithms

Genetic algorithms are inspired by the natural selection process seen in biological systems, rooted in Darwin's evolutionary theory and Mendel's genetics work (Sharma et al., 2016). They serve as a problem solving method for complex search optimization challenges with numerous potential solutions. The task of identifying the optimal solution in this vast solution space is time-consuming. These algorithms use the principle of "*survival of the fittest*" to gradually discover the best solution (Uzunbayir, 2018).

In a traditional genetic algorithm, the process begins with a population of randomly generated entities, known as chromosomes, each representing a potential solution. The effectiveness of these individuals in solving the problem is measured using a fitness function. When parent individuals are selected, their offspring are created through crossover and mutation processes, aiming to produce superior solutions over successive generations. The algorithm continues until a predetermined stopping condition is met, ultimately converging towards the most optimal solution.

A distinctive aspect of a genetic algorithm is the method of translating the problem into chromosome representations and developing a fitness function tailored to the specific domain of the problem.

**Chromosome:** A chromosome represents an individual solution in the problem solving process. It symbolizes a distinct point in the broader solution space and is composed of a series of *genes*. Each gene corresponds to a specific attribute or parameter of the solution. The structure of the chromosome is designed to facilitate its alteration through various processes of the algorithm.

**Fitness Function:** The fitness function measures the adequacy of a chromosome in solving the problem by giving it a score based on its performance. Higher scores are indicative of more effective solutions. This function is crucial to directing the algorithm in deciding which solutions to keep and which to modify in search of the optimal solution.

A basic genetic algorithm operates through six main steps as depicted in Figure 14:

**1. Initial population:** The algorithm starts by creating a random collection

of chromosomes, forming the initial population. The population size varies according to the specific dataset and the problem domain.

- 2. Evaluation:** The fitness of each individual in the population is evaluated using a predefined fitness function determined by the user. This is performed for the initial population at the start and at the end of each iteration for the new population as well.
- 3. Selection:** Individuals are selected for the next generation based on their fitness levels. Those with higher fitness values are likely to be chosen, whereas those with lower fitness may be discarded. Various selection methods, such as roulette-wheel, rank, or tournament selection, can be used.
- 4. Crossover:** Offspring are produced by merging the genes of two parent individuals. This process aims to combine the best traits of the parents, potentially leading to better solutions. Several crossover techniques exist, including single-point, multi-point, uniform, or arithmetic crossover.
- 5. Mutation:** After crossover, mutation is applied to the offspring at a low probability. This step introduces small changes to an individual, fostering genetic diversity and enabling exploration of new solution spaces.
- 6. Terminate:** The algorithm concludes when it identifies the best solution. The termination condition is crucial to managing the computation time. Possible termination criteria include reaching a fitness threshold or producing a set number of generations.

Genetic algorithms bring a significant level of efficiency to the optimization of software testing processes. Their inherent robustness enhances test effectiveness and provides the flexibility to integrate new genetic or mutation operators specific to various testing scenarios. Additionally, they can handle the dynamic nature of software development, where requirements and environments frequently change, by easily adapting to these changes and evolving the test cases accordingly. This adaptability ensures that the testing process remains relevant and effective over time, even as the software evolves.

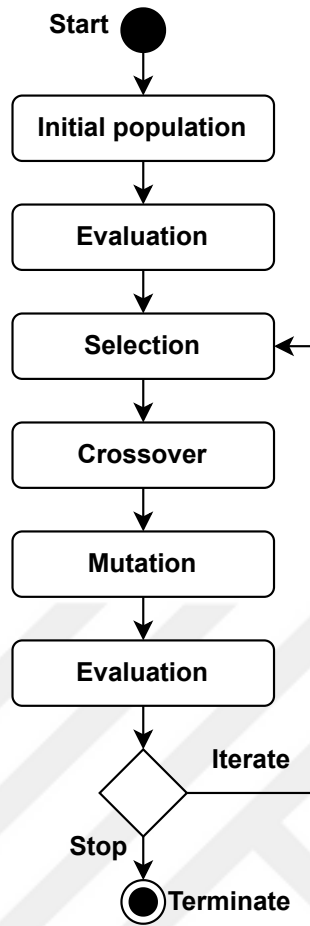


Figure 14. Traditional genetic algorithm steps.

#### 4.4. *Ant Colony Optimization*

Ant colony optimization technique, introduced by Marco Dorigo, draws inspiration from the foraging behavior of ants in nature (Dorigo et al., 2006). Ants, in their natural environment, have the ability to discover the shortest path between a food source and their nest through a self-organizing system. When ants explore an area containing food, they lay pheromone trails on their return to the nest. These pheromones are detectable by other ants, which tend to follow paths with higher concentrations of pheromones. Over time, as pheromones evaporate on less-used paths, the most efficient route, marked by the strongest pheromone trail, becomes the preferred path for the colony. This route typically represents the shortest route from the nest to the food source. The reliance on the strength of the pheromone to determine the path choice makes this a probabilistic method (Uzunbayir, 2022).



In the ant colony algorithm, the problem to be solved is structured as a graph. This graph consists of nodes, each representing a possible state or decision point, and edges, which signify the transitions between these states. The structure of the graph, directed or undirected, depends on the specific problem being addressed. This graphical representation is crucial for the algorithm's operation as it mimics the way ants navigate and make decisions in their environment.

A basic ant colony algorithm operates in six steps (see Figure 15):

**1. Parameter initialization:** Initially, the pheromone levels on the paths in the graph are set to zero, indicating an absence of pheromones. Essential parameters for the algorithm's behavior include:

- **Number of ants:** Determines the quantity of artificial ants navigating the graph to create solutions. More ants can lead to better solutions but require more computational resources and time.
- **Pheromone evaporation rate ( $\rho$ ):** Ranges from 0 to 1 and indicates how quickly pheromone trails dissipate. A higher rate allows faster adaptation, but may result in instability.
- **Pheromone intensity ( $Q$ ):** Sets initial pheromone levels. Balanced initial concentrations are crucial for algorithm efficiency.
- **Pheromone influence factor ( $\alpha$ ):** Dictates the importance of pheromone trails in ants' route decisions.
- **Heuristic information ( $\beta$ ):** An optional parameter that influences the significance of the available heuristic guidance for the ants.
- **Termination criteria:** Defines the total number of runs of the algorithm or other conditions to stop the algorithm.

**2. Path construction:** Initially, ants randomly traverse the graph, constructing solutions. In subsequent iterations, solutions are refined on the basis of probabilistic decision-making influenced by pheromone levels and optional heuristic information.

- 3. Solution evaluation:** Each ant's path is assessed using a problem-specific objective function, such as the path length for shortest path problems.
- 4. Pheromone update:** Pheromone levels increase based on the quality of the solution, with better solutions leading to higher deposits of pheromones. Some variants allow only the best or a group of elite ants to update pheromone levels.
- 5. Pheromone evaporation:** To mimic natural evaporation, unused pheromone trails are reduced, typically by multiplying the current level by  $(1 - \rho)$ .
- 6. Terminate:** The algorithm concludes once a predetermined stopping criterion is met, such as a certain number of iterations, an objective function threshold, or a custom condition.

Ant colony optimization has several benefits in tackling optimization challenges. In mutation testing, where numerous mutants are involved, it can efficiently navigate this large space, focusing on mutants more likely to reveal faults and thus decreasing the required number of test cases.

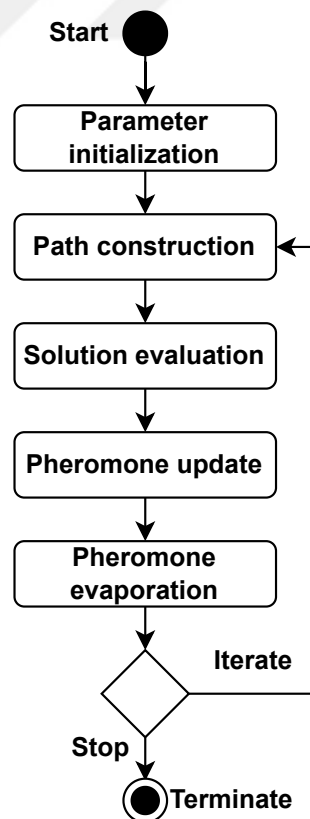


Figure 15. Traditional ant colony algorithm steps.

## **4.5. Methodology**

This section outlines the research questions and describes the proposed method called EvoColony. The method is explained, the stages are described and illustrated, and the pseudocode of the algorithm is presented.

### **4.5.1. Research Questions**

The primary goal of this chapter is to present and assess the efficacy of the proposed hybrid approach, named EvoColony, within the scope of search-mutation testing. To achieve a comprehensive evaluation, we have divided **RQ3**, initially stated at the start of this chapter, into two sub-research questions:

- **RQ3a:** Can the EvoColony algorithm effectively minimize the number of test cases in a test suite while maintaining a high mutation adequacy score?
- **RQ3b:** Compared to conventional search-based methods, including random search, genetic algorithms, and ant colony optimization, how does EvoColony perform?

The primary objective behind answering these questions is to provide an understanding of the advantages of integrating meta-heuristic algorithms into search-based mutation testing, particularly to reduce the number of test cases. Secondary objective is to assess whether the proposed approach is able to perform the task and does it make it better than the others or not.

### **4.5.2. EvoColony: A Hybrid Approach to Search-Based Mutation Testing**

EvoColony is a technique that combines genetic algorithm with ant colony optimization to leverage the strengths of both methods. This combination addresses the limitations inherent in genetic algorithms, utilizing the capabilities of ant colony optimization to mitigate these drawbacks. For example, genetic algorithms often risk converging to local optima instead of exploring global optima. On the contrary, optimization of the ant colony shows better adaptability to changes in the problem

environment, helping to avoid early convergence to local optima. Furthermore, tuning parameters in genetic algorithms, such as the crossover rate, mutation rate, and selection strategy, can be more complex and labor intensive compared to relatively simpler and more straightforward parameter tuning in ant colony optimization.

By integrating these two approaches, ant colony optimization can improve the performance of the genetic algorithm. It achieves this by using the genetic algorithm parameters to guide the initial distribution of pheromones during the graph initialization phase, thus creating a more efficient hybrid approach.

EvoColony can be segmented into four distinct stages: input preparation, genetic algorithm part, ant colony optimization part, and output, as shown in Figure 16:

- 1. Input Preparation:** The initial stage involves preparing the necessary inputs for EvoColony. These include the test program, its corresponding test suite, and the set of mutant programs. Algorithm 3 is used to start the process.
- 2. Genetic Algorithm:** The second stage initiates the genetic algorithm. This involves encoding test cases from the test suite as chromosomes, establishing a fitness function according to a specific equation 1, and setting the parameters for the genetic algorithm. The fitness function is defined as follows:

$$f(x) = M(x) + (1 - S(x)) \quad (1)$$

Here,  $f(x)$  denotes the fitness of the test suite  $x$ ,  $M(x)$  represents the mutation adequacy score for the test suite  $x$ , and  $S(x)$  is the normalized size of the test suite  $x$ . The fitness of the initial population is evaluated, followed by the selection of parents for the next generation using roulette-wheel selection.

The likelihood of chromosome selection is proportional to the values of the fitness function. The mating process then occurs, involving a single-point crossover and mutation at certain probabilities, leading to the creation of a new solution. This cycle repeats until the optimal solution is identified. Algorithm 1 presents the details of this stage for initial test case reduction in the EvoColony process.

The time complexity of this part can be approximated as follows:

- (a) The initialization of the population in has a time complexity of  $O(n)$ , as it involves creating and initializing each individual in the population.
- (b) The calculation of fitness in has a time complexity of  $O(T \times M)$ , as it involves running the test program on each individual in the population and calculating their fitness based on the number of test cases and mutants.
- (c) The while loop will run until the stopping condition is true. The number of iterations in the while loop depends on the stopping condition and the number of individuals in the population. Therefore, the time complexity of the while loop is  $O(n \times S)$ .
- (d) The discarding of the least fit tests in has a time complexity of  $O(S)$ , as it involves finding and removing the least fit individuals from the population.
- (e) The addition of the offspring in has a time complexity of  $O(1)$ , as it involves adding the offspring to the population.

Hence, overall time complexity can be approximated as:

$$O(n \times S \times T \times M)$$

where

- $n$  represents the number of iterations,
- $S$  is the population size,
- $T$  stands for the test cases,
- $M$  is the number of mutants.

**3. Ant Colony Optimization Algorithm:** In the third stage, the ant colony optimization takes place. Utilizing the best solution from the genetic algorithm as a starting point, the parameters for the ant colony are set, the ant colony graph is constructed, and initial pheromones are distributed on paths based on the genetic algorithm's solution. Ants navigate these paths, updating pheromones

along the trails. Pheromone evaporation occurs on paths less traveled. The solution is continually assessed and refined through repeated iterations until the most effective solution is determined. Algorithm 2 presents the details of this stage for the solution refinement.

The time complexity of ant colony optimization part can be estimated as follows:

- (a) Constructing the graph has a time complexity of  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.
- (b) Initializing pheromones has a time complexity of  $O(V^2)$ , as it involves initializing a matrix of size  $V \times V$ .
- (c) Constructing the path for each ant has a time complexity of  $O(V)$ , as each ant visits all vertices once.
- (d) Calculating the mutation score has a time complexity of  $O(M)$ , where  $M$  is the number of mutants.
- (e) Updating pheromones has a time complexity of  $O(V^2)$ , as it involves updating the pheromone matrix for each edge in the path.
- (f) Finding the best path has a time complexity of  $O(V)$ , as it involves iterating over all vertices to find the maximum score.

Hence, the overall time complexity can be approximated as:

$$O(n \times (a \times (V + M) + V^2))$$

where

- $n$  is the number of iterations,
- $a$  is the number of ants,
- $V$  is the number of vertices,
- $M$  is the number of mutants.

**4. Output:** The final stage involves presenting the best solution. The expected result is a reduced test set that maintains an optimal mutation score.

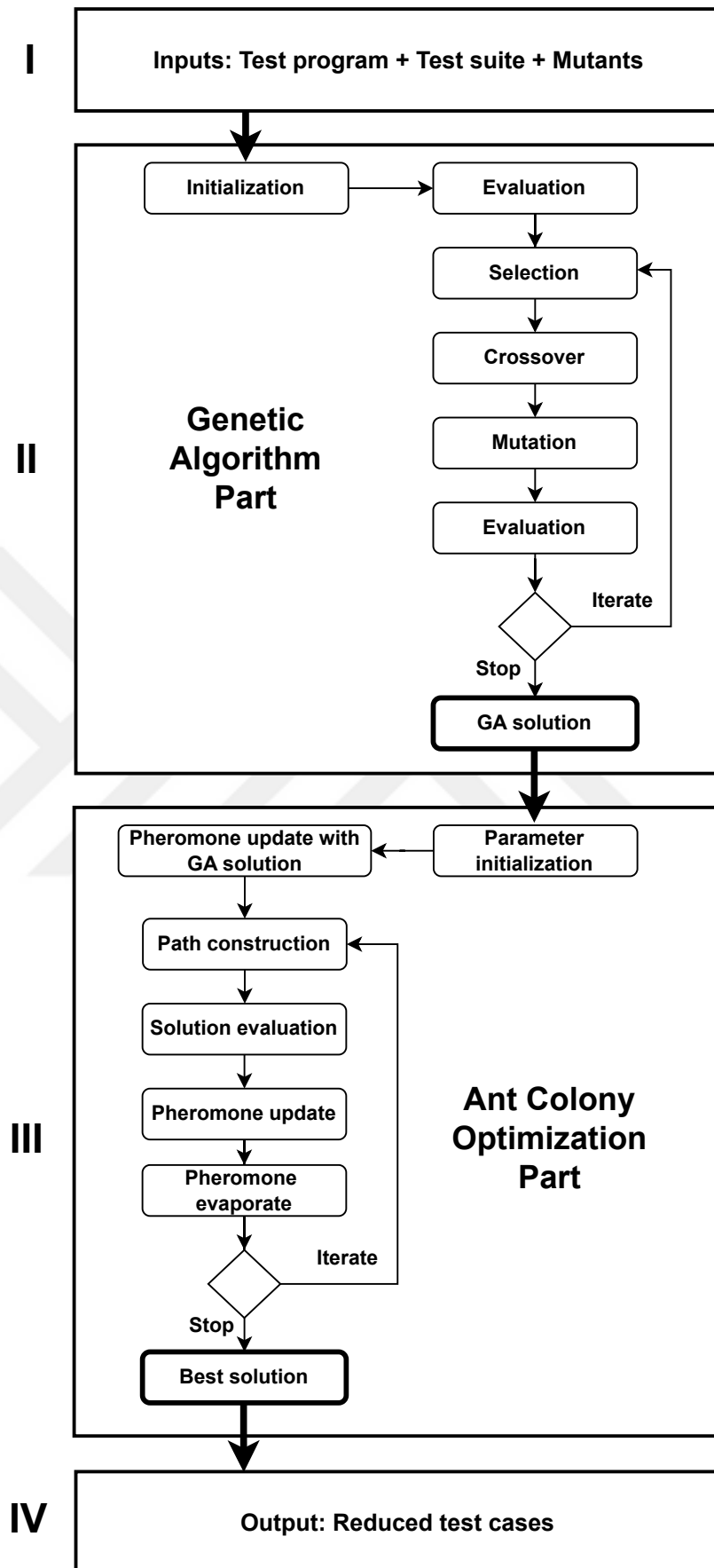


Figure 16. The EvoColony algorithm steps.

---

**Algorithm 1** EvoColonyGA for Initial Reduction

---

**Require:** *testProgram, testCases, size, mutants, cPoint, mRatio*

```
1:  $i = 0$ 
2:  $GASolution \leftarrow \emptyset$ 
3:  $population[i] \leftarrow initializePopulation(testCases, size)$ 
4:  $population[i].calculateFitness(testProgram, mutants)$ 
5: while  $stoppingCondition \neq true$  do
6:    $parents \leftarrow population[i].select()$ 
7:    $offspring \leftarrow parents.crossover(cPoint)$ 
8:    $offspring \leftarrow parents.mutation(mRatio)$ 
9:    $population[i].calculateFitness(testProgram, mutants)$ 
10:   $population.discardLeastFitTests()$ 
11:   $population.add(offspring)$ 
12:   $i \leftarrow i + 1$ 
13: end while
14:  $GASolution.add(population)$ 
15: return  $GASolution$ 
```

---

---

**Algorithm 2** EvoColonyACO for Refinement

---

**Require:** *GASolution, testProgram, testCases, ants,  $\tau_{init}$ ,  $\alpha$ ,  $\beta$*

```
1:  $i = 0$ 
2:  $ACOSolution \leftarrow \emptyset$ 
3:  $graph \leftarrow constructGraph(testCases)$ 
4:  $initializePheromones(graph, GASolution, \tau_{init})$ 
5: while  $stoppingCondition \neq true$  do
6:   for  $ant$  in  $ants$  do
7:      $path \leftarrow ant.constructPath(graph, \alpha, \beta)$ 
8:      $score \leftarrow calculateMutationScore(path, testProgram, mutants)$ 
9:      $updatePheromones(graph, path, score)$ 
10:  end for
11:   $i \leftarrow i + 1$ 
12: end while
13:  $bestPath \leftarrow findBestPath(graph)$ 
14:  $ACOSolution.add(bestPath)$ 
15: return  $ACOSolution$ 
```

---

---

**Algorithm 3** EvoColonyMain

---

```
1:  $size, cPoint, mRatio \leftarrow prepareInputsGA()$ 
2:  $ants, \tau_{init}, \alpha, \beta \leftarrow prepareInputsACO()$ 
3:  $GASolution \leftarrow EvoColonyGA(program, tests, size, mutants, cPoint, mRatio)$ 
4:  $ACOSolution \leftarrow EvoColonyACO(GASolution, program, tests, ants, \tau_{init}, \alpha, \beta)$ 
5:  $Print(ACOSolution)$ 
```

---



## 4.6. *Experimental Design*

This section details the testing environment, describes the test data, and discusses the benchmark algorithms used in this chapter.

### 4.6.1. *Test Environment*

The experiments were carried out using ten different programs, all written in C#. Some of these programs were sourced from the existing software testing literature (Wedyan et al., 2022; Bashir and Nadeem, 2017), while others were chosen from common mutation testing experiments. Programs not originally in C# were converted to this language. The selected programs offer a variety of features, including mathematical calculations, array manipulations, and intricate branching conditions, thus ensuring a diverse test set. Each program varies in size. For each of these programs, initial test suites were generated using IntelliTest and subsequently manually refined. This refinement process ensured that the tested programs were error-free and that the experimental results were not skewed by any run-time exceptions. Mutants for each program were created using VisualMutator selected based on the results of Chapter 3, as illustrated in the referenced Figure 17. For the purpose of this study, five standard mutation operators supported by VisualMutator were chosen. This selection was based on the observation that the mutation adequacy score remained largely unchanged even when additional operators were included. The specific operators selected for this research are detailed as follows:

- **Arithmetic Operator Replacement:** This involves replacing one arithmetic operator with another, such as (+, -, \*, /).
- **Relational Operator Replacement:** This changes the relational operators such as (<, <=, >, >=, ==, !=).
- **Logical Operator Replacement:** This substitutes one logical operator for another (&, |, ^).
- **Logical Connector Replacement:** This replaces logical connectors (&&, ||).

- **Operator Deletion:** This generates two mutants for each operation, including operators such as (+, -, >, <=, %).

In addition, the study incorporates four object-oriented mutation operators to account for object-oriented programming features:

- **Accessor Method Change:** This modifies the accessor methods for a class's properties or fields.
- **Accessor Modifier Change:** This changes the accessibility level of a class's property or field accessor.
- **Member Variable Initialization Deletion:** This operator removes the initialization of member variables.
- **Member Call from Another Inherited Class:** This mutates a method call within a class (or its base class) to a call to a member of another class that has the same base class.

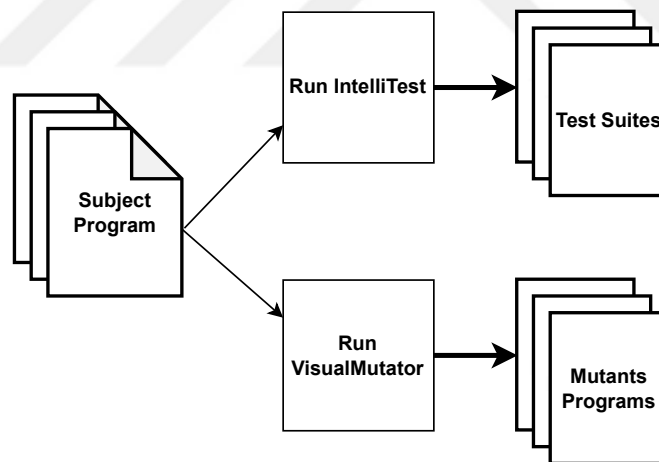


Figure 17. Test data setup.

For comparative analysis, five different algorithms were implemented: random search, two variants of the traditional genetic algorithm with single-point and double-point crossovers, a traditional ant colony algorithm, and the newly proposed Evo-Colony algorithm. These algorithms were applied to run each of the subject programs. The experiments were carried out on a desktop computer equipped with Windows 11 Pro OS and an Intel i7 9700k 2.8 GHz processor.

Table 20. Subject programs details.

<b>Subject Program</b>	<b>Size in LOC</b>	<b>Number of Mutants</b>
BubbleSort	93	85
Calendar	115	114
TriangleType	123	149
ArrayOperations	113	176
TemperatureConverter	104	95
QuadraticSolver	73	88
HashTable	107	151
BinarySearch	64	164
BankAccount	76	110
AutoDoor	169	195
<b>Total</b>	<b>1037</b>	<b>1227</b>

#### **4.6.2. Test Data**

The specifics of the programs used as subjects in this study are outlined in Table 20. This includes information on each program's size in terms of lines of code (LOC) and the total count of mutants. Across all programs, the cumulative code size amounts to 1037 LOC, and the total number of mutants is 1227. For all these programs, the initial mutation adequacy scores have been calculated to be 1.000.

#### **4.6.3. Benchmark Algorithms**

The experiments utilize the following selected benchmark algorithms:

- **Random Search (RS):** This method involves iteratively selecting test case subsets at random, assessing their mutation adequacy scores, and retaining those within a satisfactory score range. This process is repeated until the target mutation adequacy score is achieved.
- **Traditional Genetic Algorithm with Single-Point Crossover (GA-SP):** This approach starts with an initial population of random test case subsets. A single-point crossover and mutation are then applied to generate new generations. The subsets are evaluated using a fitness function based on mutation adequacy scores. The algorithm progressively refines the population, aiming to optimize fitness and decrease test cases until a predefined maximum generation is reached.

- **Traditional Genetic Algorithm with Double-Point Crossover (GA-DP):** This genetic algorithm variant uses two crossover points within the test case sequence, exchanging portions of the parent sequences to create offspring. The fitness of each offspring, determined by the mutation adequacy score, is evaluated. The double-point crossover is designed to introduce more diversity into the test case subsets, potentially leading to a more efficient reduced set. This algorithm continuously applies double-point crossover and mutation to evolve the population towards an optimal fitness level.
- **Traditional Ant Colony Optimization Algorithm (ACO):** In this algorithm, each test case is represented as a node on a graph. The algorithm involves choosing a set of test cases by forming a path through this graph. Pheromones are initially distributed evenly across all paths. Artificial ants construct solutions by following a probabilistic function influenced by pheromone levels. Upon completing a path, each ant assesses the fitness of its selected subset based on mutation adequacy scores. Pheromones are then updated accordingly, reinforcing paths leading to higher fitness scores and reducing those that don't. The process iterates until a set number of iterations are completed.
- **EvoColony:** This newly proposed algorithm is executed and its performance is compared with the other algorithms to determine the effectiveness of the test suite reduction.

#### 4.6.4. Results and Evaluation

This section details the experimental outcomes and their evaluation. The experiments were structured around the research questions outlined earlier in Section 4.5.1.

*RQ3a: Can EvoColony algorithm effectively minimize the number of test cases in a test suite while maintaining a high mutation adequacy score?*

The goal of this RQ is to determine if EvoColony can decrease the number of test cases without compromising the test suite's ability to effectively detect defects.

The parameters used for the genetic and ant colony optimization components of EvoColony are displayed in Table 21. These parameters were chosen based on a blend

of experimental results, empirical data, and considerations of computational efficiency.

Table 21. Genetic and ant colony parameters of EvoColony.

<b>Genetic Algorithm Specific Parameters</b>	
Selection type:	Roulette-wheel selection
Crossover probability:	0.8
Mutation probability:	0.06
Population size:	Twice the number of initial test cases
Chromosome size:	Number of test cases
Maximum iteration:	200 generations
<b>Ant Colony Algorithm Specific Parameters</b>	
$\alpha$ :	2
$\beta$ :	2
$\rho$ :	0.02
Q:	Based on the genetic algorithm results
Number of ants:	100
Maximum iteration:	1000

Table 22 provides a detailed overview of the EvoColony test outcomes across various subject programs. This includes comparative data such as the initial number of test cases, the size of the reduced test suite, and the test reduction ratio. The mutation adequacy scores for all experiments remained at an optimum level, even after the test case reduction. The reduction ratio was computed using the following formula (2):

$$Reduction\ Ratio = \frac{Initial\ \#\ of\ test\ cases - Reduced\ \#\ of\ test\ cases}{Initial\ \#\ of\ test\ cases} \times 100 \quad (2)$$

One of the standout observations is the significant reduction in the number of test cases across the board. For instance, BubbleSort saw its test cases drop by a substantial 50.48%, while maintaining a perfect mutation adequacy score. This implies that not only were the test cases reduced, but they were optimized so that they were still able to cover the necessary conditions and branches for effective mutation testing. However, it is important to note the variation in reduction percentages. TriangleType had the smallest reduction at 29.37%, suggesting room for improvement. Despite these reductions, the mutation adequacy scores remain perfect for all programs. This high level of mutation adequacy indicates that the sets of test cases, although reduced, are nevertheless of high quality, and effective in identifying potential faults.

The aggregate statistics presented at the end of the table provide a broader

Table 22. EvoColony results.

<b>Subject Program</b>	<b>Initial # of Test Cases</b>	<b>Reduced Test Suite Size</b>	<b>Reduction Ratio</b>
BubbleSort	105	52	50.48%
Calendar	130	70	46.15%
TriangleType	126	89	29.37%
ArrayOperations	149	81	45.64%
TemperatureConverter	115	63	45.22%
QuadraticSolver	108	72	33.33%
HashTable	119	65	45.38%
BinarySearch	123	69	43.90%
BankAccount	152	94	38.16%
AutoDoor	165	102	38.18%
<b>Total</b>	<b>1192</b>	<b>757</b>	<b>36.49%</b>

perspective, indicating that from an initial count of 1192 test cases, the number was effectively reduced to 757, marking a noteworthy overall reduction of 36.49%. Importantly, despite this reduction in test cases, the mutation adequacy score for each of the subject programs consistently remained at a perfect 1.000. This underscores the fact that reducing the number of test cases does not equate to a decline in testing adequacy, confirming that only non-essential test cases were eliminated. These experimental findings clearly demonstrate EvoColony’s capability to considerably reduce the number of test cases while still preserving the integrity and quality of the testing process. Therefore, the answer to RQ3a is yes.

*RQ3b: In comparison to conventional search-based methods, including random search, genetic algorithms, and ant colony optimization, how does EvoColony perform?*

The objective of this RQ is to evaluate the effectiveness of the EvoColony algorithm relative to other prevalent methods. The aim is to ascertain if EvoColony outperforms, underperforms, or matches other methods in improving test suite quality.

Based on the in results in Table 23, it is apparent that the RS method demonstrates the least significant reductions and is comparatively less effective than other methodologies. Subpar performance of RS can primarily be related to its limited optimization capabilities. Unlike more sophisticated algorithms such as GA-SP, GA-DP and ACO, RS lacks an iterative optimization mechanism.

Table 23. Comparative test results.

Subject Program	Initial # of Test Cases	RS		GA-SP		GA-DP		ACO		EvoColony	
		Reduced Test Suite Size	Reduction Ratio	Reduced Test Suite Size	Reduction Ratio	Reduced Test Suite Size	Reduction Ratio	Reduced Test Suite Size	Reduction Ratio	Reduced Test Suite Size	Reduction Ratio
BubbleSort	105	84	20.00%	68	35.24%	65	38.10%	60	42.86%	52	50.48%
Calendar	130	96	26.15%	83	36.15%	83	36.15%	77	40.77%	70	46.15%
TriangleType	126	109	13.49%	100	20.63%	97	22.99%	95	24.60%	89	29.37%
ArrayOperations	149	105	29.53%	98	34.23%	90	39.60%	88	40.94%	81	45.64%
TemperatureConverter	115	90	21.74%	81	29.57%	81	29.57%	75	34.78%	63	45.22%
QuadraticSolver	108	81	25.00%	78	27.78%	77	28.70%	75	30.56%	72	33.33%
HashTable	119	92	22.69%	76	36.13%	72	39.50%	70	41.18%	65	45.38%
BinarySearch	123	92	25.20%	85	30.89%	83	32.52%	79	35.77%	69	43.90%
BankAccount	152	131	13.82%	120	21.05%	116	23.68%	105	30.92%	94	38.16%
AutoDoor	165	135	18.18%	118	28.48%	115	30.30%	112	32.12%	102	38.18%

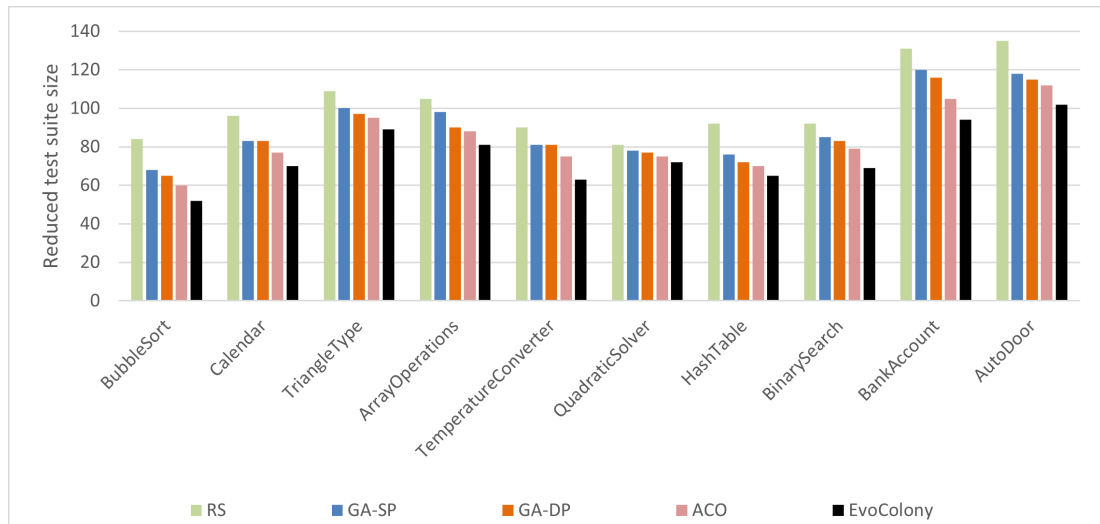


Figure 18. Reduced rest suite size comparison.

Moreover, RS does not adapt or improve its methodology based on previous results, instead relying solely on a stochastic or random selection process.

GA variants demonstrate moderate effectiveness in reducing test cases, outperforming RS and often delivering comparable outcomes. Their performance is notably stable across different subject programs, indicating their robustness and reliability for test case reduction. The two GA variants show similar results, with only minor variations in their reduction percentages, which seem to be influenced by the characteristics of the specific program being tested. For instance, both GA-SP and GA-DP achieved identical reduction rates for programs such as Calendar and TemperatureConverter, at 36.15% and 29.57% respectively, but their effectiveness varied for other programs. Moreover, the GA variants not only surpass RS in performance but also compete well with ACO and EvoColony.

While GA variants are reliable and robust, ACO tends to provide superior test case reduction in certain programs. For example, ACO achieved significant reductions in programs such as BubbleSort and Calendar, with reduction rates of 42.86% and 40.77%, respectively. This positions ACO as the second-most effective method according to these results.

EvoColony stands out as the most advantageous approach for test case reduction among the four methods evaluated. It consistently outperforms the others in terms of minimizing the number of test cases for all the subject programs. Additionally,



Figure 18 comparing the test suite reductions, in relation to the data in the comparative results in Table 23, further highlights EvoColony’s effectiveness. The findings show that EvoColony, on average, manages to reduce about one-third of all test cases. This level of efficiency establishes it as the top-performing algorithm in the experiments.

To evaluate the time efficiency of five different test case reduction methods, notable variations in execution times were observed. Figures 19 to 28 present the execution time results in minutes for each subject program individually. In all experiments, on the average, RS performed faster than the others followed by EvoColony, ACO, GA-SP, and then GA-DP.

RS, with its straightforward methodology, recorded the quickest iteration times, averaging approximately 8.73 minutes. GA-SP and GA-DP demonstrated more structured search techniques and logged average execution times of 13.13 and 14.06 minutes, respectively. These times reflect the additional duration needed for their crossover processes. ACO, renowned for its complex pheromone trail mechanics and probabilistic decision-making, required an average time of 11.95 minutes, which is lower than the genetic algorithm variants. The proposed EvoColony method, although faster than the GA variants and ACO, was slower than RS, achieving an average execution time of 11.23 minutes. This relatively efficient performance is attributed to its strategy of combining GA for broad initial searches with ACO for targeted local optimizations, thus enabling quicker convergence towards effective solutions.

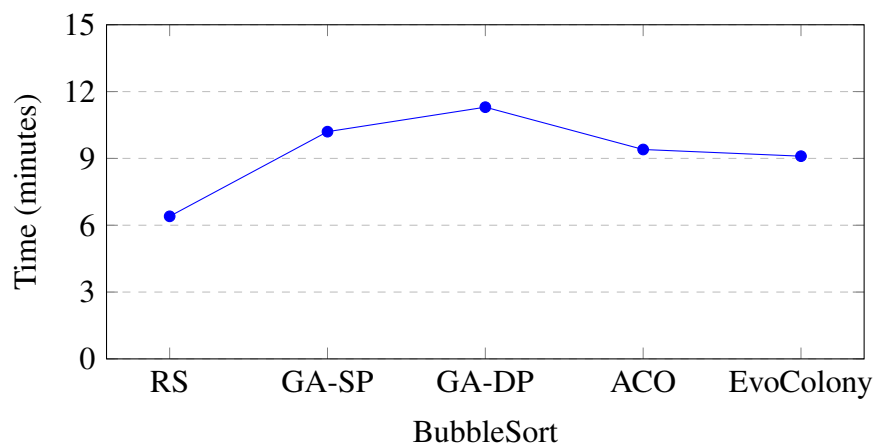


Figure 19. Run-time performance comparison for BubbleSort.

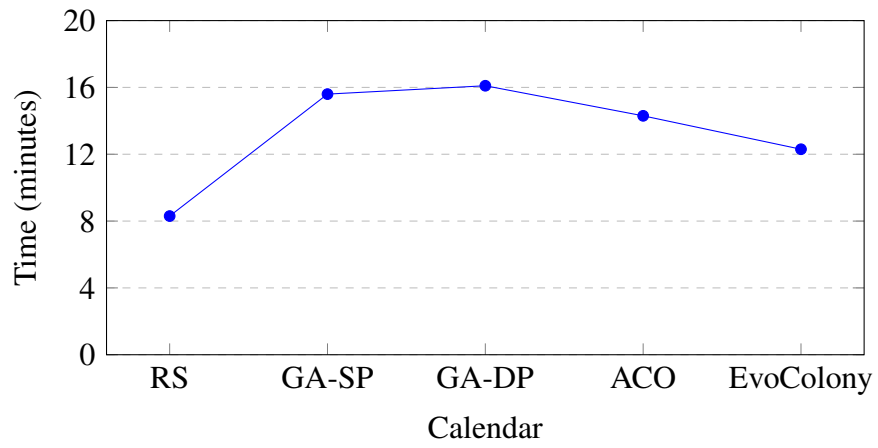


Figure 20. Run-time performance comparison for Calendar.

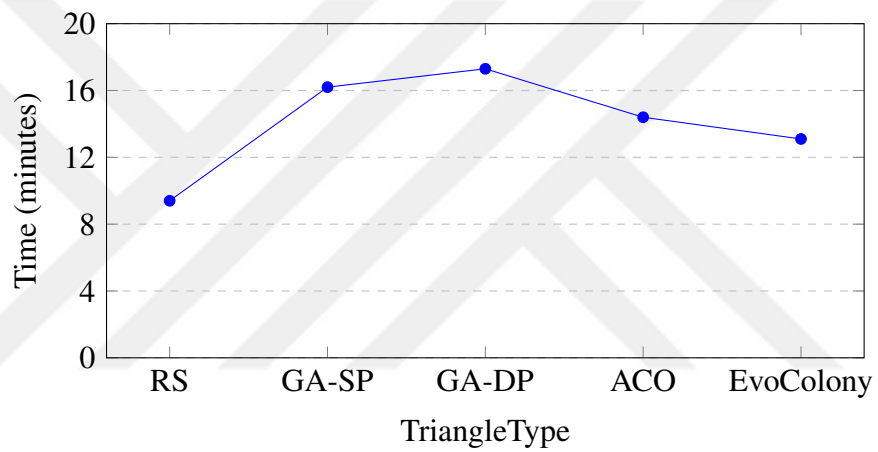


Figure 21. Run-time performance comparison for TriangleType.

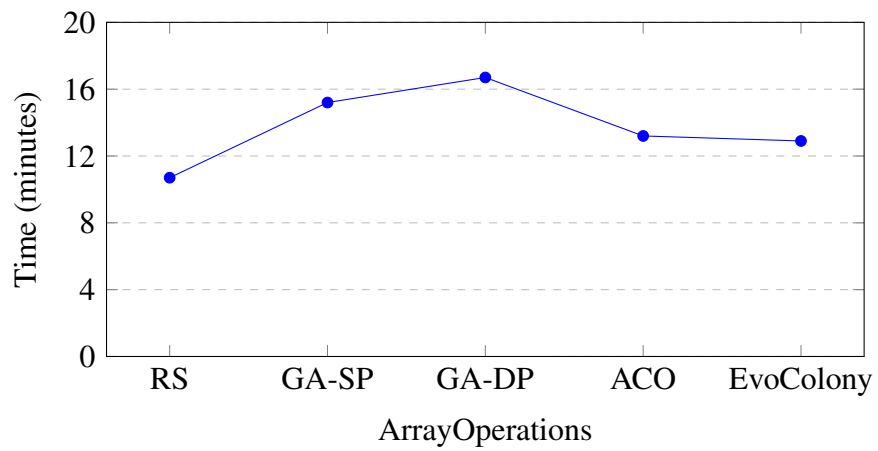


Figure 22. Run-time performance comparison for ArrayOperations.

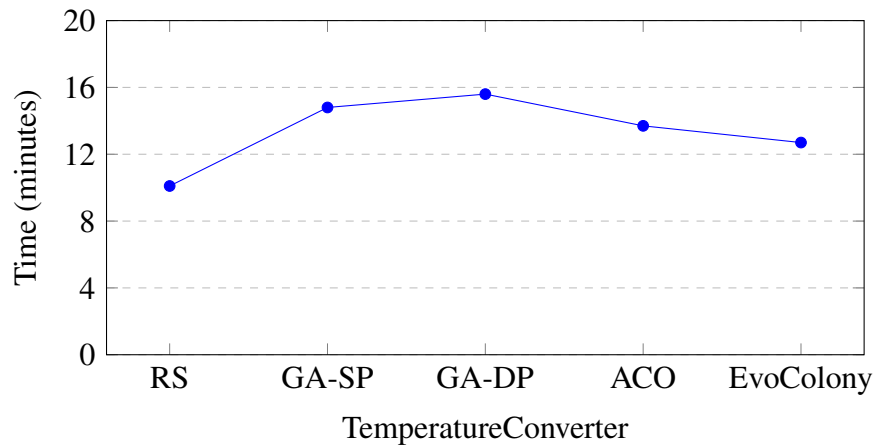


Figure 23. Run-time performance comparison for TemperatureConverter.

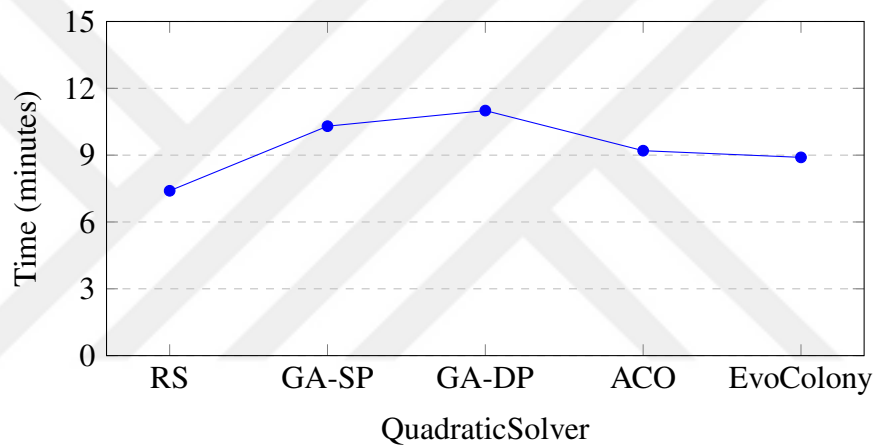


Figure 24. Run-time performance comparison for QuadraticSolver.

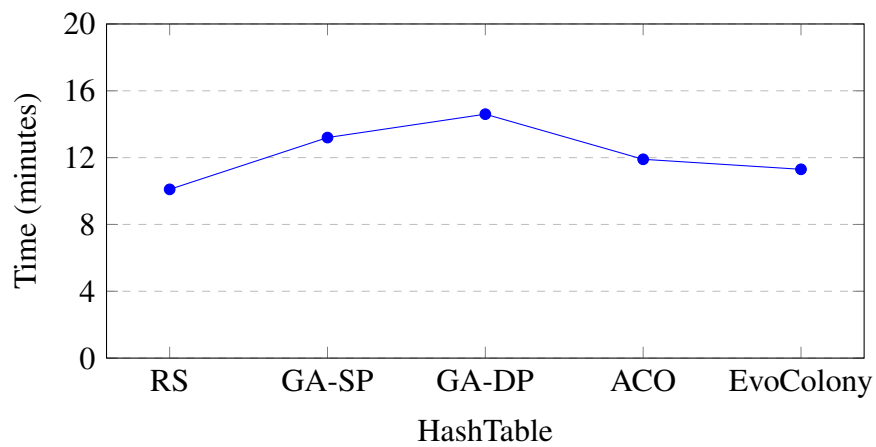


Figure 25. Run-time performance comparison for HashTable.

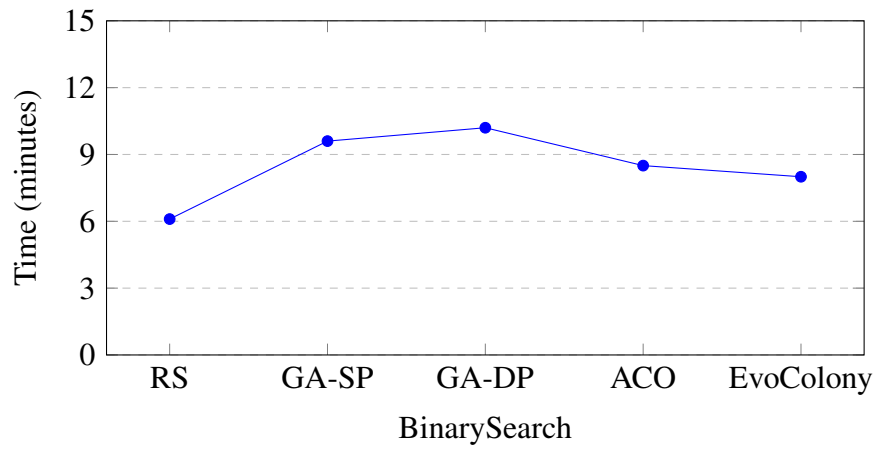


Figure 26. Run-time performance comparison for BinarySearch.

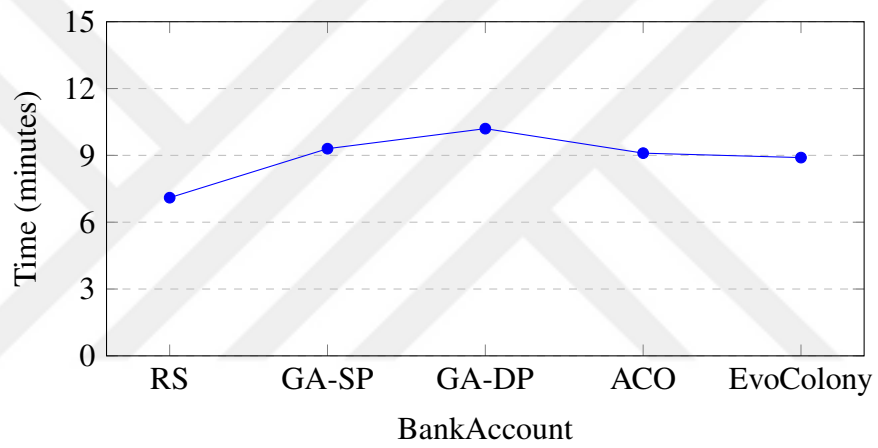


Figure 27. Run-time performance comparison for BankAccount.

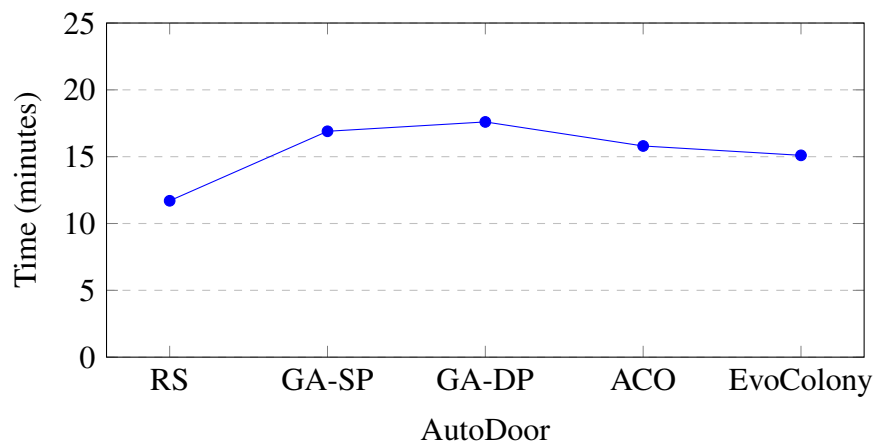


Figure 28. Run-time performance comparison for AutoDoor.

#### 4.7. Conclusion

The proposed hybrid approach excels in reducing test cases significantly more than the other compared methods while preserving the quality of the test suites. This success highlights the benefits of utilizing complementary algorithms. EvoColony's dual optimization strategy is particularly advantageous, genetic algorithms for initial search and ant colony optimization for subsequent refinement. This approach effectively harnesses the strengths of both techniques while mitigating their individual limitations.

Future research could explore enhancing EvoColony to handle more complex mutation types or combining it with additional machine learning techniques for a more adaptive testing approach. Comparisons with other search-based strategies, such as particle swarm optimization, tabu search, simulated annealing, or other hybrid methods, could also provide valuable insights.

As a result, **RQ3** outlined in Section 1.4 is answered: *“How can the number of test cases in a test suite be reduced, and how can this process be enhanced using meta-heuristic methods in search-based mutation testing?”* To answer this, EvoColony was introduced, evaluated through a comparative analysis against four traditional approaches, and showed better results. The findings of this chapter have been synthesized into a research article (Uzunbayir and Kurtel, 2023a).

## CHAPTER 5: LEVERAGING MUTANTS IN HIGHER-ORDER

This chapter aims to answer **RQ4** outlined in Section 1.4: “*How does the implementation of genetic algorithms as a search strategy in higher-order mutation testing impact the efficiency of generating high-quality mutants, particularly in reducing the production of equivalent high-order mutants?*”

Higher-order mutation testing involves modifying source code with multiple mutation operators to create varied versions of the program. This method aims to enhance the testing process, specifically its design and implementation stages, by enabling automated evaluation of test cases. One of the main challenges in higher-order mutation testing is the need to generate a substantial number of mutants and navigate a complex mutation search space. To tackle this, the issue is modeled as a search problem similar to the problem explained in Chapter 4. This chapter introduces another search strategy based on genetic algorithms for mutation testing. The goal is to lower the production of equivalent high order mutants, thereby achieving an adequate mutation score with a smaller set of mutants. Experiments were conducted to compare this approach with a random search method and four distinct genetic algorithm variants, each employing a different selection technique: roulette wheel, tournament, rank, and truncation selection.

### 5.1. *First-Order and Equivalent Mutants*

Mutation testing primarily involves duplicating the original program and introducing artificial faults using mutation operators to create a mutant. This mutant is then subjected to a test suite to identify these introduced errors. The concept is explored in detail in Chapter 2. For clarity and as a refresher, we will provide an explanation here once again. A typical example of this process is illustrated in Table 24, where the original code is cloned and altered by switching the “greater than (>)” operator with the “less than (<)” operator.

A mutant generated by applying a single mutation operator to the original program is referred as a **first-order mutant**. These mutants are considered as traditional and

simple mutants that include only one fault. This is exemplified in Table 24, where a first-order mutant is displayed.

Table 24. An original program and its first-order mutant.

<b>Original Program</b>	<b>First-Order Mutant</b>
<pre> READ a number   if (number &gt; 100)     PRINT "I AM STRONG!"   else     PRINT "I AM WEAK!"   end if </pre>	<pre> READ a number   if (number &lt; 100)     PRINT "I AM STRONG!"   else     PRINT "I AM WEAK!"   end if </pre>

An equivalent mutant yields identical results to the original program, presenting a challenge in identifying mutants during mutation analysis. The essence of the issue is that test suites fail to detect equivalent mutants caused by the introduced fault, requiring manual intervention for their identification and elimination. Studies indicate that such mutants are not uncommon; they frequently occur, at times comprising over 50% of all mutants (Grün et al., 2009). Table 25 provides an example of an equivalent mutant.

Table 25. An original program and its an equivalent mutant.

<b>Original Program</b>	<b>Equivalent Mutant</b>
<pre> a = 2 READ b   if (b == 2)     PRINT "b"     b = a + b   end if </pre>	<pre> a = 2 READ b   if (b == 2)     PRINT "b"     b = a * b   end if </pre>

## 5.2. Higher-Order Mutation Testing

The concept of second-order mutation testing introduced by Offutt (1992) in his research on the coupling effect in mutation analysis. Offutt claimed that more complex

faults could be detected through second-order mutation, observing a decrease in the survival rate of second-order mutants in experiments. This idea was further explored by Polo et al. (2009), Madeyski (2008), and Mateo et al. (2012), who demonstrated that combining first-order mutants to form second-order mutants could reduce the number of equivalent mutants without compromising the quality of the test suite. However, these studies also indicated a potential challenge: some second-order mutants proved to be harder to eliminate in mutation analysis.

The notion of higher-order mutation testing involves introducing multiple artificial faults into the original program. The number of faults inserted determines the order of the mutant; for instance, a mutant created with two faults is a second-order mutant, while one with three faults is a third-order mutant. Table 26 illustrates a higher-order mutant with two faults, categorizing it as a second-order mutant.

Table 26. Original program and its higher-order mutant.

<b>Original Program</b>	<b>Higher-Order Mutant</b>
<pre> Read A and B multiplication = 1   if (A &gt; B)     multiplication = A * B     result = multiplication / 2   end if </pre>	<pre> Read A and B multiplication = 1   if (A &gt; B)     multiplication = A + B     result = multiplication * 2   end if </pre>

In practical applications, many higher-order mutants turn out to be irrelevant. This is because if a test can eliminate a first-order mutant that is part of a higher-order mutant, it can usually also eliminate the higher-order mutant, owing to the coupling effect hypothesis in mutation testing. Nevertheless, Jia and Harman (2009) highlight that certain higher-order mutants are beneficial. They identified various categories of higher-order mutants exhibiting unique characteristics. To generate these mutants, the authors used three search-based algorithms. In their experiments with ten different programs, they found that approximately 67% of the generated higher-order mutants were indeed more challenging to kill compared to the first-order mutants.



### 5.3. Methodology

In this section, we discuss the justification for selecting the investigated methods and provide details on the implementation of the proposed genetic algorithm and the random search algorithm.

Higher-order mutation testing is acknowledged for its potential usefulness, offering advanced capabilities over first-order mutation testing. It is adept at simulating more complex and realistic programming faults, which requires the use of improved test optimization techniques (Jia and Harman, 2009). The primary goal of the proposed method is to tackle a significant challenge in higher-order mutation testing: the reduction of the number of mutants at the mutant generation stage.

To generate higher-order mutants, search-based testing techniques, particularly meta-heuristics, are employed with the aim of reducing costs. The selection of this technique is based on three main reasons. Firstly, the large set of mutants in higher-order mutation testing includes both useful and less beneficial mutants. A selective process that identifies the more useful mutants can significantly reduce costs. Secondly, a meta-heuristic search is guided by a fitness function, allowing for a more efficient and strategic search process compared to exhaustive techniques. This method can find optimal solutions without having to explore all possible options. Lastly, numerous studies in the literature present promising results for search-based mutation testing, lending support to this approach.

The choice of genetic algorithms for this study is particularly important. Genetic algorithms are renowned for their effectiveness in large search spaces and their capability to navigate towards optimal solutions efficiently. Their ability to adapt and evolve in response to the specific requirements of the problem space aligns well with the dynamic and complex nature of higher-order mutation testing. They are designed to avoid common issues such as becoming trapped in local optima, a frequent problem in other optimization methods. Their suitability for tasks involving the evaluation and selection of the best candidates from a large pool of options makes them ideal for optimizing the selection of higher-order mutants.

### **5.3.1. A Genetic Algorithm for Higher-Order Mutant Generation**

In this section, we discuss our method for generating higher-order mutants using a genetic algorithm. The algorithm is designed to identify the fittest mutants in a given search space. We discuss the specifics of chromosome representation and the fitness function, which are critical components of our approach for higher-order mutant generation.

#### **5.3.1.1. Chromosome Representation**

The algorithm operates on a population of individual entities that represent potential solutions. These solutions are embodied in chromosomes, which act upon various features. In this context, a higher-order mutant is analogous to a chromosome. This chromosome is conceptualized as an array of strings, where each string represents a line of code from the source code of a higher-order mutant. Thus, every line of a higher-order mutant is encapsulated within these chromosomes. The mutated statements within the chromosomes are derived from their corresponding first-order mutants. To accurately represent higher-order mutants, the chromosomes may contain two or more faults.

#### **5.3.1.2. Fitness Function**

A fitness function plays a crucial role in a genetic algorithm, as it assesses how close a candidate solution is to the ideal solution for a given problem. Essentially, it determines the quality of a solution, categorizing it as good, poor, or somewhere in between. This evaluation is vital for the genetic algorithm to effectively select the most suitable solution from the available search space.

Once the initial setup is complete, the algorithm proceeds to the selection phase. During this phase, the fitness of each mutant is computed. The fitness score ranges from 0 to 1. A score closer to 1 indicates that the mutant is relatively easy to eliminate, while a lower score suggests that the mutant is more resilient and may require multiple test cases to be effectively neutralized.

To calculate the fitness value  $f(c)$  for a mutant  $c$  in a population, we consider  $TC_n$ ,

which represents the  $n^{\text{th}}$  test case in the test suite that successfully kills the mutant  $c$ . With  $TC_{total}$  being the total number of test cases, the fitness value is derived using these variables (see Equation 1). This calculation helps in determining the effectiveness of each mutant in terms of how challenging it is to eliminate them, guiding the selection of the most robust mutants for further analysis.

$$f(c) = \frac{\sum_{i=1}^n TC_n}{TC_{total}} \quad (1)$$

### 5.3.1.3. Initialization

The initial population is created by randomly combining several first-order mutants. The number of first-order mutants to be merged can vary based on user preference. For instance, to generate a third-order mutant, three first-order mutants would need to be combined. This initial merging process sets the stage for the development of higher-order mutants.

### 5.3.1.4. Selection

During the selection phase, certain higher-order mutants are chosen for the production of offspring in subsequent stages. This phase is crucial as it identifies the candidate solutions that will be carried forward to the next generation. In this study, we have experimented with four widely used selection mechanisms: roulette wheel, tournament, rank, and truncation selection. Each of these mechanisms offers a different approach to selecting the most promising mutants when generating higher-order mutants. The user selects which selection method is to be executed in the implementation.

**Roulette Wheel Selection:** In the first version, the roulette wheel selection method is used to update the statements. This technique focuses on selecting the most suitable parents from a group for the creation of offspring. Unlike a traditional roulette wheel where each slot has an equal chance of selection due to uniform sizes, the roulette wheel in genetic algorithms is designed to be weighted. This means that individuals with higher fitness values have a higher probability of being selected.

For each chromosome  $c$  and its corresponding fitness value  $f(c)$  in a population  $P$  consisting of  $n$  chromosomes,  $P = \{c_1, c_2, c_3, \dots, c_n\}$ , the selection probability  $p(c_k)$  is calculated using the formula 2:

$$p(c_k) = \frac{f(c_k)}{\sum_{i=1}^n f(c_i)}, \text{ where } l = 1, 2, 3, \dots, n \quad (2)$$

This formula determines the relative fitness of each individual in the population. The roulette wheel is then divided into sections based on these calculated proportions. The wheel is spun  $n$  times, where  $n$  is the population size. Upon each spin, two individuals are chosen from the respective section where the wheel stops. This method ensures that individuals with higher fitness are more likely to be chosen, guiding the algorithm towards potentially more successful solutions.

**Tournament Selection:** In the second version, tournament selection is employed. This method involves picking a random subset of individuals from the population and then conducting a “tournament” among them. The individual with the highest fitness score within this subset is selected as a parent for the next generation. This process is typically repeated several times to select multiple parents. These parents are then used for creating offspring through crossover and mutation operations. Within the context of mutation testing, each “individual” in the population represents a test suite. The fitness in this scenario reflects the ability of the test suite to detect mutants.

**Rank Selection:** The third version of the algorithm incorporates rank selection. This technique involves first sorting individuals based on their fitness levels. Following this sorting, individuals are selected for reproduction based on their rank within this ordered list. Unlike methods such as roulette wheel selection, which depend on the absolute fitness values to determine the probability of selection, rank selection focuses on the relative position of an individual within the population. The primary advantage of this approach is its ability to prevent premature convergence by prioritizing relative fitness over absolute fitness. This aspect is particularly beneficial in mutation testing, where the objective function might have complex landscapes. By focusing on relative fitness, rank selection ensures a more diverse genetic pool, which is crucial for

effectively navigating rugged objective functions in mutation testing.

**Truncation Selection:** In the final version, the truncation selection method is implemented. This approach involves selecting the top-performing individuals from the population and discarding the ones with lower fitness. It essentially focuses on replicating the most fit individuals for the subsequent generation. Truncation selection is known for its elitist nature, making it particularly effective in situations where quick convergence towards a high-quality solution is the goal.

The formula (3) is used for determining the number of individuals to be selected is as follows:

$$\text{Number of Selected Individuals} = \lceil T \times N \rceil \quad (3)$$

The truncation threshold  $T$  is a crucial factor, ranging between 0 and 1. For example, setting  $T$  to 0.2 implies that only the top 20% of individuals, as determined by their mutation score, are selected. The symbol  $N$  represents the total population size.

#### 5.3.1.5. *Mating*

During the mating stage, the algorithm applies crossover and mutation to the selected higher-order mutants, based on specific criteria:

**Crossover:** This process involves combining two chosen higher-order mutants to create offspring. The number of crossover points is determined based on user preference. In this study, a double point crossover is used. It is important to ensure that the offspring generated are not merely first-order mutants. If such a situation arises, an additional mutation is randomly applied to generate a second-order mutant.

**Mutation:** Mutation in higher-order mutants involves either adding or removing a first-order mutant randomly. When adding, a first-order mutant is randomly selected from the pool. In contrast, when removing a first-order mutant, the choice is made among those contributing to the higher-order mutant. Users have the discretion to decide the number of first-order mutants to be added or removed. This process should not result in the transformation of a higher-order mutant back to a first-order mutant.

### 5.3.1.6. Stopping Condition

Determining the appropriate stopping point for a genetic algorithm is critical for its efficiency and effectiveness. Several criteria can be considered:

- **Fixed Number of Generations:** The simplest approach, where the algorithm halts after reaching a predetermined number of generations.
- **Convergence:** The algorithm stops when the population converges, indicated by minimal variation in the fitness values across individuals. This may suggest the discovery of an optimum, though not necessarily.
- **Threshold Fitness Value:** The process can also be halted once an individual attains or surpasses a specific fitness threshold. This criterion is useful when targeting a known solution or a minimum acceptable quality of solution.

In this research, the termination of the algorithm is controlled by a user-defined parameter, specifically the maximum number of distinct higher-order mutants. Once the algorithm reaches this limit, it ceases operation and outputs a list of the higher-order mutants that have been identified. The detailed pseudocode outlining the stages of the genetic algorithm is presented in Algorithm 4. Additionally, a flow chart of the entire algorithmic process can be found in Figure 29.

---

#### Algorithm 4 Proposed Genetic Algorithm

---

**Require:** *firstOrderMutants, size, selectionMethod, cPoint, mRatio*

```
1:  $i = 0$ 
2:  $higherOrderMutants \leftarrow \emptyset$ 
3:  $population[i] \leftarrow initializePopulation(firstOrderMutants, size)$ 
4:  $population[i].calculateFitness()$ 
5: while  $stoppingCondition \neq true$  do
6:    $parent \leftarrow population[i].select(selectionMethod)$ 
7:    $offspring \leftarrow parents.crossover(cPoint)$ 
8:    $offspring \leftarrow parents.mutation(mRatio)$ 
9:    $population[i].calculateFitness()$ 
10:   $population.discardLowestScoringMutants()$ 
11:   $population.add(offspring)$ 
12:   $i \leftarrow i + 1$ 
13: end while
14:  $higherOrderMutants.add(population)$ 
15: return  $higherOrderMutants$ 
```

---

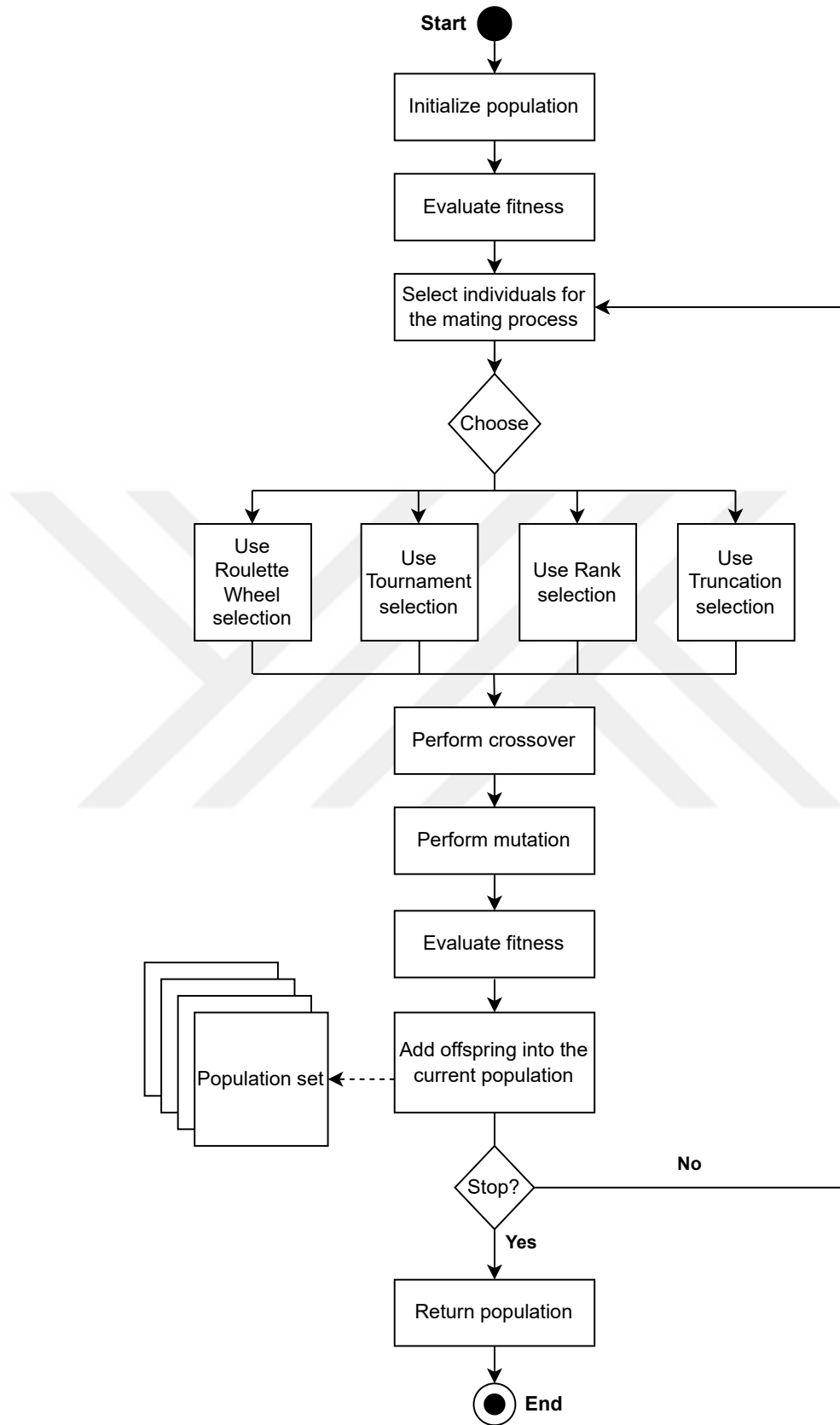


Figure 29. Flow chart of the proposed genetic algorithm.

The time complexity of the algorithm is estimated as follows:  $O(k \times n \times m)$  where  $k$  represents the number of iterations,  $n$  stands for the chromosome length,  $m$  is the population size.

### 5.3.2. A Random Search Algorithm for Higher-Order Mutant Generation

Random search is a straightforward, non-heuristic method for exploring the space of potential solutions. Unlike genetic algorithms, which evolve a population through generations using selection, crossover, and mutation, a random search algorithm uniformly samples across the search space, assessing solutions that are generated randomly. In this approach, the selection parameters are entirely random, lacking any systematic or intelligent strategy for sampling from the search space.

---

#### Algorithm 5 Proposed Random Search Algorithm

---

**Require:** *firstOrderMutants*

```

1:  $i = 0$ 
2:  $higherOrderMutants \leftarrow \emptyset$ 
3: while  $stoppingCondition \neq true$  do
4:    $randomHigherOrderMutant \leftarrow makeHigher(firstOrderMutants)$ 
5:    $randomHigherOrderMutant.execute()$ 
6:    $fitness \leftarrow randomHigherOrderMutant.calculateFitness()$ 
7:   if  $fitness == OK$  then
8:      $higherOrderMutants \leftarrow randomHigherOrderMutant$ 
9:   end if
10:   $i \leftarrow i + 1$ 
11:   $population.add(offspring)$ 
12: end while
13: return  $higherOrderMutants$ 

```

---

In this study, the random search method is employed to independently select candidate higher-order mutants from the pool of all available higher-order mutants. This process requires an initial, randomly chosen set of first-order mutants to create higher-order mutants. These mutants are then evaluated for their fitness, and based on these evaluations, selected mutants are added to a list of higher-order mutants.

Using random search as a benchmark is valuable for assessing the effectiveness of more complex algorithms. Demonstrating that the genetic algorithm surpasses the performance of a random search provides concrete evidence of its added value. It suggests that the algorithm's success is not merely due to random chance but is a result



of its sophisticated navigation through the search space. The steps and processes of the random search algorithm are detailed in the pseudocode provided in Algorithm 5.

### 5.3.3. *Research Questions*

The objective of this chapter is to introduce a genetic algorithm to evaluate the effects of different selection methods when generating higher-order mutants. To achieve a detailed evaluation, we have divided **RQ4**, mentioned at the start of this chapter, into three sub-research questions:

- **RQ4a:** *What is the effectiveness of different selection methods in minimizing the generation of equivalent higher-order mutants?* This question investigates various selection strategies used during the genetic algorithm's selection phase. The goal is to identify the selection method that most effectively reduces the number of equivalent mutants during the creation of higher-order mutants.
- **RQ4b:** *How do different selection methods compare in terms of execution cost?* This research question seeks to assess the efficiency of various selection methods. Specifically, it aims to determine which method is the fastest and which is the slowest, on average, among those implemented in this study.
- **RQ4c:** *What proportion of higher-order mutants is produced at each mutation order?* A significant challenge is managing the volume of mutants generated. This question intends to examine whether varying the order of mutants can still reduce the total number of mutants generated while still yielding effective results.

## 5.4. *Experimental Design*

This section explains the experimental design used to evaluate the research objectives. It covers the setup of the test environment, the details of subject programs including their size, mutants, and test cases, configuration of genetic algorithm parameters, and analysis of results.

### 5.4.1. Test Environment

Our test environment, depicted in Figure 30, was established to create first-order mutants using the following configuration and tools:

- The experiments were conducted on a desktop computer equipped with a Windows 11 operating system and powered by an Intel i7 9700k 2.8 GHz processor.
- For this experiment, we used VisualMutator based on the evaluation from Chapter 3. VisualMutator was used to generate, compile, and run first-order mutants, which were then used as inputs for assessing test cases.
- To develop and manage test cases in the test suite, IntelliTest and SentryOne tools were used. These tools are automated test case generators tailored for unit testing in the NUnit format. The primary objective of using both tools was to ensure comprehensive coverage of the subject programs, aiming to eliminate all generated first-order mutants. In instances where mutants remained uneliminated, additional test cases were incorporated manually.

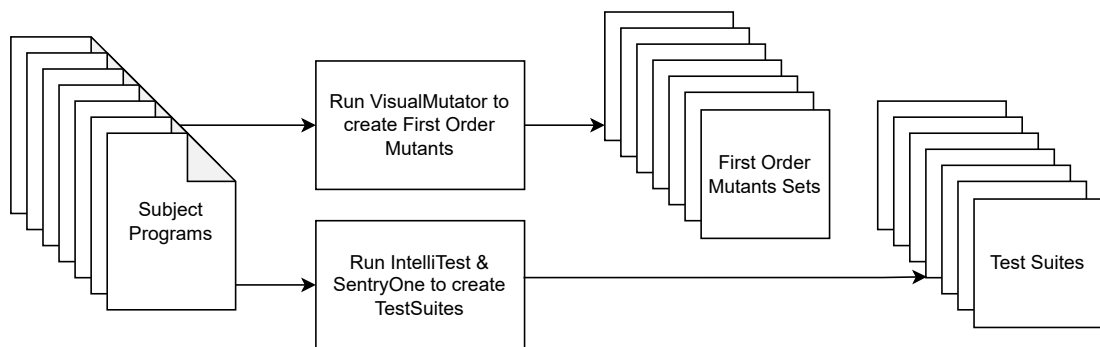


Figure 30. Test environment to create first-order mutants.

Once the first-order mutants for all the subject programs have been generated, the study proceeds to execute both the random search algorithm and four different versions of the genetic algorithm. These are implemented to create higher-order mutants, allowing for the evaluation and comparison of results. Figure 31 illustrates this process through a diagram.

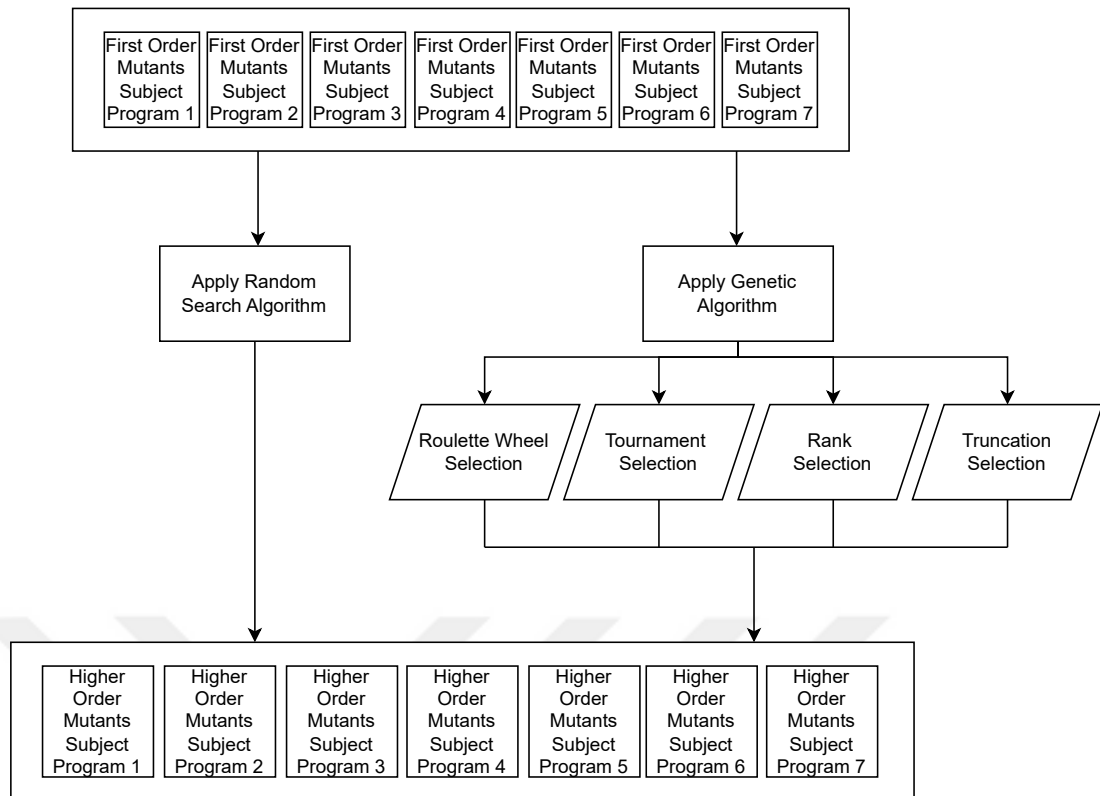


Figure 31. Experiment details to create higher-order mutants.

#### 5.4.2. Subject Programs

The subject programs used in this study are developed in C# and their details, including size, the number of first-order mutants, and the number of test cases, are provided in Table 27. These programs are described as follows:

- **TriangleType:** This program determines the type of a given triangle - whether it is equilateral, isosceles, or scalene.
- **PrintPrimes:** It is designed to identify and display all prime numbers up to a specified input value.
- **CalculateDays:** This program computes and outputs the number of days between two specified dates.
- **HashTable:** This program involves creating a hash table based on data from a given test file and then retrieving values using their respective keys.
- **CocktailSort:** A variation of the bubble sort algorithm, this program sorts an

input array by traversing from both the start and the end, rather than just from the beginning.

- **MatchPattern:** It searches for and identifies occurrences of a specified pattern within an expression.
- **MoonPhases:** This program calculates the current phase of the moon and estimates the remaining days in the lunar cycle.

Table 27. Subject programs.

Subject Program	Lines of Code	# of First-Order Mutants	# of Test Cases
TriangleType	123	102	77
PrintPrimes	49	99	65
CalculateDays	51	85	89
HashTable	107	172	56
CocktailSort	75	204	83
MatchPattern	62	68	71
MoonPhases	165	381	95

#### 5.4.3. Genetic Algorithm Parameter Settings

After conducting various experiments to optimize the genetic algorithm parameters, the following settings were selected for the proposed genetic algorithm:

- **Probability of Crossover:** This value is set at 0.7 which is commonly utilized in genetic algorithms and is considered an effective balance between exploring new solutions and exploiting existing effective solutions.
- **Probability of Mutation:** A lower mutation probability is used as 0.07 to introduce a moderate level of randomness into the algorithm, without significantly altering the primary characteristics of the individuals. This helps in avoiding local optima.
- **Maximum Number of Iterations:** Fixed at 1000 iterations. It was observed that beyond this point, there were no significant changes in the results, indicating an optimal stopping point for the algorithm.

- **Chromosome Size:** Determined to be equal to the number of lines of code in the subject program. This size effectively represents the crucial aspects of the subject program and facilitates better optimization.

To ensure the reliability of these settings, the experiments were repeated 30 times for each subject program. This repetition was aimed at determining if there were any variations in the results across trials. After 30 trials, no significant changes were observed in the results, leading to the conclusion of the experimental phase.

## 5.5. Results

This section details the results obtained from averaging 30 experiments conducted on each of the seven test programs.

The data in Table 28 addresses RQ4a, which asks: “*What is the effectiveness of different selection methods in minimizing the generation of equivalent higher-order mutants?*” The table compares the proportion of equivalent mutants generated for each test program across different methods: random search and four genetic algorithm (GA) versions with distinct selection strategies (roulette wheel, tournament, rank, and truncation selection). While each approach produced equivalent mutants, the ratios varied.

Among all methods tested, random search was found to be the least efficient, with an average equivalent mutant ratio of 26.8%. In contrast, the truncation selection variant of the proposed genetic algorithm exhibited a slightly lower equivalent mutant ratio (18%) compared to the rank selection approach, making it the most effective among the tested strategies.

Specifically, the MatchPattern program showed a 33% ratio of equivalent mutants when using random search, while the HashTable program demonstrated an 11% ratio of equivalent mutants when using the GA with the roulette wheel selection strategy.

Table 29 provides an answer to RQ4b: “*How do different selection methods compare in terms of execution cost?*” This research question aims to identify the fastest performing selection strategy in terms of average execution time. The results indicate that rank selection was the least efficient in this regard, taking an average of

Table 28. Ratio of generated equivalent mutants.

Subject Program	Random Search	GA with Roulette Wheel Selection	GA with Tournament Selection	GA with Rank Selection	GA with Truncation Selection
TriangleType	30%	25%	25%	21.2%	14%
PrintPrimes	30.3%	28.4%	24.1%	25.5%	21%
CalculateDays	23.2%	15.2%	18.8%	15%	20%
HashTable	16%	11%	15%	13%	15%
CocktailSort	22.5%	18%	12.3%	16.3%	14.2%
MatchPattern	33%	20.7%	17.6%	22.5%	20.4%
Moon Phases	28.1%	21.5%	25%	20.1%	18%
<b>Average</b>	<b>26.8%</b>	<b>20%</b>	<b>19.7%</b>	<b>19.1%</b>	<b>18%</b>

Table 29. Execution cost of different selection strategies.

Subject Program	Roulette Wheel Selection	Tournament Selection	Rank Selection	Truncation Selection
TriangleType	10.2	10.4	13	9.4
PrintPrimes	11	11.3	13.2	10.2
CalculateDays	8.2	8.5	10.5	7.3
HashTable	10.2	10.6	12.3	10
CocktailSort	14	15.3	16	12.3
MatchPattern	8.5	9	10	6.3
MoonPhases	12.7	12.9	15	11.1
<b>Average</b>	<b>10.7</b>	<b>11.1</b>	<b>12.9</b>	<b>9.5</b>

12.9 minutes. Following this, tournament selection was the next slowest, averaging 11.1 minutes.

Comparatively, tournament selection was found to be 0.4 minutes slower on average than roulette wheel selection. Overall, the quickest selection was truncation selection, averaging an execution time of 9.5 minutes.

Looking at individual programs, MatchPattern recorded the fastest execution time at 6.3 minutes using truncation selection. Conversely, the slowest execution time was observed in the CocktailSort program, which took 16 minutes using rank selection.

Table 30 is dedicated to addressing RQ4c, which asks: “*What proportion of higher-order mutants is produced at each mutation order?*” This question explores whether generating mutants of a higher order can lead to a reduced number of mutants that yield more effective results. The results clearly indicate that an increase in the mutation order correlates with a decrease in the number of mutants generated.

For this experiment, all subject programs were subjected to both random search and the various versions of the genetic algorithm, with the average results displayed in Table 30. For example, the proportion of second-order mutants generated using GA with truncation selection stands at 50.1%. However, this percentage drops to 14.2% for fourth-order mutations, signifying an overall reduction of 35.9% in the mutant ratio as the order increases. This trend demonstrates the effectiveness of higher-order mutations in decreasing the overall quantity of mutants while potentially improving the quality of the results.

Table 30. Percentage of the generated higher-order mutants from each mutation order.

<b>Subject Program</b>	<b>Random Search</b>	<b>GA with Roulette Wheel Selection</b>	<b>GA with Tournament Selection</b>	<b>GA with Rank Selection</b>	<b>GA with Truncation Selection</b>
SecondOrder	55.5%	46.3%	55.2%	64.2%	50.1%
ThirdOrder	37.6%	29.4%	26.1%	23.1%	29.2%
FourthOrder	30.2%	18.1%	15.2%	15.5%	14.2%

Overall, these research questions and experiments collectively explore the complications of higher-order mutation testing, focusing on the optimization of selection strategies, resource efficiency, and the structural characteristics of higher-order mutants. Our results show that the generation of higher-order mutants can be facilitated with genetic algorithms using first-order mutants generated from mutation testing tools.

## **5.6. Conclusion**

This chapter focused on higher-order mutation and a search-based mutation testing approach using genetic algorithms. These algorithms were applied to first-order mutants to generate higher-order mutants, aiming to tackle the issue of equivalent mutants. To thoroughly evaluate its effectiveness and reliability, we selected seven diverse test programs, each accompanied by a specific test suite tailored to detect and eliminate higher-order mutants. These test environments provided a comprehensive platform for assessing our algorithm's performance.

As a result, **RQ4** outlined in Section 1.4 is answered: *“How does the implementation of genetic algorithms as a search strategy in higher-order mutation testing impact the efficiency of generating high-quality mutants, particularly in reducing the production of equivalent high-order mutants?”* Four genetic algorithm variants, each incorporating a different selection method, for higher-order mutant generation was proposed. The findings indicate that genetic algorithm variants effectively reduces the number of equivalent mutants when forming higher-order mutants. Specifically, truncation selection is found to be the best selection strategy. The findings of this chapter have been synthesized into a research article (Uzunbayir and Kurtel, 2023b).



## CHAPTER 6: CONCLUSION

### 6.1. Summary

In this thesis, mutation testing was comprehensively explored via experimentation. We asked four RQs in Section 1.4 and addressed them in the subsequent chapters separately.

Firstly, various aspects and methodologies were blended to form a comprehensive overview of the field to address **RQ1**. The background of the topic was described, which involved identifying the fundamental principles, applications, and challenges of mutation testing involving artificial intelligence approaches. It was pointed out that the steady increase in academic studies and tools indicate that the area is growing rapidly.

Following this, the focus shifted to mutation testing tools specific to the C# programming language to address **RQ2**. This segment of the thesis offered an in-depth analysis of these tools, comparing and contrasting their features and effectiveness. This comparative approach is invaluable for guiding testers in selecting the most suitable tools for C# applications, providing critical insights into the practical aspects of mutation testing.

Next, EvoColony, an innovative approach that merges genetic algorithms and ant colony optimization for effective mutation test suite reduction, is introduced to address **RQ3**. This part of the study not only demonstrated EvoColony's greater efficiency compared to other methods, but also highlighted its dual optimization strategy.

Lastly, higher-order mutation testing through the lens of genetic algorithms is explored to address **RQ4**. Several selection methods for the algorithm is discussed and the ability to reduce the number of equivalent mutants is evaluated. A comparative analysis presented here demonstrates that the performance of these selection methods varies, with particular emphasis on the efficiency of a genetic algorithm-based approach compared to other methods.

Together, all RQs of this thesis coalesce to form a comprehensive and detailed examination of mutation testing, representing a significant contribution to the field. Therefore, thesis statement mentioned in Section 1.1 has been successfully addressed.

## 6.2. *Future Work*

The thesis outlines several important paths for further exploration in the domain of mutation testing. Of these, a key focal area is the integration of mutation testing with various other testing techniques, potentially leading to the creation of new, innovative hybrid methods that exploit aspects of traditional and higher-order mutation testing, combined with search-based and machine learning approaches. For example, these approaches can be used for various purposes, for example, natural language processing (NLP), in which certain requirements are extracted from texts. This involves parsing the text, identifying key phrases and terms, and structuring them into formal requirements. Once requirements are extracted using NLP, mutation testing can be used to validate the tests derived from these requirements, ensuring they align with the intended functionality. Combining NLP for extraction and mutation testing for validation creates an iterative process where extracted requirements are continuously refined based on the effectiveness of the tests.

Furthermore, there is strong potential for advancing existing hybrid methods, such as EvoColony. Adapting these approaches to address would allow the development of more intricate mutation types and their integration with other sophisticated artificial intelligence techniques. This evolution would eventually lead to more adaptive and efficient testing strategies.

Investigating alternative selection strategies and optimization techniques for higher order mutation testing is another emerging research direction. Investigating alternative selection strategies and different search-based optimization methods, e.g., ant colony optimization or particle swarm optimization, could be highly beneficial. Comparative analyses involving larger-scale projects under test conditions similar to the ones employed here may provide valuable insights, enabling us to refine our approach and potentially uncover a more universally effective solution to the challenges associated with higher-order mutants and the equivalent mutant problem.

These areas of research signify the potential for continued growth and innovation in mutation testing. Pursuing these directions, the field can bring more advanced and efficient tools and methodologies for software testing.

### **6.3. *Final Remarks***

Overall, this study underlines the dynamic and evolving nature of mutation testing in software engineering. By incorporating traditional methods with advanced computational techniques, the approach taken in this thesis not only draws on the current developments in mutation testing, but also opens opportunities for innovative future solutions. This work lays a strong foundation for further exploration and development in the field, highlighting the vital role of mutation testing in enhancing software quality and testing efficiency. Thus, these findings will significantly contribute to a deeper and more nuanced understanding of mutation testing techniques, which will be increasingly valuable as the field continues to evolve.

## REFERENCES

- Abuljadayel, A. and Wedyan, F. (2018) *An approach for the generation of higher order mutants using genetic algorithms*, International Journal of Intelligent Systems and Applications, Vol. 12 (1), pp. 34.
- Adamopoulos, K., Harman, M. and Hierons, R. M. (2004) How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution, *Genetic and Evolutionary Computation Conference*, Springer, pp. 1338–1349.
- Adams, B. and McIntosh, S. (2016) Modern release engineering in a nutshell—why researchers should care, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5, IEEE, pp. 78–90.
- Andrews, J. H., Briand, L. C., Labiche, Y. and Namin, A. S. (2006) *Using mutation analysis for assessing and comparing testing coverage criteria*, IEEE Transactions on Software Engineering, Vol. 32 (8), pp. 608–624.
- Arasteh, B., Imanzadeh, P., Arasteh, K., Gharehchopogh, F. S. and Zarei, B. (2022) *A source-code aware method for software mutation testing using artificial bee colony algorithm*, Journal of Electronic Testing, Vol. pp. 1–14.
- Bashir, M. B. and Nadeem, A. (2017) *Improved genetic algorithm to reduce mutation testing cost*, IEEE Access, Vol. 5, pp. 3657–3674.
- Beller, M., Wong, C.-P., Bader, J., Scott, A., Machalica, M., Chandra, S. and Meijer, E. (2021) What it would take to use mutation testing in industry—a study at facebook, *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, pp. 268–277.
- Boubeta-Puig, J., Medina-Bulo, I. and García-Dominguez, A. (2011) Analogies and differences between mutation operators for ws-bpel 2.0 and other languages, *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, IEEE, pp. 398–407.
- Budd, T. A. and Angluin, D. (1982) *Two notions of correctness and their relation to testing*, Acta Informatica, Vol. 18 (1), pp. 31–45.
- Byoungju, C. and Mathur, A. P. (1993) *High-performance mutation testing*, Journal of

Systems and Software, Vol. 20 (2), pp. 135–152.

Cai, G., Su, Q. and Hu, Z. (2021) *Automated test case generation for path coverage by using grey prediction evolution algorithm with improved scatter search strategy*, Engineering Applications of Artificial Intelligence, Vol. 106, pp. 104454.

Chekam, T. T., Papadakis, M., Traon, Y. L. and Harman, M. (2017) An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 597–608.

Chen, L. and Zhang, L. (2018) Speeding up mutation testing via regression test selection: An extensive study, *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 58–69.

Chen, Y. T., Gopinath, R., Tadakamalla, A., Ernst, M. D., Holmes, R., Fraser, G., Ammann, P. and Just, R. (2020) Revisiting the relationship between fault detection, test adequacy criteria, and test set size, *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 237–249.

CREAM (2008) *Cream*. [Online]. Available at: <http://galera.ii.pw.edu.pl/adr/CREAM/>. (Accessed: 16 May 2023).

Delamaro, M. (1993) *Proteum—a mutation analysis based testing environment*, Master's thesis, University of São Paulo.

DeMillo, R. A., Guindi, D. S., McCracken, W., Offutt, A. J. and King, K. (1988) An extended overview of the mothra software testing environment, *Workshop on Software Testing, Verification, and Analysis*, IEEE, pp. 142–151.

DeMillo, R. A., Krauser, E. W. and Mathur, A. P. (1991) Compiler-integrated program mutation, *Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International*, IEEE, pp. 351–356.

DeMillo, R. A., Lipton, R. J. and Sayward, F. G. (1978) *Hints on test data selection: Help for the practicing programmer*, Computer, Vol. 11 (4), pp. 34–41.

Derezińska, A. (2006) Advanced mutation operators applicable in c# programs, *Software Engineering Techniques: Design for Quality*, Springer, pp. 283–288.

Derezińska, A. and Rudnik, M. (2017) Evaluation of mutant sampling criteria in object-oriented mutation testing, *Computer Science and Information Systems*

(FedCSIS), *2017 Federated Conference on*, IEEE, pp. 1315–1324.

Dorigo, M., Birattari, M. and Stutzle, T. (2006) *Ant colony optimization*, IEEE Computational Intelligence Magazine, Vol. 1 (4), pp. 28–39.

Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R. and Guimarães, M. P. (2019) *Machine learning applied to software testing: A systematic mapping study*, IEEE Transactions on Reliability, Vol. 68 (3), pp. 1189–1212.

Garg, A., Ojdanic, M., Degiovanni, R., Chekam, T. T., Papadakis, M. and Le Traon, Y. (2023) *Cerebro: Static subsuming mutant selection*, IEEE Transactions on Software Engineering, Vol. 49 (1), pp. 24–43.

Gong, D., Wang, T., Su, X. and Zhang, Y. (2022) *Equivalent mutants detection based on weighted software behavior graph*, International Journal of Software Engineering and Knowledge Engineering, Vol. 32 (06), pp. 819–843.

Gopinath, R., Ahmed, I., Alipour, M. A., Jensen, C. and Groce, A. (2017) *Mutation reduction strategies considered harmful*, IEEE Transactions on Reliability, Vol. 66 (3), pp. 854–874.

Gopinath, R., Alipour, M. A., Ahmed, I., Jensen, C. and Groce, A. (2016) On the limits of mutation reduction strategies, *Proceedings of the 38th International Conference on Software Engineering*, pp. 511–522.

Grün, B. J., Schuler, D. and Zeller, A. (2009) The impact of equivalent mutants, *2009 International Conference on Software Testing, Verification, and Validation Workshops*, IEEE, pp. 192–199.

Haga, H. and Suehiro, A. (2012) Automatic test case generation based on genetic algorithm and mutation analysis, *2012 IEEE International Conference on Control System, Computing and Engineering*, IEEE, pp. 119–123.

Halabi, D. and Shaout, A. (2016) *Mutation testing tools for java programs—a survey*, International Journal of Computer Science Engineering, Vol. 5, pp. 11–22.

Harman, M., Jia, Y. and Langdon, W. B. (2010) A manifesto for higher order mutation testing, *Third International Conference on Software Testing, Verification, and Validation Workshops*, IEEE, pp. 80–89.

Harman, M., Jia, Y. and Langdon, W. B. (2011) Strong higher order mutation-based test data generation, *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th*

- European Conference on Foundations of Software Engineering*, New York, NY, USA, Association for Computing Machinery, p. 212–222.
- Hierons, R., Harman, M. and Danicic, S. (1999) *Using program slicing to assist in the detection of equivalent mutants*, *Software Testing, Verification and Reliability*, Vol. 9 (4), pp. 233–262.
- Hussain, S. (2008) *Mutation clustering*, Master's thesis, King's College London.
- Jackson, D. and Woodward, M. R. (2001) *Parallel firm mutation of java programs*, *Mutation Testing for the New Century*, Springer, pp. 55–61.
- Jammalamadaka, K. and Parveen, N. (2022) *Equivalent mutant identification using hybrid wavelet convolutional rain optimization*, *Software: Practice and Experience*, Vol. 52 (2), pp. 576–593.
- Jatana, N. and Suri, B. (2020) *Particle swarm and genetic algorithm applied to mutation testing for test data generation: A comparative evaluation*, *Journal of King Saud University-Computer and Information Sciences*, Vol. 32 (4), pp. 514–521.
- Jatana, N., Suri, B. and Rani, S. (2017) *Systematic literature review on search based mutation testing*, *e-Informatica Software Engineering Journal*, Vol. 11 (1), pp. 59–76.
- Jia, Y. and Harman, M. (2009) *Higher order mutation testing*, *Information and Software Technology*, Vol. 51 (10), pp. 1379–1393.
- Jia, Y. and Harman, M. (2010) *An analysis and survey of the development of mutation testing*, *IEEE Transactions on Software Engineering*, Vol. 37 (5), pp. 649–678.
- Jia, Y. and Harman, M. (2011) *An analysis and survey of the development of mutation testing*, *IEEE Transactions on Software Engineering*, Vol. 37 (5), pp. 649–678.
- Kapoor, K. (2006) *Formal analysis of coupling hypothesis for logical faults*, *Innovations in Systems and Software Engineering*, Vol. 2 (2), pp. 80–87.
- Khanfir, A., Degiovanni, R., Papadakis, M. and Traon, Y. L. (2023) *Efficient mutation testing via pre-trained language models*. [Online]. Available at: <https://arxiv.org/pdf/2301.03543.pdf>. (Accessed: 16 May 2023).
- Kim, S., Clark, J. A. and McDermid, J. A. (2000) *Class mutation: Mutation testing for object-oriented programs*, *Proc. Net. ObjectDays*, Citeseer, pp. 9–12.
- King, K. N. and Offutt, A. J. (1991) *A fortran language system for mutation-based software testing*, *Software: Practice and Experience*, Vol. 21 (7), pp. 685–718.

- Kintis, M. (2016) *Effective Methods to Tackle the Equivalent Mutant Problem when Testing Software with Mutation*, PhD thesis, Athens University of Economics and Business.
- Kintis, M., Papadakis, M. and Malevris, N. (2010) Evaluating mutation testing alternatives: A collateral experiment, *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, IEEE, pp. 300–309.
- Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N. and Le Traon, Y. (2018) *How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults*, *Empirical Software Engineering*, Vol. 23 (4), pp. 2426–2463.
- Klampfl, L., Chetouane, N. and Wotawa, F. (2020) Mutation testing for artificial neural networks: An empirical evaluation, *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, pp. 356–365.
- Krasner, H. (2022) The cost of poor software quality in the us: A 2022 report, <https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf>. Accessed: 20 February 2023.
- Krauser, E. (1991) *Compiler-Integrated Software Testing*, PhD thesis, Purdue University.
- Kusharki, M. B., Misra, S., Muhammad-Bello, B., Salihu, I. A. and Suri, B. (2022) *Automatic classification of equivalent mutants in mutation testing of android applications*, *Symmetry*, Vol. 14 (4), pp. 820.
- Laricchia, F. (2023) Apple inc’s expenditure on research and development from fiscal year 2007 to 2023, <https://www.statista.com/statistics/273006/apple-expenses-for-research-and-development/>. Accessed: 20 November 2023.
- Laurent, T., Guillot, L., Toyama, M., Smith, R., Bean, D. and Ventresque, A. (2017) Towards a gamified equivalent mutants detection platform, *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*, IEEE, pp. 382–384.
- Li, Y., Shen, W., Wu, T., Chen, L., Wu, D., Zhou, Y. and Xu, B. (2022) *How higher order mutant testing performs for deep learning models: A fine-grained evaluation of test effectiveness and efficiency improved from second-order mutant-classification*



*tuples*, Information and Software Technology, Vol. 150, pp. 106954.

Lima, J. A. P. and Vergilio, S. R. (2018) Search-based higher order mutation testing: A mapping study, *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*, New York, NY, USA, Association for Computing Machinery, p. 87–96.

Lin, H.-Y., Wang, C.-Y., Chang, S.-C., Chen, Y.-C., Chou, H.-M., Huang, C.-Y., Yang, Y.-C. and Shen, C.-C. (2012) A probabilistic analysis method for functional qualification under mutation analysis, *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, pp. 147–152.

Liu, J. and Song, L. (2021) Second-order mutation testing cost reduction based on mutant clustering using som neural network model, *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, IEEE, pp. 974–979.

Ma, L., Zhang, F., Sun, J., Xue, M., Li, B., Juefei-Xu, F., Xie, C., Li, L., Liu, Y., Zhao, J. et al. (2018) Deepmutation: Mutation testing of deep learning systems, *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, pp. 100–111.

Ma, Y.-S. and Kim, S.-W. (2016) *Mutation testing cost reduction by clustering overlapped mutants*, Journal of Systems and Software, Vol. 115, pp. 18–30.

Ma, Y.-S. and Offutt, J. (2005) *Description of class mutation operators for java*, Electronics and Telecommunications Research Institute, Korea, Vol. .

Ma, Y.-S., Offutt, J. and Kwon, Y. R. (2005) *Mujava: An automated class mutation system*, Software Testing, Verification and Reliability, Vol. 15 (2), pp. 97–133.

Madeyski, L. (2008) *Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites*, Software Process: Improvement and Practice, Vol. 13 (3), pp. 281–295.

Madeyski, L., Orzeszyna, W., Torkar, R. and Jozala, M. (2014) *Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation*, IEEE Transactions on Software Engineering, Vol. 40 (1), pp. 23–42.

Mao, D., Chen, L. and Zhang, L. (2019) An extensive study on cross-project predictive mutation testing, *2019 12th IEEE Conference on Software Testing, Validation and*

*Verification (ICST)*, IEEE, pp. 160–171.

Mateo, P. R. and Usaola, M. P. (2013) *Parallel mutation testing*, *Software Testing, Verification and Reliability*, Vol. 23 (4), pp. 315–350.

Mateo, P. R., Usaola, M. P. and Alemán, J. L. F. (2012) *Validating second-order mutation at system level*, *IEEE Transactions on Software Engineering*, Vol. 39 (4), pp. 570–587.

Mateo, P. R., Usaola, M. P. and Offutt, J. (2010) Mutation at system and functional levels, *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 110–119.

Mateo, P. R., Usaola, M. P. and Offutt, J. (2013) *Mutation at the multi-class and system levels*, *Science of Computer Programming*, Vol. 78 (4), pp. 364–387.

Mathur, A. P. and Krauser, E. W. (1988) Modeling mutation on a vector processor, *Proceedings of the 10th international conference on Software engineering*, IEEE Computer Society Press, pp. 154–161.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R. and Micco, J. (2017) Taming google-scale continuous testing, *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, IEEE, pp. 233–242.

Mishra, D. B., Acharya, B., Rath, D., Gerogiannis, V. C. and Kanavos, A. (2022) *A novel real coded genetic algorithm for software mutation testing*, *Symmetry*, Vol. 14 (8), pp. 1525.

Mraz, R. T., Howe, A. E., von Mayrhauser, A. and Li, L. (1995) System testing with an ai planner, *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, IEEE, pp. 96–105.

Naeem, M. R., Lin, T., Naeem, H. and Liu, H. (2020) *A machine learning approach for classification of equivalent mutants*, *Journal of Software: Evolution and Process*, Vol. 32 (5).

Naeem, M. R., Lin, T., Naeem, H., Ullah, F. and Saeed, S. (2019) *Scalable mutation testing using predictive analysis of deep learning model*, *IEEE Access*, Vol. 7, pp. 158264–158283.

Nayyar, Z., Rafique, N., Hashmi, N., Rashid, N. and Awan, S. (2015) Analyzing test

case quality with mutation testing approach, *2015 Science and Information Conference (SAI)*, IEEE, pp. 902–905.

Nester (2002) *Nester*. [Online]. Available at: <http://nester.sourceforge.net/>. (Accessed: 01 June 2023).

Nica, S. and Wotawa, F. (2012) *Using constraints for equivalent mutant detection*, *Electronic Proceedings in Theoretical Computer Science*, Vol. .

NinjaTurtles (2014) *Ninjaturtles*. [Online]. Available at: <http://www.mutation-testing.net/>. (Accessed: 16 May 2023).

Noemmer, R. and Haas, R. (2020) An evaluation of test suite minimization techniques, *International Conference on Software Quality*, Springer, pp. 51–66.

Offutt, A. (1989) *The coupling effect: Fact or fiction*, *ACM SIGSOFT Software Engineering Notes*, Vol. 14 (8), pp. 131–140.

Offutt, A. J. (1992) *Investigations of the software testing coupling effect*, *ACM Transactions on Software Engineering and Methodology*, Vol. 1 (1), pp. 5–20.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H. and Zapf, C. (1996) *An experimental determination of sufficient mutant operators*, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 5 (2), pp. 99–118.

Offutt, A. J. and Pan, J. (1997) *Automatically detecting equivalent mutants and infeasible paths*, *Software Testing, Verification and Reliability*, Vol. 7 (3), pp. 165–192.

Offutt, A. J., Pargas, R. P., Fichter, S. V. and Khambekar, P. K. (1992) Mutation testing of software using a mimd computer, in *1992 International Conference on Parallel Processing*, Citeseer.

Offutt, A. J. and Untch, R. H. (2001) Mutation 2000: Uniting the orthogonal, *Mutation Testing for the New Century*, Springer, pp. 34–44.

Omar, E. and Ghosh, S. (2012) An exploratory study of higher order mutation testing in aspect-oriented programming, *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, IEEE, pp. 1–10.

Örgård, J. (2022) *Recommendations for mutation testing as part of a continuous integration pipeline: With a focus on c++*, Master's thesis, University of Gothenburg.

Orzeszyna, W. (2011) *Solutions to the equivalent mutants problem: A systematic*

*review and comparative experiment*, Master's thesis, Blekinge Institute of Technology.

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G. and Abraham, A. (2022) *A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools*, Engineering Applications of Artificial Intelligence, Vol. 111, pp. 104773.

Palomo-Lozano, F., Estero-Botaro, A., Medina-Bulo, I. and Núñez, M. (2018) Test suite minimization for mutation testing of ws-bpel compositions, *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1427–1434.

Panichella, A. and Liem, C. C. (2021) What are we really testing in mutation testing for machine learning? a critical reflection, *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, IEEE, pp. 66–70.

Papadakis, M., Delamaro, M. and Le Traon, Y. (2014) *Mitigating the effects of equivalent mutants with mutant classification strategies*, Science of Computer Programming, Vol. 95, pp. 298–319.

Papadakis, M., Jia, Y., Harman, M. and Le Traon, Y. (2015) Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique, *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, pp. 936–946.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y. and Harman, M. (2017) *Mutation testing advances: An analysis and survey*, Advances in Computers, Vol. .

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y. and Harman, M. (2019) Mutation testing advances: An analysis and survey, *Advances in Computers*, Vol. 112, Elsevier, pp. 275–378.

Papadakis, M. and Le Traon, Y. (2014) Effective fault localization via mutation analysis: A selective mutation approach, *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1293–1300.

Papadakis, M. and Malevris, N. (2010) An empirical evaluation of the first and second order mutation testing strategies, *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, IEEE, pp. 90–99.

Parsai, A., Murgia, A. and Demeyer, S. (2016) A model to estimate first-order mutation

coverage from higher-order mutation coverage, *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 365–373.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D. and Keller, B. (2017) Evaluating and improving fault localization, *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, pp. 609–620.

Petrović, G., Ivanković, M., Fraser, G. and Just, R. (2021) *Practical mutation testing at scale: A view from google*, *IEEE Transactions on Software Engineering*, Vol. 48 (10), pp. 3900–3912.

PexMutator (2010) *Pexmutator*. [Online]. Available at: <https://www.csc2.ncsu.edu/techreports/tech/2010/TR-2010-10.pdf>. (Accessed: 16 May 2023).

Pitts, R. (2021) Random selection might just be indomitable, *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–6.

Polo, M., Piattini, M. and García-Rodríguez, I. (2009) *Decreasing the cost of mutation testing with second-order mutants*, *Software Testing, Verification and Reliability*, Vol. 19 (2), pp. 111–131.

Rani, S., Suri, B. and Khatri, S. K. (2015) Experimental comparison of automated mutation testing tools for java, *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*, IEEE, pp. 1–6.

Schuler, D., Dallmeier, V. and Zeller, A. (2009) Efficient mutation testing by checking invariant violations, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ACM, pp. 69–80.

Schuler, D. and Zeller, A. (2009) Javalanche: efficient mutation testing for java, *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ACM, pp. 297–298.

Schuler, D. and Zeller, A. (2013) *Covering and uncovering equivalent mutants*, *Software Testing, Verification and Reliability*, Vol. 23 (5), pp. 353–374.

Sharma, V., Kumar, R. and Tyagi, S. (2016) *A review of genetic algorithm and mendelian law*, *International Journal of Scientific & Engineering Research*, Vol. 7 (12).

- Shen, W., Wan, J. and Chen, Z. (2018) Munn: Mutation analysis of neural networks, *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, pp. 108–115.
- Siami Namin, A., Andrews, J. H. and Murdoch, D. J. (2008) Sufficient mutation operators for measuring test effectiveness, *Proceedings of the 30th International Conference on Software Engineering*, ACM, pp. 351–360.
- Silva, R. A., de Souza, S. d. R. S. and de Souza, P. S. L. (2017) *A systematic review on search based mutation testing*, Information and Software Technology, Vol. 81, pp. 19–35.
- Singh, M. and Srivastava, V. M. (2017) Extended firm mutation testing: A cost reduction technique for mutation testing, *Image Information Processing (ICIIP), 2017 Fourth International Conference on*, IEEE, pp. 1–6.
- Souza, B. B. and Gheyi, R. (2020) *Most higher mutants are useless for method-level mutation operators using weak mutation*. [Online]. Available at: <http://dspace.sti.ufcg.edu.br:8080/jspui/bitstream/riufcg/20158/1/BEATRIZ%20BEZERRA%20DE%20SOUZA%20-%20TCC%20CIE%cc%82NCIA%20DA%20COMPUTAC%cc%a7A%cc%83O%202020.pdf>. (Accessed: 01 June 2023).
- Souza, F. C. M., Papadakis, M., Le Traon, Y. and Delamaro, M. E. (2016) Strong mutation-based test data generation using hill climbing, *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pp. 45–54.
- Strug, J. and Strug, B. (2012) Machine learning approach in mutation testing, *IFIP International Conference on Testing Software and Systems*, Springer, pp. 200–214.
- Stryker (2014) *Nester*. [Online]. Available at: <https://stryker-mutator.io/>. (Accessed: 16 May 2023).
- Tambon, F., Majdinasab, V., Nikanjam, A., Khomh, F. and Antonio, G. (2023) Mutation testing of deep reinforcement learning based on real faults, *International Conference of Software Testing (ICST2023)*.
- Untch, R. H., Offutt, A. J. and Harrold, M. J. (1993) Mutation analysis using mutant schemata, *ACM SIGSOFT Software Engineering Notes*, Vol. 18, ACM, pp. 139–148.
- Usaola, M. P. and Mateo, P. R. (2010) *Mutation testing cost reduction techniques: A survey*, IEEE Software, Vol. 27 (3), pp. 80–86.

- Uzunbayir, S. (2018) A genetic algorithm for the winner determination problem in combinatorial auctions, *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, IEEE, pp. 127–132.
- Uzunbayir, S. (2022) Reverse ant colony optimization for the winner determination problem in combinatorial auctions, *2022 7th International Conference on Computer Science and Engineering (UBMK)*, IEEE, pp. 19–24.
- Uzunbayir, S. and Kurtel, K. (2019) An analysis on mutation testing tools for c# programming language, *2019 4th International Conference on Computer Science and Engineering (UBMK)*, IEEE, pp. 439–443.
- Uzunbayir, S. and Kurtel, K. (2023a) *Evocolony: A hybrid approach to search-based mutation test suite reduction using genetic algorithm and ant colony optimization*, International Journal of Intelligent Systems and Applications in Engineering, Vol. 12 (1), pp. 437–449.
- Uzunbayir, S. and Kurtel, K. (2023b) *Leveraging genetic algorithms for efficient search-based higher-order mutation testing*, Computing and Informatics, Vol. 43 (3).
- Uzunbayir, S. and Kurtel, K. (2024) Mutation testing reinvented: How artificial intelligence complements classic methods. Unpublished manuscript.
- Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V. (2010) Soot: A java bytecode optimization framework, *CASCON First Decade High Impact Papers*, IBM Corp., pp. 214–224.
- Van Nho, D., Vu, N. Q. and Binh, N. T. (2019) A solution for improving the effectiveness of higher order mutation testing, *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*, IEEE, pp. 1–5.
- VisualMutator (2013) *Visualmutator*. [Online]. Available at: <https://visualmutator.github.io/web/>. (Accessed: 16 May 2023).
- von Mayrhauser, A., Scheetz, M., Dahlman, E. and Howe, A. E. (2000) Planner based error recovery testing, *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, IEEE, pp. 186–195.
- Wedyan, F., Al-Shishani, A. and Jararweh, Y. (2022) *Gasubtle: A new genetic algorithm for generating subtle higher-order mutants*, Information, Vol. 13 (7), pp. 327.
- Wong, W. E. (1993) *On mutation and data flow*, PhD thesis, Purdue University.

- Wright, C. J., Kapfhammer, G. M. and McMinn, P. (2013) Efficient mutation analysis of relational database structure using mutant schemata and parallelisation, *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, IEEE, pp. 63–72.
- Yao, X., Zhang, G., Pan, F., Gong, D. and Wei, C. (2020) *Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing*, IEEE Transactions on Software Engineering, Vol. 48 (4), pp. 1169–1184.
- Yao, Y., Liu, J., Huang, S., Hui, Z., Wu, K., Chen, L., Yang, S. and Chen, Q. (2019) Testing adequacy of convolutional neural network based on mutation testing, *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, pp. 536–537.
- Yu, M. and Ma, Y.-S. (2019) *Possibility of cost reduction by mutant clustering according to the clustering scope*, Software Testing, Verification and Reliability, Vol. 29 (1-2), pp. e1692.
- Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y. and Zhang, L. (2018) *Predictive mutation testing*, IEEE Transactions on Software Engineering, Vol. pp. 1–1.
- Zhang, L., Gligoric, M., Marinov, D. and Khurshid, S. (2013) Operator-based and random mutant selection: Better together, *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, pp. 92–102.