Using Genetic Algorithms to Automatically Generate Unit Tests

Manuel Ferreira Magalhães

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Nuno Gonçalo Coelho Costa Pombo

Covilhã, setembro de 2024

Declaração de Integridade

Eu, Manuel Ferreira Magalhães, que abaixo assino, estudante com o número de inscrição M12239 de/o Engenharia Informática da Faculdade de Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o Código de Integridades da Universidade da Beira Interior.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 09/09/2024

Assinado por: **Manuel Ferreira Magalhães**
Num. de Identificação: 15756367
Data: 2024.09.09 12:20:04+01'00'

Agradecimentos

A conclusão deste mestrado e a elaboração desta dissertação não seriam possíveis sem a presença de um conjunto de pessoas que me acompanharam ao longo desta etapa muito importante da minha vida.
Um agradecimento aos meus colegas de curso e amigos da Covilhã por todas as horas de diversão e de trabalho, todos os bons momentos e sobretudo por toda a companhia e amizade durante estes dois anos de mestrado.
Um agradecimento enorme também aos meus pais e avós por todo o apoio e sacrifício para que esta grande etapa da minha vida fosse realizada. Um obrigado também aos restantes membros da minha família por todo o incentivo.
Um agradecimento especial ao Professor Doutor Nuno Pombo por todas as oportunidades fornecidas ao longo deste mestrado bem como todo o acompanhamento demonstrado ao longo da realização desta dissertação.
Um agradecimento final à Universidade da Beira Interior pela construção da minha vida académica e desenvolvimento do meu futuro a nível profissional.

This work was supported by the "Coding-OUT - Coding in prison as a valuable OUTside tool for employment" project, funded by the European Union's Erasmus+ Programme for Education under KA2 grant.

Resumo

Testagem unitária é uma das atividades mais importantes durante um ciclo de vida de um produto de *software*. Nesta operação é possível efetuar testes para unidades individuais isoladas de um *software* de forma testar a sua funcionalidade geral sem interações externas e verificar se corresponde aos requisitos funcionais de sistema definidos inicialmente no ciclo de vida de um produto de *software*.
A criação de testes unitários pode ser atualmente feita de forma manual, com recurso a atividade humana ou automaticamente com uso de ferramentas, *frameworks*, inteligência artificial ou com algoritmos dedicados para a geração dos mesmos. A automação nesta área de geração de testes unitários revela grande potencial no que diz respeito a eficiência do processo e na redução de custos e tempo de desenvolvimento de um *software*. Esta área de estudo tem um grande impacto na qualidade de *software*, uma vez que afeta diretamente o ciclo de vida do mesmo, onde o desenvolvimento das grandes partes depende estritamente do bom desenvolvimento e testagem das unidades individuais.
Este estudo apresenta uma abordagem automática para a geração de testes unitários a partir de um pedaço de código individual através da utilização de algoritmos genéticos. Adicionalmente, são efetuados *benchmarks* entre os algoritmos para avaliação de desempenho do processo automatizado e da qualidade dos testes unitários gerados.

Palavras-chave

Testagem unitária, algoritmos genéticos, qualidade de *software*, *search-based test generation*, geração automatizada de testes unitários

Resumo Alargado

Introdução

O *software* tem uma presença importante na vida das pessoas. É utilizado num vasto espetro de áreas de investigação, comércio e entretenimento. O mercado de qualquer tipo de produto de *software* está em constante atualização para responder às novas expectativas de determinadas bases de clientes. O tempo de mercado é o período de tempo que se assenta desde a ideia inicial do produto até ao seu lançamento no mercado. O tempo de mercado para um *software* é um aspeto importante a ter em conta, por poder ser um fator decisivo para o sucesso do produto de *software*.

O ciclo de vida de um *software* consiste num conjunto de operações de desenvolvimento de um produto de *software* que se situa entre as fases de planificação e de implementação. A fase de testes é uma fase bastante importante, uma vez que determina se o que foi desenvolvido corresponde às expectativas definidas nos requisitos do produto. Nesta fase, um processo de automatização organizado e eficiente pode reduzir o tempo gasto nos testes e consecutivamente reduzir o tempo de implementação de um produto de *software*.

Este trabalho apresenta uma concetualização e aplicação de uma abordagem de geração automática de testes unitários para testar partes de um sistema de *software* isoladamente. Este projeto visa também detalhar a importância da automação na geração de testes unitários no que diz respeito à velocidade de desenvolvimento e à eficiência na cobertura global de código de um pedaço de *software*. Saber quanto código foi coberto por um teste é um aspeto importante a ter em conta, uma vez que determina se o teste conseguiu cobrir a maioria do comportamento do segmento de código testado. Adicionalmente, pretende-se que a geração automática de testes unitários seja conseguida com o auxílio de algoritmos baseados em pesquisa, utilizando algoritmos genéticos, para conseguir uma maior cobertura de código e eficiência nos testes unitários gerados.

Motivação e Âmbito

Este trabalho foi fortemente inspirado num projeto realizado na disciplina de Qualidade de *Software*, leccionada na Universidade da Beira Interior (UBI) no segundo ciclo de Engenharia Informática durante o ano letivo 2022/2023. O projeto consistiu na avaliação de uma *framework* de geração automática de testes unitários para a linguagem de programação Python. Esta *framework* empregou o uso de variações complexas de algoritmos genéticos para auxiliar a geração de testes unitários conforme o contexto de classes. Consegui publicar um artigo sobre esta *framework* na conferência internacional Institute of Electrical and Electronics Engineers (IEEE) *AITest* 2023. Recebi algumas ideias e críticas muito úteis sobre o trabalho realizado. Isto favoreceu imenso o meu interesse pela área da geração automática de testes unitários e acabou por decidir a escolha do tema para esta dissertação. Este projeto, insere-se no tema da qualidade de *software*, focando-se na forma como uma abordagem de geração automática de testes unitários pode ser benéfica, a longo prazo, para reduzir o tempo de

desenvolvimento e melhorar os testes unitários em termos de cobertura de código.

Abordagem ao Problema e Objetivos

O problema abordado por este trabalho é a falta de otimização e eficiência no desenvolvimento, nos custos e no tempo de desenvolvimento quando a geração manual de testes unitários é utilizada num ciclo de vida de um *software*. Estes aspetos conseguem ser otimizados pelo uso de um processo automatizado de geração de testes unitários usando uma geração de testes baseada em pesquisa, ou seja, criar automaticamente testes unitários para um trecho de código individual via algoritmos genéticos. Este projeto, sob o tema da qualidade de *software*, pretende atingir o seguinte conjunto de objetivos:

- **Conceituação sobre testes unitários**: uma visão organizada e focada no tema e a sua importância na qualidade de *software*, compreendendo conjuntos de definições, abordagens, propósitos, estruturas, diferenças e técnicas no âmbito dos testes unitários;
- **Impacto dos testes unitários na qualidade do *software***: como os testes são uma parte essencial do ciclo de vida do *software*, o tema da testagem unitária deve ser focado na razão pela qual esta atividade pode contribuir favoravelmente para a qualidade de um *software*;
- **Delineação da importância da geração automática de testes unitários**: uma explicação detalhada com factos fundamentados deve ser apresentada para justificar o uso desejado de automação em vez de trabalho manual nos testes unitários;
- **Explicação da geração de testes baseada em pesquisa**: como os algoritmos genéticos são um dos principais fatores para o desenvolvimento desta dissertação, deve ser feita uma introdução necessária sobre algoritmos de geração de testes baseados em pesquisa, incluindo meta-heurísticas, conceitos básicos dos fenómenos de evolução natural, operadores genéticos, investigações sobre a aplicação de algoritmos genéticos e otimizações ao processo genético geral;
- **Aplicação de algoritmos genéticos e discussão de resultados**: uma abordagem automática de geração de testes unitários com a utilização de algoritmos genéticos deve ser apresentada, incluindo a consequente análise e discussão dos resultados relativamente aos testes unitários gerados. Uma análise comparativa entre o desempenho dos algoritmos genéticos também deve ser realizada.

Contexto

O capítulo 2 explicita diversos conceitos técnicos sobre a testagem unitária, incluindo abordagens ao tema, explicações sobre a natureza isolada dos testes unitários, técnicas, bem como distinções entre abordagens e métricas de cobertura de código.
Adicionalmente, retrata com imenso detalhe a questão da geração de testes unitários, incluindo técnicas para gerar testes unitários, explicitação de algoritmos evolucionários e explicação de conceitos inerentes aos algoritmos genéticos.

Trabalhos Relacionados

O capítulo 3 explora um conjunto diversificado de trabalhos e investigações sobre o tema de geração de testes unitários. Este capítulo encontra-se dividido em duas partes. A primeira parte apresenta um conjunto de investigações sobre otimizações a aplicar nos operadores genéticos em algoritmos genéticos para melhorias de desempenho. A segunda parte demonstra alguns trabalhos já efetuados relativamente à geração automatizada de testes unitários.

Detalhes de Implementação

O capítulo 4 detalha aspetos relacionados com metodologia adotada para a implementação deste projeto. Agrupa explicações sobre a definição do problema dos testes unitários num contexto genético, o esquema de representação dos indivíduos incluindo fenótipo e genótipo, as tecnologias e bibliotecas bem como materiais necessários para a realização deste trabalho. Apresenta também a estrutura da implementação e o procedimento de execução para os algoritmos genéticos e desafios, limitações e ameaças à validade do trabalho que foram encontradas durante o desenvolvimento desta dissertação.

Discussão de Resultados

O capítulo 5 apresenta os resultados obtidos das execuções de diversos algoritmos genéticos em dois momentos diferentes. O primeiro momento refere-se à avaliação das otimizações aplicadas aos elementos constituintes de um algoritmo genético. A partir desta primeira avaliação, as melhores configurações para cada um dos atributos genéticos vão ser consideradas para novas instâncias de algoritmos genéticos. O segundo momento refere-se à utilização de algoritmos genéticos otimizados, com base nos resultados do primeiro momento de avaliação, e respetiva análise dos resultados obtidos para decidir o melhor algoritmo genético aplicado. Adicionalmente, o melhor teste unitário gerado de todas as execuções dos algoritmos genéticos será avaliado em estrutura e qualidade.

Conclusão e Trabalho Futuro

O capítulo 6 apresenta as conclusões retiradas a partir do desenvolvimento desta dissertação e trabalho futuro a implementar.
Apresenta uma revisão sobre os conceitos de testagem unitária, a importância que os testes unitários têm sobre a qualidade de *software* bem como as técnicas que podem ser utilizadas para gerar os testes unitários onde a utilização de algoritmos genéticos é valorizada. Além disso, refere também a importância dos trabalhos relacionados em melhorar o desenvolvimento deste projeto quer no desempenho dos algoritmos genéticos, quer na elaboração da geração automatizada de testes unitários. Apresenta uma análise geral dos resultados obtidos bem como principais observações da discussão dos resultados. O trabalho futuro é mencionado com algumas sugestões a implementar para um desenvolvimento mais completo e complexo do trabalho realizado nesta dissertação.

Abstract

Unit testing is one of the most important activities during the lifecycle of a software product. In this operation, it is possible to perform tests for isolated individual units of software to test their overall functionality without external interactions and to verify if they correspond to the defined system functional requirements initially set in the software product's lifecycle. The creation of unit tests can currently be done manually, with the use of human activity, or automatically, either through the use of tools, frameworks, artificial intelligence, or dedicated algorithms for their generation. Automation in this area of unit test generation shows great potential in terms of process efficiency and in reducing costs and development time. This area of study has a significant impact on software quality, since it directly affects the software lifecycle, where the development of major components depends strictly on the good development and testing of individual units.
This study presents an automatic approach for the generation of unit tests from an individual code snippet using genetic algorithms. Additionally, benchmarks are also performed between the algorithms to assess the performance of the automated process and the quality of the generated unit tests.

Keywords

Unit testing, software quality, genetic algorithms, search-based test generation, automated unit test generation

Contents

List of Figures

List of Tables

Lista de Acrónimos

EA	Evolutionary Algorithm
GA	Genetic Algorithm
PC	Path Constraint
SR	Success Rate
AAA	Arrange-Act-Assert
AES	Average Number of Evaluations to a Solution
BMI	Body Mass Index
BMR	Basal Metabolic Rate
CLI	Command Line Interface
CPU	Central Processing Unit
CUT	Class Under Test
DHC	Dynamic Decreasing of High Crossover Ratio
DHM	Dynamic Decreasing of High Mutation Ratio
ILC	Dynamic Increasing of Low Crossover Ratio
ILM	Dynamic Increasing of Low Mutation Ratio
ISO	International Organization for Standardization
MBF	Mean Best Fitness
NHS	National Health Service
OGA	Optimized Genetic Algorithm
SUT	System Under Test
TGA	Traditional Genetic Algorithm
TTM	Time to Market
RAM	Random Access Memory
REE	Resting Energy Expenditure
RLT	Remaining Lifetime
SSD	Solid-State Drive
SSH	Secure Shell
TSP	Travelling Salesman Problem
UBI	Universidade da Beira Interior
XML	Extensible Markup Language
APGA	Adaptive Population Size Genetic Algorithm
ASGA	Adaptive Selection Genetic Algorithm

HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
JSON	JavaScript Object Notation
RDEE	Resting Daily Energy Expenditure
RVPS	Random Variation of Population Size
TDEE	Total Daily Energy Expenditure
TOGA	Test Oracle GenerAtion
APOGA	Adaptive Population Pool Size Based Genetic Algorithm
SACGA	Self Adaptation of Crossover Genetic Algorithm
SAMGA	Self-Adaptive Mutation Genetic Algorithm
SAXGA	Self-Adaptive Crossover Genetic Algorithm
GAVaPS	Genetic Algorithm with Varying Population Size
PRoFIGA	Population Resizing on Fitness Improvement Genetic Algorithm
ILM/DHC	Dynamic Increasing of Low Mutation/Dynamic Decreasing of High Crossover
DHM/ILC	Dynamic Decreasing of High Mutation Ratio/Dynamic Increasing of Low Crossover Ratio

Chapter 1

Introduction

Software has a major presence in people's everyday lives. It is used in a wide spectrum of areas of research, commerce, and entertainment. The market for any type of software project is constantly updating itself in order to meet new expectations regarding certain costumer bases. A software Time to Market (TTM) is the period of time from the initial concept of the product to its release onto the market [6]. TTM can severely impact the success of the software [6]. A business that arrives too late to the market, when deploying a product, can expect a loss of about 33% of after-tax profit when deploying a product with a delay of six months, according to the article [7].

A software life cycle consists in an arrangement of operations to develop a software product that ranges from the planification stages to the deployment stages. There are a wide variety of software life cycle models that differ regarding the inherent stages of a software life cycle, however all of them agree in allocating the testing stage before the deployment one [8]. This brings attention to the testing stage, where an organized and efficient automation process can reduce the time spent in it [9] and consecutively reduce the TTM deployment of a software product.

This work provides a conceptualization and application of an automated unit test generation approach when testing software system parts in an isolated form. In accordance to the TTM argument point made earlier, this project also aims to detail the importance of automation in unit test generation regarding development speed and efficiency in overall code coverage of a piece of software. Knowing the extent of code coverage by a test is important, as this determines if the test covered the vast majority of the behavior of the segment of code under test. Additionally, the automated generation of unit tests is intended to be achieved with the aid of search-based algorithms - i.e. genetic algorithms - in order to achieve higher code coverage and efficiency within the generated unit tests.

This chapter is divided into the following sections:

- **Motivation and Scope**: reason as to why this dissertation was conceived and where it's placed in areas of research;
- **Problem Statement and Objectives**: presents the main problem tackled by this work and details a list of objectives to be achieved during the development of this dissertation;
- **Main Contributions**: lists the main contributions of this dissertation;
- **Document organization**: details the structure of this document while describing briefly the contents of each chapter.

1.1 Motivation and Scope

This work was heavily inspired by a project done in the subject of Software Quality, lectured in UBI under the second cycle degree in Computer Science and Engineering during the academic year 2022/2023. The project consisted in evaluating an automated unit test generation framework for the Python programming language. This framework employed the use of complex variations of genetic algorithms to aid the generation of unit tests according to class context. I was able to publish a paper on this framework in the IEEE AITest 2023 international conference titled "Unlocking the Potential of Dynamic Languages: An Exploration of Automated Unit Test Generation Techniques" [10] which received some very useful insights and reviews. This favored immensely my interest about the area of automated unit test generation and ultimately decided the theme for this dissertation. This project inserts itself in the software quality theme, focusing on how an automated unit test generation approach can be beneficial to, in the long term, reduce development time and improve overall unit testing in terms of code coverage, when used in a software life cycle.

1.2 Problem Statement and Objectives

The problem addressed by this work is the lack of optimization in terms of development efficiency, costs and time when performing manual creation of unit tests during a software's life cycle. These aspects can be optimized by creating an automated unit test generation process using search-based test generation techniques, i.e, automatically creating unit tests for an individual code snippet by using genetic algorithms.
This work, under the theme of software quality, aims to achieve the following set of objectives:

- **Conceptualization about unit testing**: an organized and focused view under the unit testing theme and its importance in software quality must be given, comprising sets of definitions, approaches, purposes, structures, dissimilarities and techniques within the unit test realm;

- **Impact of unit tests in software quality**: as testing is an essential part of a software life cycle, unit testing should be focused as to why it can contribute to a software's quality;

- **Delineation of importance about automated generation of unit tests**: a detailed explanation with grounded facts must be presented in order to justify the desired use of automation instead of manual work in the generation of unit tests;

- **Search-based test generation explanation**: as genetic algorithms are one of the main factors for the development of this project, a necessary introduction must be given about search-based test generation algorithms including metaheuristics, basic concepts of natural evolution phenomena, genetic operators, researches about the application of genetic algorithms and optimizations to the overall genetic process;

- **Application of genetic algorithms and results discussion**: an approach to automatically generate unit tests using genetic algorithms should be presented with consequent analysis and results discussion regarding the generated unit tests. An analysis between the performance of all genetic algorithms must also be done.

1.3 Main Contributions

This dissertation aims to provide the following set of contributions:

- In-depth study about the automated generation of unit tests;
- Propose the usage of search-based algorithms for the automated generation of unit tests;
- Expand the genetic algorithms optimization theme;
- Enhance the genetic algorithms' performance when generating unit tests;
- Present a novel approach for an automated unit test generation.

1.4 Document Organization

The present document is organized in the following chapters:

- **Chapter 2 - Background**: conceptualization of unit testing in software quality, including an explanation about the available methods and respective optimizations for the automated unit test generation process;
- **Chapter 3 - Related Work**: brief discussion about researches and applications of unit test generation algorithms and genetic algorithms;
- **Chapter 4 - Implementation Details**: thorough explanation about the adopted methodology for the implementation of this dissertation, as well as challenges and limitations associated to it. Additionally, it also includes threats to the validity of this dissertation;
- **Chapter 5 - Results Discussion**: presents this dissertation experiments with associated results, as well as a detailed explanation about the relevance of the results themselves;
- **Chapter 6 - Conclusion and Future Work**: shows the conclusions obtained from the work performed in this dissertation, including guidelines in terms of future research and development.

Chapter 2

Background

2.1 Introduction

This chapter will introduce concepts about unit testing and the automated generation of unit tests. A set of approaches, techniques, attributes and definitions will be presented for both themes. In summary, this chapter is divided into two main sections:

- **Unit Testing**: the section 2.2 presents definitions and approaches for unit testing. For each approach, unit testing will be explained in accordance while also including comparisons between them. Additionally, this section also explains specific concepts regarding unit testing which include unit test structure, unit test quality assessment, which metrics are used for code coverage in unit testing and which techniques are applied in the realm of unit testing;

- **Unit Test Generation**: the section 2.3 demonstrates a vast set of information about automating unit test generation. Includes a much necessary overview about the potential benefits of automating the generation against the manual generation, it also explains a set of techniques that can be used to generate unit tests automatically. This section also delves, with great detail, into the search-based test generation realm, where genetic algorithms are the main focus of research.

2.2 Unit Testing

Unit testing can be defined as an activity in which the main objective is to perform quick tests in small segments of code in an isolated state, separated from the rest of the constituent parts of the system. These isolated, small segments of code are considered as a System Under Test (SUT). In order for these small segments of code to be considered, as a whole, as a unit test, they must follow three requirements [1]. The unit tests must be quick to execute and need to be isolated from other tests, as well as only verify a single unit of behavior (small segment of code).

The realm of unit testing is a highly multifaceted theme, where differing perspectives and interpretations are commonplace. Amongst the various approaches, two stand out as prominent — the Classical School approach and the London School of unit testing. These approaches, while both focused on unit testing, have their own distinct variations in their understanding and handling regarding the concept of “isolation” within a unit test.

2.2.1 London School of Unit Testing

The London School approach considers the isolation of a unit from its collaborators (shared or mutable dependencies), which means if a certain class is the selected SUT and it holds a dependency on another class, then this dependency must be substituted by a test double. Only the immutable dependencies are excluded from the replacement [1]. Separating collaborators can guarantee that the behavior of the class under test won't be affected by any external influence. Figure 2.1 shows how the isolation in this approach is achieved by replacing SUT dependencies by test doubles.

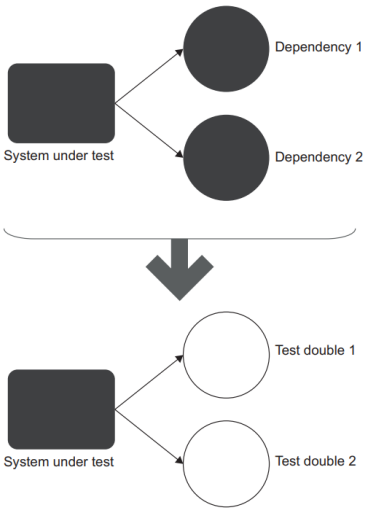


Figure 2.1: Dependencies replacement using test doubles in London School approach [1].

One of the major advantages of this approach is shown in an eventual case of failure when testing. In these situations, it's easy to identify where the failure occurred, as it happened in the SUT. The failure couldn't happen at any other place, as the SUT neighbors are test doubles. When a unit test is executed, the test double replaces the dependencies of the SUT by simpler objects. The test double is configured to provide certain inputs and outputs or to record information about how it was called by the SUT. These provide the ability to control the SUT's behavior and to verify if it occurred as expected. Test doubles can be used in a multitude of ways, it can be done manually or by using a testing library that provides this sort of functionality.

2.2.2 Classical School Approach

The Classical School approach argues that unit tests must be in an isolated state regarding the rest of the system. Additionally, each unit test must be independent of each other, so that there is no external interference in the behavior of unit tests. Unit tests can exercise a set of classes at once, however these must not reach a shared state, they must not interfere with each other. An example of shared state is an out-of-process dependency like a database, for example [1].
In cases where multiple tests are executed in parallel, the use of shared resources can lead to unexpected and undesirable behavior. For instance, two unit tests that are designed to

perform different actions on a shared database, one adding a record and the other removing a record. If these tests are run concurrently, and the second test completes before the first (removal of a record), the first test (addition of a record) may fail, not due to any issue within the test itself, but rather due to the interference caused by the second test altering the state of the shared resource. This highlights the possible dangers of shared dependencies within unit testing. Some can cause undesirable behaviors because of interference, as the unit tests are not isolated. There are three main types of dependencies that can affect unit testing:

- **Shared dependency**: dependency shared between tests, providing means for the tests to affect each other's behavior;
- **Private dependency**: dependency that isn't shared with tests, it only exists internally within the unit being tested;
- **Out-of-process dependency**: dependency that runs outside an application execution process. It can usually correspond to a shared dependency, but not always. A database, for example, is both shared and out-of-process, it runs outside the test's scope and is used by the tests to alter data. However, a read-only database, is only out-of-process because it can't be used by the tests to alter data and consecutively tests can't affect each other's behavior [1].

These dependencies are handled differently in the Classical School approach. In this approach, only shared dependencies are replaced, and private ones are maintained, as the Figure 2.2 shows.

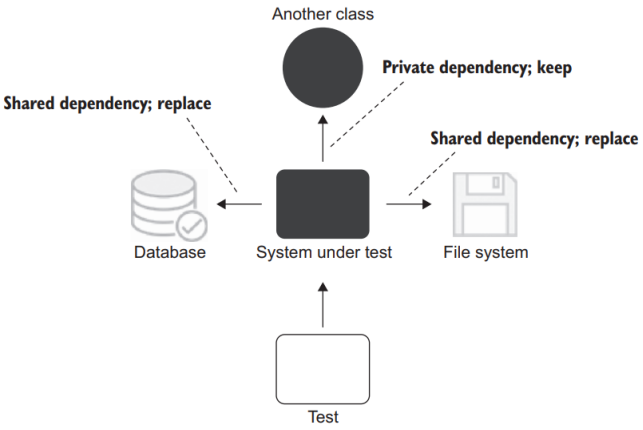


Figure 2.2: Isolation of unit tests from shared dependencies [1].

Shared dependencies are shared between unit tests, not between units. In this way of thinking, a dependency is not shared if it's possible to create a new instance of it for each test. If there is only one instance of a dependency in production code, then this latter is shared between all unit tests in the SUT as a shared dependency. Replacing shared dependencies proves to be quite useful regarding the increase of test execution speeds [1]. Calls to shared dependencies take more time than calls to the privates ones, because the shared dependencies are out of execution process of the tests. Unit testing requires the execution of quick tests, being this an important aspect taking into account.

2.2.3 Dissimilarity between Approaches

Both approaches generally agree on the definition of what a unit test is supposed to be, however they have their own differences. The root difference between the two approaches is the isolation attribute of a unit test. The Classical School approach defends the isolation must be between the unit tests themselves, while the London School approach defends that the isolation must be done for the selected SUT which represents a unit, separating the latter from its collaborators. A summary of the principal differences between each approach can be found in the Table A.1.

2.2.4 Impact in Software Quality

Software quality can be described as a field of study and practice that focuses on the desirable attributes for software products [11]. This notion of quality, in software, is heavily associated to how the characteristics of a software product can satisfy the needs of stakeholders and provide value [12]. According to the International Organization for Standardization (ISO) 25010 standard [12], a software's quality must be evaluated according to nine main characteristics being **functional suitability**, **performance efficiency**, **compatibility**, **interaction capability**, **reliability**, **security**, **maintainability**, **flexibility** and **safety**.

Unit testing has been seen as an essential element of software quality assurance, as it can determine how reliable a product can be [13]. Additionally, many developers use testing as a main method of assessing software quality [13], as it analyses the requirements, design, and architecture of the product. Unit testing provides contributions to some attributes specified in the ISO 25010 standard [12]. A few contributions are listed:

- **Fault detection**: unit testing can ensure a higher probability of detecting faults within a system while ensuring a general testing of every part that composes a system. These tests are performed to verify it the implemented functionalities are in accordance to the system requirements defined in early stages of development, contributing to **functional suitability** and **reliability** attributes [12]. In other words, it's used to verify the correctness of a certain segment of code. The identification of possible faults in the early stages of development is more important than to find them at later stages.

  The correction of a fault is minimized when found at early stages of development, where it's located at a certain unit. When a system runs and a fault occurs, it's more time-consuming and costly to correct it at a system level rather than at a unit level, where dependencies can induce complex interactions [1]. According to McConnell [2], unit testing can detect up to 50% of software defects in the development stage of a software product, when compared to another defect-detection techniques shown in Figure 2.3;

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Figure 2.3: Defect detection rates comparison between common defect-detection techniques [2].

- **Code reusability and adaptability**: unit testing can lead to code reuse. In this case, a portion of the testing can be applied again in different parts of a system using parametrized unit tests [14]. A unit test can also adapt to the system. The unit test can be updated according to the code developed for the system. In this way, tests are constantly updated to accompany the evolution of a system during the development stage. Overall, this contributes to the **maintainability** attribute;
- **Development speed**: the application of unit testing can enable a sustainable growth of a software project. In a really early stage of development, there is a low quantity of code to be worried about, enabling a rapid development and progress in the creation of the product. However, as time goes by, more code is created, more architecture ideas are developed and the probability of finding a fault is higher [1]. This situation causes the development speed to be much slower, when performed in its raw form without tests in the code, as it can be seen in Figure 2.4. This can reach points where progress is halted due to higher complexity when creating the software.

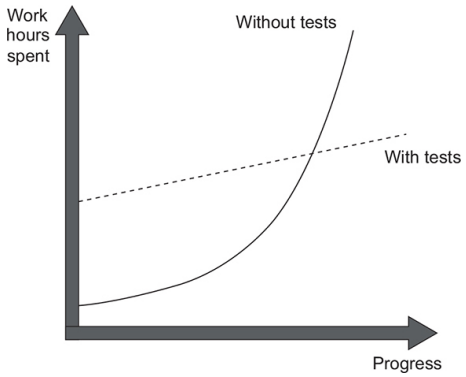


Figure 2.4: Difference in growth between projects who apply testing and do not apply testing [2].

This sudden decrease in the development speed can be designated as software entropy. Entropy is a mathematical concept that states, for a closed, independent system, the amount of disorder doesn't decrease overtime, it can only increase or be stable [15]. In software, the more changes are made, more is the disorder of the latter and its entropy increases. As the development of the software product goes on, more code is created

and the quantity of disorder, or entropy, increases [1]. As the code increases, so does the complexity of the system and higher the chance of bugs appearing. Fixing bugs in a complex system, which is composed of a great variety of functions and relationships between components, can prove to be very difficult. Changing a part of the software can affect others negatively, fixing a bug can introduce other bugs that were hidden. It is harder for a system to maintain its stability in this situation.

Unit testing can help reduce this problem. By applying tests in a specific component of the system, it is easier to verify its correctness when the software is evolving. As more code is built, more testing is applied on specific units to make sure the existing units were not affected negatively by new code or refactoring operations. This can ensure quicker development speed because a constant testing and verification in units is made as the system is evolving. Time and progress are achieved with better results when performing a constant testing during development. Without tests, more time would be spent trying to find and fix bugs as a continuous verification is not made and because of this, it's more difficult to solve errors and inconsistencies in the system. Additionally, by ensuring a consistent testing during development in later stages after the product release, the testing can be performed again for new features contributing to the **maintainability** attribute of the ISO 25010 standard.

2.2.5 Unit Test Structure

A unit test can be created following different structure patterns. One example is the Arrange-Act-Assert (AAA) pattern [1]. This pattern is divided in three parts, which are clarified below:

- **Arrange** - identifying the segment of code to be under test. In other words, a part of the system is chosen to apply tests, being this part the SUT;
- **Act** - deciding which tests to perform in the SUT. A test example would be making method or constructor calls;
- **Assert** - verifying the output result of the test performed. In this phase, the output of the test is compared to the expected output in order to determine the overall correctness. It's made an evaluation of the expected behavior, ultimately deciding if a test passes or fails.

A use example of the AAA pattern is presented in the following Listing 2.1.

```
1 public class DivisionTest {
2     @Test
3     void Divide() {
4     // Arrange section
5     var div_operation = new Division();
6
7     // Act section
8     var result = div_operation.divide(4,2);
```

```
9
10      // Assert section
11      assertEquals(2,result);
12      }
13  }
```

Listing 2.1: Using AAA pattern to create a unit test in Java programming language.

In the above example 2.1, arrange, act and assert phases are implemented. It has a class that contains a test to apply for a SUT. The unit test with the name "Divide" will test a simple division operation divided into three sections being the arrange, act and assert sections respectively. The **Arrange** section holds the SUT to be tested. In this case, the constructor of the class "Division" is called which holds division operations. This will be the SUT. The **Act** section is intended to perform an "action" on the SUT chosen. A method call of the "Division" class is performed, and the result is stored in a variable. The **Assert** section verifies the outcome from the act section. A verification will be made to see if the value of the variable from the previous section corresponds to the expected result. This determines if the test passes or fails.

Another possible pattern that is used for the creation of unit tests is the Give-When-Then pattern [1]. This one is similar to the AAA pattern and also advocates for the division of a unit test in three parts: **Given**, that describes the initial context of the test (correlates to the arrange phase); **When**, that describes the action or method being tested (correlates to the act phase) and **Then**, that describes the expected outcome (correlates to the assert phase).

The only difference between the patterns is how they structure each phase of testing, as demonstrated above. In order to design a unit test, according to the AAA pattern, the arrange section can be the first step. There is a need to specify first which section of the system will be under test before performing the act and assert phases.

2.2.6 Unit Test Quality

A unit test of good quality covers the overall functionality of the system units and is also manageable and representative of a system domain. For a unit test to be considered of high quality it must follow four attributes [1] being:

- Protection against regressions;
- Resistant to refactoring operations;
- Provide fast feedback;
- Maintainability.

A unit test can be evaluated according to these attributes, as all tests exhibit some degree of each attribute. Each of these attributes possess different particularities, being this explained in order of appearance.

2.2.6.1 Protection Against Regressions

In the scope of software, regression is when a feature stops working as intended after a certain event was performed [1]. The term regression can also be called "software bug". Regression is more likely to appear when a project has a large code base. As more functionalities are developed, higher the chances software bugs will appear and because of this a much-needed protection must be present in order to sustain a project in the long run. A test can be measured regarding its protection against regressions according to two metrics:

- **How much code is executed during the test**: higher the amount of code executed, higher the chances the test will reveal a presence of a regression. Testing large amounts of code can help determine how well the latter is protected against possible regressions;
- **Complexity of the code executed**: complex code is better than trivial code. It's rare to occur regressions in trivial code rather than complex code, as it is more simple. Simple code sometimes does not have complex elements, like libraries or frameworks for example, that increase the overall complexity of the code and can introduce a higher chance of occurrence of software bugs.

2.2.6.2 Resistance to Refactoring

This attribute refers to how much a test can sustain refactoring operations without being compromised or, in other words, keeping its functionality without failing. Refactoring is an operation that consists in changing existing code without altering its observable behavior, being normally done for code improvements [1]. In order to evaluate how resistant a test is to refactoring operations, a term must be introduced being this the **false positive** term. The lower the false positives, the higher the resistance the test has and vice versa.

When refactoring operations are made in a working system, often times, certain tests accuse failures on their execution. These failures don't accuse anything wrong with the system features, as these continue to work as intended despite the failures from the tests. The problem here is that the tests are tightly coupled to the implementation details of the SUT and with SUT changes, that were made by refactoring operations, these tests raise a false alarm. This type of situation is referred to as a false positive [1]. The test fails however the functionality it covers works as intended.

False positives often appear in refactoring situations, where a change is made in the implementation, but the observable behavior is kept intact. They can be devastating for the overall development of a project, as it can damage the perception of the overall correctness of the project. Initially, developers take test failures as a serious matter, dealing with them accordingly. However, if a constant sequence of failures is presented, developers tend to ignore them more each time and this is because of previous false alarms they had (provoked by false positives). Eventually, a released software will have a good amount of bugs, lowering the value of the latter. This is why false positives can be damaging, and the tests should not couple too much to the implementation details of the unit they are testing [1].

In order to reduce the brittleness of the tests, these need to verify the end result of the SUT and not couple too much to the implementation details within [1]. A verification of the result can provide more resistance to refactoring operations, the test is not too concerned about which steps were necessary to do something, it's concerned about the end result and if this corresponds to the expected result.

2.2.6.3 Fast Feedback

A unit test that produces a faster feedback is always more advantageous than one that does not. Faster tests enable faster feedback, which then can be used to fix bugs rapidly in case they appear. Slower tests produce a slower feedback, increasing the period of time to correct a certain bug and thus increasing overall time of correction of successive bugs that appear in the long run. Slower tests can also cause a "discouragement" effect for the developer, as these tests take too much time to run and most of them will be ignored due to the extensive time they take to execute, compromising the quality of the software in the end [1].

2.2.6.4 Maintainability

The maintainability metric corresponds to what extent a unit test can suffer modifications to be improved, modified or updated without introducing bugs or issues. A unit test that is maintainable is better than one who is not. This attribute can be divided in two important aspects:

- **Is it hard to understand the test?**: this question is correlated to the size of the test. A test with a huge amount of code is harder to read and modify. A test with a smaller amount of code should always be the priority, as it's easier to read and to modify it. The reduction of the unit test size should always be sought out without compromising its quality;
- **Is it hard to run the test?**: this question corresponds to the complexity of execution of the test. A test that is harder to run, requires more time or more effort than ones that are simpler. Tests that use out-of-process dependencies, for example, need much more effort and time to execute than one that does not. Such tests are harder to modify.

2.2.6.5 Test Accuracy

The attributes mentioned in section 2.2.6.1 and 2.2.6.2 are related to the accuracy of a unit test. Both of them contribute to the accuracy in different forms, as the main objective it's to try to maximize the latter. When speaking about test accuracy, three additional terms must be explained, as the false positive term was already introduced in section 2.2.6.2. These terms are shown in Table 2.1.

	Correct functionality	**Broken functionality**
Test result passes	Correct inference (True negative)	False negative
Test result fails	False positive	Correct inference (True positive)

Table 2.1: Test accuracy terms regarding functionality [1].

- **True negative**: a test passes and the behavior of the SUT occurs as expected;
- **True positive**: a test fails and the correspondent functionality of the SUT is broken. It's expected for the test to fail if the functionality is broken;
- **False negative**: the test passes and the functionality does not present an expected behavior. This is associated with the **protection against regressions** attribute. Tests with good protection against regressions help avoid false negatives;
- **False positive**: the test fails, but the functionality occurs as expected. This is associated with the **resistance to refactoring** attribute. Tests that have a good resistance to refactoring help avoid false positives.

The accuracy of a unit test can be measured according to the probability of false positives and false negatives. The lower the probability, the higher the accuracy of the test. As mentioned above, each of these correspond to two important attributes for a good unit test, being **protection against regressions** (false negatives) and **resistance to refactoring** (false positives). The accuracy is measured on how good the test indicates the presence of bugs (lack of false negatives corresponding to the protection against regressions attribute) and how good the test indicates the absence of bugs (lack of false positives corresponding to the resistance to refactoring attribute) [1].

2.2.6.6 Mutation analysis

The unit test ability to detect the presence of potential faults is an important aspect to consider when evaluating its quality. Mutation analysis can be used in order to determine how effective the test cases are for a given SUT [16]. This activity consists in injecting artificial defects into software, and its test cases are executed with these faults. The principal objective is to execute these tests in conjunction with their normal versions and verify if the defect itself is detected [17], i.e, if the faulty-version of the unit test provides a different outcome from its normal version.

This last sentence brings forth the concept of mutants. The mutant can be considered as the program version that contains the fault where the test suite is executed. During the mutation analysis, the mutant during this process can be considered as a `live mutant` which shows that the corresponding test suite failed to detect a fault [17]. The mutant can also be considered as a `dead mutant` which shows that the corresponding test suite detected the fault during the execution. These "detections" are all based upon the outcome of both versions, as stated above. The mutations themselves can replace relational operations, delete statements or modify conditional statements [18].

The mutation score is used to determine the percentage of killed mutants in respect to the total number of mutants. This metric is used to evaluate the effectiveness of the test cases in detecting faults. The formula to its calculation can be represented by the equation 2.1

$$Mutation\ score\ =\ (Number\ of\ killed\ mutants\ /\ Number\ of\ mutants)\ *\ 100\% \quad (2.1)$$

where the mutation score is obtained by the quotient between the number of killed mutants (detected defects) and the total number of mutants. A desired outcome, for this score, should be a high percentage of mutation score. This means almost all the mutants added were killed, or in other words, the test cases were able to identify the added faults, concluding the test was well constructed for this purpose.

2.2.7 Code Coverage Metrics

A code coverage metric can be defined as an operation to show how much code was executed. This operation is primarily used in the realm of unit testing. The main objective is to perform an assessment of quality of a test suite [19]. Coverage metric can be measured in percentage, i.e, from 0% to 100% representing percentage of code covered. In this situation, a higher value is perceived as a good indicator however, in reality, it's not that simple. A higher code coverage value does not fully represent if a code is of quality or not, it only means the test was able to cover an $x\%$ amount of code. It can be said for lower values that the test is not exercising enough code, however a higher amount cannot guarantee a good-quality unit test. There are a plethora of code coverage metrics and the most notable examples are listed as follows:

- **Line coverage**: it is one of the most used metrics to evaluate coverage within a piece of code. This metric evaluates the ratio between the number of lines of code executed by the test and the total number of lines present in the code [1];
- **Branch coverage**: this metric assesses coverage regarding branches in the code. In other words, it focuses on control structures like if statements, for example. It evaluates how many branches of code were traversed by a test. This metric only accounts for the number of branches, not taking into consideration the quantity of lines of code needed to implement them [1]. The calculation of branch coverage is obtained by the **quotient** between the **number of branches traversed** and **total number of branches** in the code. Important to mention that in order for a branch to be fully covered, it needs to be verified in all possible outcomes (true and false scenarios) [1];
- **Statement coverage**: metric used to evaluate the extent to which a test suite executes all the individual statements in the code under test. Its calculation is obtained by the **quotient** between the **number of statements executed** and **total number of statements** in the code. This metric tries to cover all possible paths, statements, and lines in the code [19].

2.2.7.1 Underlying Problems

One of the big problems associated to coverage metrics is the unreliability to determine the quality of a test suite for a SUT. Two main problems can be identified when applying these metrics [1] which are the impossibility to guarantee if a test verifies all possible outcomes and that external code paths are not covered. These two problems can be explained further by the use of the following example in Listing A.1 with corresponding explanation in the following bullet points:

- **Not all outcomes are verified**: in the example shown in listing A.1, it's possible to observe that the *lowerNumber* method, which will be executed by the test (specifically *test* method), provides two different outcomes, one implicit and one explicit. The explicit one is encoded by the return value, meanwhile the implicit one is the new value of the *lowerThanTen* variable. In spite of not verifying the implicit outcome, the coverage metrics would still show the same result regarding line and branch coverage. It would show 100% **line coverage** as all lines are executed and 50% **branch coverage** as only the **branch whose return value is false** is verified. As stated, these metrics do not guarantee that the code is tested, only that it is executed at some point and to some extent;

- **External libraries code paths are not covered**: another problem associated with these coverage metrics is the non-coverage in code paths of external libraries. When an SUT is chosen and tested, it can use certain methods that require the use of functions that are provided by external libraries. In this particular case, coverage is made only for the corresponding results (obtained by said functions) and not for the code path executed by them. In the example shown in listing A.1, more concretely for *integerToString* method, we can observe this type of situation. A 100% branch coverage is obtained by the test, verifying all components within the SUT however the hidden code paths of the Java's method for integer conversion into string is not verified. For example, three paths can be possible for the method mentioned above. The parameter *number* can follow different paths such as being a null value, being an actual integer type or even not being of that type. These kinds of paths are not covered by the tests and the values for branch coverage would change if these hidden paths were taken into account.

Although coverage metrics do not consider code paths in external libraries, this does not necessarily make them a poor evaluation method. While these paths should not be covered normally, it is important to recognize that coverage metrics alone do not provide a comprehensive measure of test quality. As the problems were stated in the above bullet points, using solely coverage metrics to evaluate a quality of a test suite is not a good approach. Coverage metrics are a good negative indicator, but a bad positive one [1]. Obtaining a lower coverage number can indicate that the test is not executing as much code as originally intended, but a higher value does not mean anything, as there could still be some inconsistencies in the implementation. Coverage metrics can provide a quantitative measure of the test coverage and identification of possible gaps. Certainly they can be used as a point to evaluate quality, however it can't be solely used to evaluate this attribute because they do not guarantee if the tests are adequate to the implementation objective.

2.2.8 Unit Testing Techniques

Unit testing operations can be performed in different ways. Testing is all about verifying the correctness of the overall units of a system either in functional or structural aspect or even both. In the following bullet points, the main techniques behind unit testing are explained

with examples:

- **Black box testing**: black box testing is a software testing method that analyzes the functionality of the software without knowing almost nothing about its internal structure [20]. It derives tests through the specification of requirements of the software to be tested, being a test driven development operation where all tests are elaborated according an expected result for a functionality of the software. One of the techniques associated with black box testing is regression testing.
- **White box testing**: white box testing is a software testing method that evaluates the code and the internal structure of a software. Emphasizes code evaluation, measuring a multitude of attributes within the structure of the latter in order to verify if it obeys to the specifications made [21]. Opposite to the black box testing method, tests are created to verify the internal structure of the code and are not made from the specification of requirements. One of the techniques associated with white box testing is mutation testing.

2.3 Unit Test Generation

In section 2.2, the fundamentals about unit testing comprising definitions, theoretical concepts, structural designs as well as unit test quality attributes and desirable characteristics, provide a much necessary conceptualization before introducing section 2.3. This section will present the main necessity and purpose behind the automated unit test generation, principles, practices and optimizations behind the automation process and explain in detail how metaheuristic algorithms can aid the search to find optimal unit tests for code units.

2.3.1 Automated vs Manual Testing

The generation of unit tests is one of the most important aspects of software development. Creating unit tests to evaluate the behavior of the different units within the system is a vital procedure to ensure the system works as intended. However, one must take into account how this procedure can be performed. Unit tests can be generated using human effort, manually, or can be generated automatically through the use of state-of-the-art unit test generation algorithms. This last sentence brings forth a relevant question: "Which of the methods should be used over the other?". This work focuses on the importance of using an automated approach against a manual one and in order to further demonstrate the advantages of automation in this theme, a brief explanation about the differences between both approaches is presented.

- **Scaling problems**: manual generation can be an exhaustive and a time-consuming process. It scales with the size of the project. Projects with a great amount of parts requires even more unit tests to be created. Manually creating these tests can hinder the development speed of the software. Automated generation, being an automated process, can help reduce the time needed to perform this process. A manual procedure tends to be more time-consuming than an automated one [9];

- **Coverage values and mutation scores**: according to studies that evaluate manual and automated test generation coverage values [9, 22, 23], automated generation of unit tests proves, in the majority of unit tests generations, to have a higher capability of achieving better coverage values than the manual approach. The ability to identify mutants in unit tests (identification of allocated defects) is generally better, with a few exceptions, in unit tests generated automatically [22];

- **Fault detection**: while automated generation tends to be superior in terms of code coverage and mutation scores, there is less consistency in favor of automated methods when it comes to fault detection [22, 24] as the manual generation seems to perform comparatively better in this regard. Automatically generated tests are not fully capable to find real faults [22]. This situation must not be misinterpreted in regard to the allocated defects in mutation testing. The identification of faults in this case refers to possible regression or errors added during the development of a project, not by changing conditional operators in method calls as it happens in mutation testing;

- **Cost efficiency**: both approaches consume human resources when practiced, only the automated approach requires an additional cost, which is machine resources. Both approaches when compared relatively to overall cost in testing time, according to the research [9], automated generation provided very low costs and overall better cost efficiency against the manual generation.

The researches [9, 22, 23, 24] focused on evaluating either automated unit test generation tools or automated generation approaches in unit testing against manually approaches using metrics such as code coverage, mutation coverage, fault detection rate, cost efficiency and execution time. All the researches incline for the importance of using an automated approach (being either by applying the approach through a tool or other technique) as in the vast majority of cases proved to be a better approach than the manual approach.

2.3.2 Automated Unit Test Generation Techniques

Unit test generation techniques have been constantly developed throughout the years. There are a plethora of different techniques to apply an automated generation as this operation itself can derive from different software artifacts such as source code, software specifications or models, program execution information and others [3]. As there are many techniques for this operation, this work will only present two techniques, aside from search-based test generation, which is the main focus of this work with the employment of genetic algorithms. Additionally, for each technique, only a brief explanation will be outlined.

2.3.2.1 Symbolic Execution

Technique that executes a software program using symbolic values instead of inputs and represent values of program variables as symbolic expressions [25]. Symbolic execution targets every feasible path within a program execution using a boolean formula entitled as *path constraint* [16, 3]. The formula accumulates a set of constraints that the inputs must satisfy

in order to follow a certain path. It follows a tree-like structure where it ramifies between satisfiable or non-satisfiable paths, the nodes represent program states and the edges represent transitions between program states [3].

The Path Constraint (PC) is updated at each branching point. If the inputs do not satisfy the PC, the path is infeasible and symbolic execution does not continue for that path. If the inputs satisfy the PC then the path is feasible and the inputs that solve the PC are considered, and the symbolic execution continues in that path until it reaches the end state of the program. A graphical representation of this process can be visualized in Figure 2.5, where it outlines the overall structure of this operation, found in the survey [3] by Anand et al. about automated unit test generation techniques. This operation is a white-box testing technique that can ensure a wide set of tests by covering every path within a program execution, by treating standard inputs of a program into a symbolic constraint solving problem.

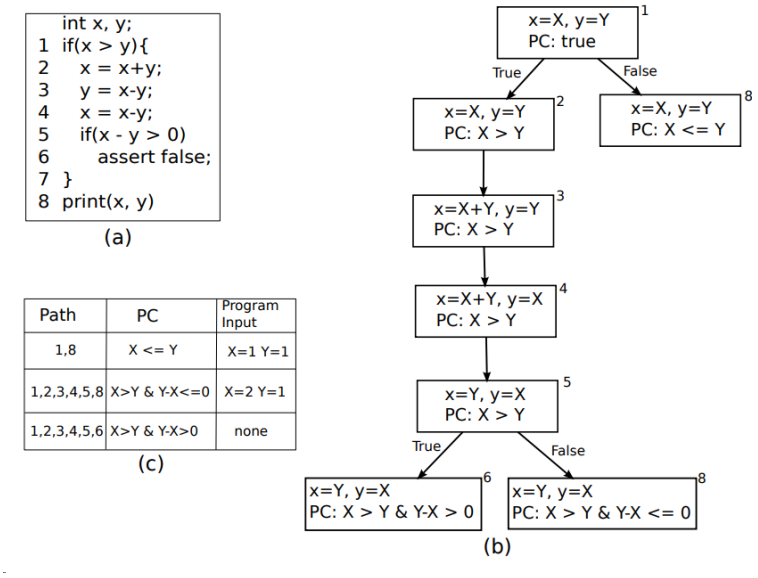


Figure 2.5: Symbolic execution structure. (a) Segment of code to swap two integers, (b) symbolic execution tree and (c) test data path, corresponding PC and program input that solves PC [3].

2.3.2.2 Model-based Testing

Model-based testing is an approach that derives test suites from models of software systems. It gathers information about the correctness of a program by applying incomplete tests [3]. It focuses on the behavior of a program and its functionality rather than its structure, being classified as a black-box testing technique. A SUT is evaluated according to its behavior, and the model defines sets of possible input/output sequences [3]. The model, after the sets are determined, is used by a test selection algorithm to derive the test cases from it, evaluating the sequences and producing feasible test cases that represent the SUT. This operation is one that derives test cases from functional requirements from the SUT rather than its structure. The area of model-based testing is one with vast variations and one of complex nature [3] as to why its information and procedures can be rather difficult to analyze.

2.3.3 Search-Based Test Generation

Test creation can be a rather tedious and intensive task to be executed manually. It requires selecting sequences of program input as well as oracles to evaluate if the given test is executing as expected. Automating this process can effectively reduce this intensive and time-consuming task, being this possible by applying an automated test creation technique called **Search-Based Test Generation**.
Search-Based Test Generation consists in a technique to seek the best test suites for a given SUT, using metaheuristic search algorithms within a restricted time limit. When generating test cases, certain goals must be met such as achieving the best code coverage, triggering assertions within the SUT or even finding possible faults. The principal objective is to generate tests to meet certain goals, as this can be considered as a search problem altogether [26]. A search is being made to find the best possible solutions (test suites) that achieve specific goals. The generation is obtained by executing a segment of code, being this process guided by cost functions, in other terms "fitness functions", to achieve the best test data possible [27]. This process is also aided with the application of metaheuristic algorithms, which provide a solution for optimization problems found within unit test generation.

2.3.3.1 Fitness Functions

Fitness functions play a crucial role in search-based test generation. These functions are responsible to determine the quality of a given test case. They evaluate the generated test cases and indicate how close they are to achieving a desired goal [28]. Fitness functions must adhere to the following requirements:

- Return continuous scores, as to offer better feedback for the metaheuristic algorithms;
- Return only numeric values in order to properly evaluate the generation of test cases each time;
- Indication of how close the generation was to being optimal. It should not indicate quality but a distance to optimal quality [28].

Fitness functions serve as a guiding principle for the metaheuristic algorithms, as they take the attributed score for the generated tests to reformulate them for the next iteration of the generation process [29].

2.3.3.1.1 Desired Goals A fitness function gives a score for each generation attempt, which can detail how close the generated tests are to achieving a specific goal. However, what is this goal concretely? When generating tests, one must ensure the tests are properly adequate. It's not entirely sure if a generated test is always relevant for the content in the SUT. An adequacy criteria must be set to ensure the test represents the necessary testing rationale for the SUT. Common methods to measure adequacy are coverage of structural elements of the software including executions of statements, branches flow and boolean conditional statements. These were already previously mentioned in section 2.2.7, where different

metrics for code coverage - such as branch, statement or conditional coverage - can prove to be a good adequacy criteria for the fitness function. For example, a fitness function can return a score for a generated test given its fulfillment of an adequacy criteria (coverage metric). This can guide the generation process, providing enough detail to help differentiate candidate solutions and always aim for optimal candidate solutions [29].

2.3.3.2 Metaheuristic Algorithms

Metaheuristic algorithms implement a search procedure to find the best solution possible within a search space while also obeying a restrict time limit or search budget. They are also a powerful aid to ensure the search of a near-optimal solution with the available resources [30]. The objective - behind these algorithms - is to find new strategies to resolve a problem, as they use heuristics for that matter.

Applying metaheuristic algorithms in conjunction with fitness functions can help make a stable and efficient search for near-optimal solutions in the context of search-based test generation. The process of generating test cases can be seen as an optimization problem, where the objective is to search within a space of possible inputs and improve the quality of the solutions over time. The use of fitness functions in this optimization process is crucial. Fitness functions assign scores to individual test cases based on their adherence to specific adequacy criteria, such as coverage metrics. These scores guide the metaheuristic algorithms to explore the search space and find better solutions with each iteration.

By leveraging the power of metaheuristic algorithms and fitness functions, search-based test generation aims to continuously improve the quality of generated test cases within the constraints of a given search budget. The process involves iteratively evaluating and refining the solutions based on their fitness scores, ultimately leading to test suites that better achieve a specific goal. A few examples of metaheuristic algorithms include the hill climber and genetic algorithms. The former is a metaheuristic algorithm that focuses in searching a local optimal solution within a search space. The algorithm starts with a single initial solution chosen randomly and makes continuous searches around its neighborhood to find better solutions. If a better solution is found - within that neighborhood - then the current solution is replaced by the better solution. This last process is repeated continuously until the search budget is reached or if no improved neighbors can be found. It is considered as a single-solution based metaheuristic [31]. The latter, being one of the main focus of this dissertation, is explained in detail in section 2.3.3.3.

2.3.3.3 Genetic Algorithm

Not all metaheuristic algorithms delve onto the local search scope, as some of them also perform a global search and consider a wide set of candidate solutions within the search space. These algorithms are called “Population-based metaheuristics” and can maintain diversity in the population while also prevent the search to be stuck in a local optima [31]. A few examples of these algorithms are particle swarm optimization, Genetic Algorithm (GA) [32] and ant colony optimization.

This work aims to use genetic algorithms to aid the automated generation of unit tests

in software, and for this, a much-needed contextualization about evolutionary algorithms must be presented. An evolutionary algorithm is an algorithm who uses a search strategy to evolve candidate solutions using genetics and natural selection inspired operators [33]. A more detailed definition can be defined as: "given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection (survival of the fittest)" mentioned by A.E. Eiben and J.E. Smith [5].
Evolutionary Algorithm (EA)s are inspired by the Darwinian evolution, where the solutions are identified as individual organisms in a population (set of solutions). Within that population, each individual is tested for fitness (how well they can resolve a problem) and a set of them are selected for reproduction, which includes crossover and mutation operators. Genetic algorithm is a class of the evolutionary algorithms and as such also employs bio-inspired operators for its generation of solutions.

2.3.3.3.1 Exploration versus Exploitation The search procedure behind these algorithms can be split in exploration and exploitation phases. Initially, the search must focus in **exploring different regions of the search space** to find feasible solutions, being this identified as an **exploration** phase [5]. Additionally, when good solutions are found, more of **these solutions must be sought after around their vicinity**, i.e., more search should be made in the region where the good solutions were found being this an **exploitation** phase [5]. However, neither of these phases must have a higher impact than the other because it can lead to undesired outcomes. A higher exploration can hinder the performance of the algorithm as it can induce an inefficient search and a higher exploitation can lead to premature convergence where a good solution is found too quickly which causes the search to get stuck in a region where no improvement can be made (better solutions can exist) not enabling a proper search of the search space [5]. These two phases must be complemented between each other and balanced, as it can prevent the algorithm to be stuck in local optima and provide a higher chance for a global optima to be found within the search space.

2.3.3.3.2 Natural Phenomena A genetic algorithm is an optimization algorithm that uses bio-inspired operators for its generation of solutions. Being a population-based search algorithm, it employs the concept of survival of the fittest [32] where only the individuals with the highest fitness scores survive. New populations are formed iteratively through genetic operators applied on the individuals of the present population. These genetic operators will alter the fittest individuals of the population and will apply crossover and mutation operations to them in order to generate even better individuals. This process is repeated until a certain budget is met and the population will contain the best generated individuals until then.
The factor that determines the selection of individuals relies upon the objective that targets the value of an evaluation function (fitness function), i.e., if it's targeted towards maximization or minimization of the function. An example of a target for maximization for a fitness function can be presented in the following equation 2.2 [34]

$$f(x^*) \geq f(x); \forall x \in D \tag{2.2}$$

where the objective is to find an individual x^* in the finite set D (population) between all the individuals present in the set which maximizes the function $f(x)$. For minimization problems, the thought process would be reversed, where the individual x^* which minimizes the function $f(x)$ in the set D is sought after.
The application of GA, with biological terms, can be explained in a simple six-step scheme:

1. **Population**: a set of chromosomes, where each one of them represents one individual, is initialized randomly;

2. **Selection**: operation that selects the fittest individuals of the current population. The chosen individuals (chromosomes) are selected through their fitness score value;

3. **Recombination**: the chosen chromosomes undergo crossover operations, i.e., they exchange information between each other according to the type of crossover specified. The newly created chromosomes are now referred to as "offsprings";

4. **Mutation**: according to the type of mutation operator, small changes will be made to the offsprings in order to add more evolution/information;

5. **Adding offsprings**: after the mutation process, the offsprings are placed in a new population, replacing the old population entirely. After this step, the search will continue in the new population;

6. **Cycle between steps two and five**: the steps two through five are repeated until a certain limit, budget or objective is reached within the generation process.

2.3.3.3.3 Individuals Representation In natural evolution terms, an individual or chromosome components are called genes, the values for each component are referred to as alleles and their position within the sequence of the chromosome is called locus. In this work, these biological terms will be referred to interchangeably between future sections.
Before applying a genetic algorithm to a domain problem, it is necessary to decide how the components of the domain (more accurately, the solutions) will be represented as. In this case, the chromosomes (solutions) must be stored and manipulated in a way that a computer can understand it. The objects that **encompass possible solutions within the problem context** are called as **phenotypes** and their **respective encoding form** are called **genotypes**. This first step is called representation, where mapping from phenotypes onto a set of genotypes is necessary for proper representation of the solutions as the search is conducted in the genotype space [5].
A type of encoding is necessary for the individuals, where for problems involving genetic algorithms, a binary encoding is often recommended [31]. However, it's worth noting that there are other encoding schemes that can be used in genetic algorithms, including octal, hexadecimal, permutation, value-based, and tree encodings [35]. These are the standard encoding schemes used for genetic algorithms, however encoding schemes are dependent on the problem domain and not all the mentioned encoding schemes may work for more specific problems. In a binary encoding scheme, a chromosome is represented as a string of values 1 or

0 (binary string) and in this encoding each bit represents the characteristics of the solution. Using a binary encoding provides a faster implementation of crossover and mutation operators [31] as each allele is one of two values, providing simple operations.

2.3.3.3.4 Selection Types After a proper representation of the individuals, a few of them must be selected to start reproduction. This selection is made according to the fitness score value of the individuals - in a population - according to how well they achieve a defined objective. In this process, the n fittest chromosomes will be chosen to act as parents for the next stage of the algorithm (recombination). This selection process is typically randomized, with the probability of selection depending on the fitness of the individuals [36]. The higher the fitness of the individual (when the evaluation function is targeted for maximization), the higher the chance he will be picked as a parent for reproduction.
There are two major categories of methods for using fitness in the selection operation:

- **Deterministic methods**: these methods involve selecting n-fittest individuals of a population. Only the individuals with the highest fitness scores are chosen for reproduction. This can lead the population to reach a local maximum, which consecutively stops the evolution process;
- **Stochastic methods**: as one of the major objectives of genetic algorithms is to provide a global search within the population, stochastic methods are often recommended to avoid the population to get stuck in local optima. These methods select individuals randomly with a probability according to fitness values [36].

A list of the principal selection types used in traditional genetic algorithms is presented.

- **Random selection**: the chromosomes are selected randomly, within the population, without any analysis on the fitness scores of the individuals. This in on itself is a huge problem for proper generation of unit tests that are representative of the problem context, because random selection of the parent chromosomes generally does not provide an adequate generation in the later stages of the genetic algorithm [36];
- **Roulette Wheel selection**: one of the traditional genetic algorithm techniques where an individual is selected, from a population, with a probability proportional to the fitness score [36] being it considered as a stochastic method. The roulette wheel selection can be described, in simple terms, as a wheel divided in sequential spaces where each of these spaces or slots are proportional to the fitness score of each individual within the population. This technique promotes a linear search through the wheel (population) defining a target value to be achieved. This target value corresponds to a random proportion of the sum of the fitness values of all individuals in the population. The population is searched until the target value is reached. This however constitutes a potential problem:
  - It's not guaranteed that fit individuals are selected despite having bigger chances (higher fitness scores). If a fit individual does not exceed the target value, there

might be a chance that the next individual in line can exceed the target value while being weaker than the previous one. In this technique, the population must not be sorted by fitness scores in order to prevent bias in the selection [36];

– Another problem associated to this technique is that more predominant individuals (with high fitness scores) will introduce bias to the initial search as they occupy a major portion of the wheel. This leads to premature convergence and loss of diversity as other individuals have low chances to be selected. This situation can hinder the search process as it almost limits the search to only one individual in the population, being this almost compared to the situation of local optima in single-solution based metaheuristics.

A pseudocode for the algorithm behind roulette wheel selection is presented in Algorithm 1;

- **Rank selection**: a ranking operation is made for each chromosome within the population. Each individual receives its rank according to the fitness value that each one has. The worst individual will have rank 1 while the best individual will have rank N whereas N refers to the number of chromosomes within the population.

  In this technique, the individuals in a population are initially ordered by their fitness scores in ascending order of fitness. After the ordering, each respective individual is assigned a rank starting from the initial position of the ordered values. The rank is assigned from 1 (worst individual with the lowest fitness score) to N (the best individual with the highest fitness score). After the rank assignment, the rank selection technique determines the selection using the ranks and not the fitness scores themselves. A pseudocode for the ranking selection algorithm, with selection of two parents, can be visualized in Algorithm 2;

- **Tournament selection**: this selection method is a variation of rank-based selection approaches, as it consists in a random selection of k individuals from a population (with or without replacement). This is made to select the fittest individual out of the k individuals selected in order to become a parent for the reproduction process. This process can be repeated N times until the required number of parents are selected and it does not apply the sorting of fitness as the base rank selection method does [33].

  This technique vastly differs from the previous ones regarding the selection operation when it comes to the size of the population. As this method selects k individuals from a population for the tournaments, a global knowledge of the population is not necessary [5]. However, **the quantity of individuals in each tournament** (tournament size) is **an important parameter to evaluate**. The larger the tournament size, the greater are the chances to obtain better fit individuals, the smaller the tournament size, the greater the chances to obtain less fit individuals. As the tournament size increases, higher the probability of high-fitness individuals to be selected and lower the probability of low-fitness individuals to be selected [5]. Standard used tournament sizes are 2, 4 and 7 [37].

The sampling from the population can be done with or without replacement. This is of great importance as it can bring some inconsistencies during the selection process as the same individual can be sampled from the population multiple times for the tournament, being this problem referred to as a multi-sampled issue [38]. The sampling issue of a tournament with replacement can be easily solved by not replacing any chosen individuals for the tournament, i.e., not returning individuals to the population that were chosen for the tournament until the parent is chosen. A pseudocode for the tournament selection algorithm is presented in Algorithm 3. The variable k in the pseudocode is referred to as **tournament size**, where a predetermined number of individuals is chosen to represent the number of contestants in a tournament selection round.

2.3.3.3.5 Recombination Operators After the parents are selected, the next phase of the genetic algorithm consists in exchanging information between the selected individuals in order to generate offsprings. This phase is designated as **recombination**, where a new individual is created from information of two or more parents [5]. This introduces genetic diversity, as genes will be exchanged between individuals to create new chromosomes with different information from their parents. While the term "recombination" is more commonly used to refer to the exchange of information between parents, the term **"crossover"** can also be used for this phase. The recombination process is one of a probabilistic nature. This process is applied according to a crossover probability (p_c). This probability decides whether the crossover happens between two parents selected from the population. This characteristic does have some implications according to the selected probability value:

- A higher probability does apply crossover operations between parents more often. However, this can lead to a faster convergence to an optimal solution which, in certain situations, cannot cover a major portion of the individuals in the population resulting in a loss of diversity in the solutions;
- A lower probability results in a lower rate of crossover operations, leading to the offspring being similar to their parents. Although this slower convergence may delay the attainment of a solution, it can also limit the acquisition of fitter individuals compared to the initial population. In terms of the crossover probability p_c, the likelihood of a crossover not occurring is $1 - p_c$.

A multitude of different crossover operators apply different strategies for the recombination process in genetic algorithms. The standard crossover operators used in genetic algorithms applications are:

- **One-Point crossover**: exchange of genes between parents occurs after a specific position within the chromosome sequence. The selection of the split position is determined randomly, choosing one gene within the range from the second to the penultimate position in the chromosome sequence. In this type of crossover, the head and tail of one chromosome cannot be passed onto the offspring and if they contain good genetic information this can be a drawback where the offspring can lack good genetic information

[36]. In this operator, after the splitting position has been decided, the succeeding genes are exchanged. Figure A.1 demonstrates an example of the One-Point crossover between two chromosomes;

- **N-Point crossover**: similar to the one-point crossover operator, however, n splitting points can be chosen to exchange the genes. The genes situated between the splitting positions are exchanged. This type of crossover enables the passage of information from both the head and tail, of a chromosome sequence, to the offspring, unlike the one-point crossover. Figure A.2 demonstrates an example of the N-Point crossover between two chromosomes;

- **Uniform crossover**: in this operator, each gene is treated independently, i.e, there is no exchange based on sequences of contiguous genes. This process makes a random choice to select the genes to exchange between parents.

  For the length of the chromosome sequence, a list of random values in a uniform distribution $[0, 1]$ is made and a parameter p is chosen to permit the exchange (usually 0.5). For each gene, its corresponding value in the list will be compared to the parameter created. If the value is below the parameter p, the gene is inherited from the first parent. In the opposite situation, the gene is inherited from the second parent. A second offspring is created according to the inverse mapping of the first offspring created by this process [5]. The Figure A.3 shows the uniform crossover for the list $[0.3, 0.6, 0.1, 0.4, 0.8, 0.7, 0.3, 0.5, 0.3]$ and $p = 0.5$.

2.3.3.3.6 Mutation Operators Mutation is the next phase after the recombination process. Mutation is a variation operator that adds more diversity into the offsprings generated from the recombination by adding new information to them. Adding new information consists in altering a gene within the chromosome sequence in order to generate an individual with new information. This operation prevents the genetic algorithm to enter a local minimum and recovers any genetic information lost from the recombination process. Mutation is viewed as a process that maintains the genetic diversity [36] as it introduces new information in the population by randomly changing genes.
The mutation parameter is crucial to consider, as varying mutation rates can have a significant impact on the generation of fit individuals. A lower mutation rate is often recommended for populations who contain a high proportion of high fitness individuals, as to not disrupt the quality of the population [5]. Conversely, where only a small percentage of the population is composed of fit individuals, a higher mutation rate is recommended to introduce more diversity [5]. Mutation operations are representation dependent (different encoding schemes have different ways to apply mutation). The standard mutation operators used in genetic algorithms applications are:

- **Bitwise**: this mutation operator consists in changing a gene value to its inverse. This is typically associated with bit flipping, changing values from within the interval of integer values $[0, 1]$. In this method, each gene has a probability p_m for its value to be changed (bit flip of a gene). The number of values to be changed is not fixed, as it is a

random choice in the interval of integer values $[1, L]$, where L represents the length of the chromosome sequence undergoing mutation. On average, $L \times p_m$ genes will undergo changes within the chromosome sequence [5]. Table A.2 demonstrates mutation flipping in chromosomes;

- **Interchanging**: two random genes in the chromosome sequence are chosen and interchanged. Two random genes are chosen from the interval of integer values $[1, L]$ (the genes must be different from each other), occurring an interchange between those genes. Table A.3 shows how the interchanging mutation works;

- **Reversing**: a random gene is chosen in the chromosome sequence and all the locus after it are reversed between each other. For the interval of integer values $[1, L]$, a random number n is chosen and all the values in the interval of $(n, L]$ (whose values refer to the loci of the chromosome) have their corresponding genes reversed between each other. This process is represented in Table A.4.

2.3.3.3.7 Termination Criteria After the mutation process occurs, the new generated individuals will be placed in a new population. This new population will undergo the same evolution process mentioned in the earlier sections, comprising the selection, crossover and mutation methods. However, this evolution cannot go on forever as it would cost a great amount of computational resources as well it would not be efficient for the generation process itself as a convergence point needs to be reached. If the algorithm reaches a point where the fittest individual is found and there is no margin for more evolution, then the algorithm must stop generating individuals. This situation brings forth the question: "When does the generation stop?" and for this situation, a stopping criteria must be defined for the algorithm. Stopping criteria enable the algorithm to achieve definitive outcomes while conserving computational resources and minimizing execution times. By carefully defining these criteria, the algorithm can strike a balance between thorough exploration and efficiency. Some standard stopping criteria used for genetic algorithms are listed below:

- **Maximum number of generations**: the algorithm continues the generation process until it reaches a maximum number of generations/iterations where the best seen solution, until then, is taken as the optimal one [34];

- **Time budget**: after a certain time limit has elapsed, the generation process stops and the best seen solution is taken as the best one. This time can be measured as absolute time or Central Processing Unit (CPU) time [36];

- **Stagnation of fitness score**: if the overall fitness score of the generation process does not improve for a specified number of generations, the execution stops and the best seen individual until then is obtained [36] [5];

- **Maximum number of objective function evaluations**: the algorithm stops after reaching a maximum number of objective function evaluations [5].

2.3.3.4 Genetic Algorithms Parameters Optimization

Parameter optimization is a crucial aspect in genetic algorithms. These parameters dictate if a genetic algorithm can provide an optimal or near-optimal solution to a given problem, as they strongly affect the search process during the execution. The mentioned parameters are population size, selection probability, crossover and mutation rates. These parameters drive the search process of the genetic algorithm and reflect a major part in the **exploration versus exploitation** phases of the latter [39]. They can provide stronger or weaker effects to each of the phases and guide the search to an optimal or a suboptimal solution.
This topic is an important area of research in genetic algorithms and it has been known as **"parameter setting"** [4]. One of the first works that delved upon the importance of adjusting these parameters is the work by Eiben, Hinterding and Michalewicz [4] where the authors classify two major forms of setting parameters values in genetic algorithms. These forms are **parameter tuning** and **parameter control**, where the first one consists in **finding appropriate values for the parameters before the run** of the algorithm, **keeping them fixed during the run** and the second one consists in **changing the parameter values during the run** of the algorithm. For parameter control, three different techniques can be applied:

- **Deterministic parameter control**: the parameter is adjusted according to a predetermined rule. This rule alters the parameter value without using any guidance from the search process, being an independent and deterministic method;
- **Adaptive parameter control**: feedback is obtained from the search process during the run, being it used to modify the parameter accordingly;
- **Self-adaptive parameter control**: the parameters are included in the chromosomes representation and undergo the same evolution process. In this way, this technique is often called "evolution of evolution" [39].

A graphical representation of the general view behind parameter setting, including the different methods for parameter optimization, can be visualized in Figure 2.6.

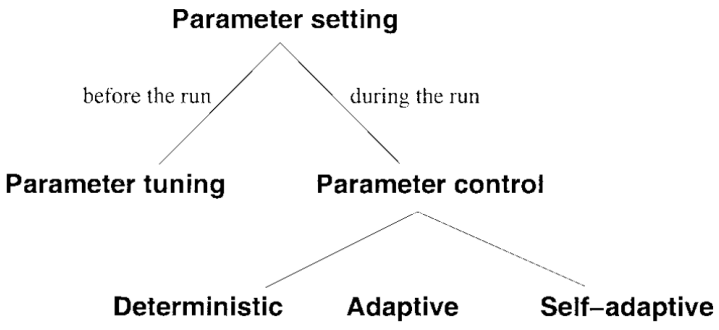


Figure 2.6: General view regarding parameter setting in EA's [4].

Parameter tuning and parameter control despite having the same objective, they largely differ in terms of potential benefits. A major drawback behind the parameter tuning process

is the lack of an adaptive mechanism to guide the parameter optimization. The optimization in this method often relies on the manual modification of values after several runs of the genetic algorithm (parameter tuning based on experimentation). Trying all possible values is practically impossible, as trying all combinations is an "aimless" approach, as well as being a time-wasting process. Not even a vast majority of the values are guaranteed to be appropriate for a given search problem. The same can be said about the deterministic parameter control. Even a deterministic rule can't determine the best guidance, as the search progress is not being analyzed [39]. This can hinder the search process, guiding it to suboptimal solutions that do not provide a good enough result for the given search problem. Adaptive and self-adaptive methods, however, can correct these guidance flaws, as taking account several aspects of the search process can enable a much more stable guidance to optimal solutions than one that follows a rule or its constantly altering parameter values in hopes of finding optimal or near-optimal solutions.
The following sections 2.3.3.4.1, 2.3.3.4.2, 2.3.3.4.3 will introduce a brief summary about each parameter importance, its drawbacks when using standard or conventional values and respective necessity for optimization.

2.3.3.4.1 Population Size Optimization A small population size can induce a quicker search for a good solution, however the chances associated with the search being stuck in a local optimum are much higher with a reduced population size despite requiring less computational power. A higher population size can prevent the search to be stuck in local optima situations. This gives more room for the search to find a global optima, however more computational resources are needed as more evaluations of individuals are made for bigger populations. Given the problems according to the sizes of the population, an improvement must be sought after. By convention, the population size must be a fixed value until the generation ends. However, a better implementation would be the dynamic control of population size during the generation process.

2.3.3.4.2 Selection Optimization Another major important part of the genetic algorithm is the selection process where appropriate parents are chosen in order to enable the generation of better individuals to include diversity into the population. Premature convergence is a serious problem to consider when selecting parents for reproduction. A local optimum can be found rather quickly during the search process, however, this only allows finding a suboptimal solution. This is not desired, as the optimal solution (global optimum) must be always sought after. Convergence of the population is often associated with the concept of **selection pressure**.
Selection pressure can be defined as how much the better individuals, in the population, are considered to be selected [36]. A higher selection pressure means that the fittest individuals are favored, on the contrary, a lower selection pressure means that these fittest individuals are not favored as much. A higher selection pressure results in a higher convergence of the population to an optimal solution, while a lower selection pressure results in a low convergence. This concept is what drives the genetic algorithm to find optimal solutions during the search process [36]. Ideally, **selection pressure should be low in the early stages of a**

genetic algorithm in order to allow a wide search of the search space to evaluate almost all possible solutions (incentive on an exploration phase). As the generation process advances, **selection pressure should gradually be higher** to find the global optimum space and then proceed to search solely on that solution vicinity to decide on the best solution possible (incentive on an exploitation phase). The convergence should be lower in the beginning phase and should rise towards the end to find the optimal solution [40].

2.3.3.4.3 Crossover and Mutation Optimization Crossover and mutation are necessary operations to introduce new variety into the population. Exchange of information between selected parents (crossover) happens to obtain a new individual being it followed by an information change operation (mutation) within the chromosome genes to introduce even more diversity into the individual where both operations happen in this specific order (these do not happen simultaneously). Both of these operations have a probability to occur during the run of the genetic algorithm and are a vital aspect for the efficiency of the algorithm as it aids the search process to find optimal solutions. Without these operations, only parents would be selected and no new individual would be created, leading the search to be useless. These operators can have a huge impact on the search process according to their probability of execution:

- **Crossover rate**: higher crossover rates can introduce new individuals into the population more frequently however if it is too high, individuals with high-fitness (that were selected previously) can lose good genes with the crossover operation [41]. Lower crossover rates do not create new individuals and the search might stagnate early on;
- **Mutation rate**: higher mutation rate introduces new information into the individuals however if it is too high, it induces a random search [41] potentially creating worse individuals and affecting existing individuals with high fitness. Lower mutation rates maintain the genetic information within the individual. In situations where the individual, after the crossover operation, did not present any major improvement in fitness, the lower mutation rate will maintain that genetic information and consecutively not add any type of variety into the population.

2.3.3.5 Genetic Algorithm Quality

The performance of a genetic algorithm is essentially based upon the quality of the solution found in the search and how quick the algorithm achieved the desired solution [5]. The time needed for the algorithm to achieve a solution can be measured in CPU or wall-clock time. Additionally, the time itself can also be measured in terms of average number of generations needed by the algorithm to achieve the solution [5]. The notion of convergence speed in a genetic algorithm can primarily take these two forms. The notion of solution quality is associated to the fitness function used in the genetic algorithm. This function is what determines how close a solution is to achieve a desired goal [28] as previously mentioned in section 2.3.3.1. There are three main combinations of solution quality and convergence time used to evaluate a genetic algorithm performance [5]:

- **Maximum running time**: performance is reflected upon the quality of the final solution (fitness value) during the time interval of execution;
- **Maximum running time and minimum fitness**: performance is evaluated according to a minimum fitness value during a limited time of execution. The execution is successful if the minimum fitness value is obtained during that time interval;
- **Minimum fitness**: performance is determined according to the execution time needed to reach this fitness value.

In cases of maximization problems, the objective is to reach the highest fitness value possible, so the fitness criteria above changes according to the problem's nature.
The performance of a genetic algorithm is only properly evaluated when multiple executions of the algorithms with same parameter values (on the same problem) and statistical measures of each individual execution are carried out [5]. In summary, two performance measures can be defined in evolutionary computing:

- **Efficiency**: speed of the algorithm to achieve a desired solution. A metric used to evaluate efficiency is Average Number of Evaluations to a Solution (AES);
- **Effectiveness**: how good the solution is according to the problem context. The metrics that evaluate the effectiveness of the genetic algorithm are Mean Best Fitness (MBF) and Success Rate (SR).

Each measure was explained briefly in terms of principal objective, however one must take into account that each of them have different conditions in order to be applied:

- The SR measure demonstrates the **percentage of algorithm runs where the desired solution was obtained**. It is calculated by the quotient between the number of successful runs and the total number of runs of the genetic algorithm [42]. This measure can only be applied to problems where the optimal solution or value can be defined beforehand. This measure cannot be applied for problems that do not have a desired solution [5];
- The AES measure determines **how many steps were required during the algorithm execution to find a solution**. The operations considered as steps in the AES measure include any type of operation that performs a fitness evaluation [5], that is, the number of newly generated solutions in successful runs [43] which groups the principal genetic operators (crossover and mutation). This performance measure has its limitations:

    1. Some executions can take more time than others because of the stochastic nature of evolutionary algorithms. There could be different values for AES on multiple instances of a genetic algorithm, with the same parameters or configurations, that are not caused by the implementation details themselves;

2. Genetic operators can have additional search procedures implemented in their structure [5]. These additional features are invisible to the AES measure, as they only count the application of the genetic operator as one step, without considering additional procedures for the search;
3. Comparing evolutionary algorithms that use different search spaces or genetic operators, essentially of different natures, is a huge problem for this performance measure [5]. Algorithms of different natures and procedures won't have similar steps and consecutively the number of evaluations will largely differ.

There is an alternative to AES considering the limitations stated above, which is to measure the progress speed of the genetic algorithm, plotting the best, average or worst fitness value against a time axis [5].

- The MBF measure is used to evaluate the **mean best fitness out of a set of genetic algorithm executions**. The best fitness is recorded for each genetic algorithm execution when the latter reaches a termination condition. Out of all the measures mentioned previously, this measure is always valid to use in genetic algorithms [5] despite the actual usefulness of it may differ in different problems. For problems that define the quantity of unsatisfied clauses as a fitness measure, knowing the MBF after execution won't say almost anything about the end objective. In this particular case, SR might be more appropriate to apply, as it refers specifically to the main objective.

The performance measures can be used in conjunction to determine an algorithm performance. For example, the SR and MBF performance measures can be used together, however the meaning of the results also changes. Low SR and a high MBF indicate that the algorithm is of an approximation nature [5], it gets close to the solution but majority of the times it cannot reach it. High SR and low MBF indicates that the algorithm, in situations where the search is nowhere near the solution, can present terrible results.
The time needed for a genetic algorithm to find a solution is also an important aspect to take into account when talking about performance measurement. In this case, the subject is more "delicate". Measuring execution time is a subjective matter as elements like hardware, operating systems, programming language and compilers largely affect the performance of the genetic algorithm [42].
Additionally, another way to measure a genetic algorithm performance is by evaluating its robustness. There is no clear definition for robustness in the literature [5, 42] however a genetic algorithm robustness can be considered as the deviation behavior of the algorithm across multiple runs on the same problem [42]. A **robust algorithm** is one that **presents low variance in its behavior across multiple runs**, while an **unrobust one presents a high variance**.

2.4 Conclusion

This chapter 2 presented a much-needed conceptualization about the unit testing realm and its automated generation possibilities. A unit test structure, the code coverage metrics im-

pact and the search-based test generation techniques are fundamental concepts regarding the theme of this dissertation. Additionally, the possibility of optimization of genetic algorithms operators is also an important factor to understand, as it can induce better performance in a genetic algorithm execution and consecutively potentially create even better and representative unit tests.

Chapter 3

Related Work

3.1 Introduction

This chapter presents a wide set of works and researches on the automated generation of unit tests and genetic algorithm realms. This chapter is composed of two main sections:

- **Employment of Genetic Algorithms**: section 3.2 shows a large set of researches that delve upon genetic operators optimizations in genetic algorithms;
- **Automated Unit Test Generation**: section 3.3 demonstrates major works behind the automated unit test generation theme.

3.2 Employment of Genetic Algorithms

As this work delves upon the importance of using genetic algorithms to aid the automated generation of unit tests, several researches who apply genetic algorithms, in any shape or form, were analyzed.
The following works implement different strategies to optimize the parameters in traditional genetic algorithms and related evolutionary algorithms. These works are listed according to different parameters' optimization:

- **Population size optimization**:
  - **"GAVaPS-a genetic algorithm with varying population size" [44]**: an age-based population size control, presented by Arabas, Michalewicz and Mulawka, which introduces the concept of "age" for chromosomes in the evolution process [44]. The algorithm employed by this work is denominated as Genetic Algorithm with Varying Population Size (GAVaPS). The authors applied this method to find optimal values for test functions, being targeted for a maximization problem [44]. This algorithm does not considerate any type of selection mechanism during the generation, as this is replaced with the concept of an aging process for the chromosomes. The age-based method of this algorithm consists in assigning a lifetime parameter to the chromosomes applying an aging process to them, i.e., each chromosome will have a fixed lifetime expectancy. In each iteration of the generation, the age of each chromosome is increased by 1 (their age is initially set to 0). After a chromosome exceeds its lifetime, the chromosome will be eliminated from the population. This strategy aims to reduce computational costs provided by standard selection methods by providing easier calculations [44].

The authors applied three different strategies for lifetime allocation in GAVaPS, delving upon proportional, linear and bi-linear lifetime equations. A total of 20 execution runs were executed with different established parameters. The results of this work were compared between the lifetime calculation strategies themselves and a simple genetic algorithm without optimizations. In summary, the linear strategy proved to be the best performing one but with higher computational cost, the bi-linear strategy was the cheapest method but didn't have a performance as good as the linear method and finally the proportional technique had medium performance and medium cost when considering all three different applications. However, all strategies performed better than the simple genetic algorithm in terms of cost (average number of function evaluations) and performance (average of maximal fitness values) [44];

– **"Evolutionary Algorithms with On-the-Fly Population Size Adjustment" [45]**: A population size mechanism introduced by Eiben, Marchiori and Valkó [45] called Population Resizing on Fitness Improvement Genetic Algorithm (PRoFIGA), consists in controlling the population size during the generation process using three different methods according to the best fitness value improvement. The mechanism becomes more biased towards exploration when the best fitness of the population improves [45]. In situations where the best fitness is improved, the population size increases in proportion to the fitness improvement and the remaining number of evaluations according to the max number of evaluations. When the best fitness is not improved, the population size will gradually decrease where a percentage between 1% to 5% is used, however in stagnation periods (where the best fitness has not been improved over x fitness evaluations) the population grows in size.

  PRoFIGA was evaluated in comparison with a traditional genetic algorithm, a parameter-less genetic algorithm [46], GAVaPS [44], Adaptive Population Size Genetic Algorithm (APGA) [47] and three variants of Random Variation of Population Size (RVPS) genetic algorithm in multi-modal problems. The results suggest that PRoFIGA was the second-best genetic algorithm with 20% fewer fitness evaluations across 100 runs on a multi-modal problem, being the APGA the best one in terms of efficiency and speed being followed by PRoFIGA, traditional genetic algorithm, parameter-less genetic algorithm and RVPS;

– **"An Emperical Study on GAs "Without Parameters"" [47]**: a different approach by Bäck, Eiben and Vaart [47], addresses the same issue about managing population size through an adaptive solution. In their work, they nominated the aging process as Remaining Lifetime (RLT) where in each chromosome a lifetime value is allocated, decreasing by one every iteration of the genetic algorithm. The RLT value is maintained for the fittest member, and an individual is removed from the population when the RLT reaches zero. The authors implemented this lifetime strategy and allocated it in a variation of a genetic algorithm made only to adapt the population size during the run. This variant is called APGA.

  The authors also implemented other variants of genetic algorithms to optimize

different parameters, such as crossover and mutation. They also included a traditional genetic algorithm in the research. The authors executed in total 750 runs for different test functions using the genetic algorithms and the results, regarding population size optimization impact, suggest the algorithm APGA has a better performance than the traditional genetic algorithm. They ranked all the algorithms according to speed of execution and how good they were in reaching the optimal value for the test functions (targeted for minimization problems) [47]. The APGA came in second place, while the traditional genetic algorithm came in third place;

- **"APOGA: An Adaptive Population Pool Size based Genetic Algorithm"** [48]: Adaptive Population Pool Size Based Genetic Algorithm (APOGA) was proposed by Rajakumar and George [48], where they delve upon the problem of population size control using an algorithm, created by them, that enables the population increase or decrease regarding their algorithm performance. The problem tackled by the authors uses a fitness function focused on a minimization problem, which means the lower the fitness scores, the better the results are. The major part of their work relies on their population resizing method. They use two different strategies to control the population size, being one targeted to increase the population and the other to decrease the population.

  For population increase, they evaluate the best fitness improvement, similar to the strategy shown in [47], the authors grow the population when the best fitness score is improved in each iteration or if the best fitness is not improved for an x amount of iterations. For population decrease, the aging process RLT is applied to this work. The aging method is essentially the same as described in research [47] where initially each chromosome is assigned a lifetime whose values are decreased by 1 for each iteration during the generation process. Those chromosomes who have a RLT value of 0 are removed from the population. Additionally, if neither of the above conditions are met (no improvement of the best fitness in each iteration or for some x amount of iterations), then a $D\%$ of chromosomes are removed from the population, being this a parameter from the interval $(0, 1)$. There is only one major difference to their application. The authors do not decrease the lifetime of the best chromosome, i.e, the chromosome who has the best fitness score of the population. The RLT verification happens each iteration of the generation process.

  This algorithm was tested against a standard genetic algorithm for an unimodal test function during 500 iterations of both approaches for five runs with different solution spaces (in this case for five different populations). The APOGA approach demonstrated a huge minimization (lower fitness values) in the end of the iterations compared to the standard GA revealing that the population pool size is starting to turn adaptive to the fitness function [48].

- **Selection optimization**:

– **"Adaptive selection routine for evolutionary algorithms"** [49]: Pham and Castellani [49] proposed an adaptive selection routine for evolutionary algorithms that consists in applying a stochastic noise to determine which individuals should mate (should be selected for reproduction). This work defends the importance of selection pressure and how its control can be favorable towards the optimization of the selection process in evolutionary algorithms. The authors divided their efforts into two main stages:

  * **Selection pressure adjustment**: as selection pressure is an important factor regarding population convergence for an optimal solution, the authors decided to control the selection pressure by adding individuals to two different structures and mating them between those two. Initially, a noise addition procedure was made to introduce a random positive perturbation to the fitness of each individual [49]. This process is made to control the selection pressure as the search goes on in the genetic algorithm. In early stages, the average fitness is likely to be low and if the population has individuals with high values of fitness compared to others, the fitness variance will be high. Adding a noise perturbation can help reduce these large variances and consecutively reduce the supremacy of better individuals in early low average fitness populations and aid the search to continue, which consecutively promotes a soft selection pressure at the beginning of the search. Over the iterations of the algorithm, the population will converge, fitness values will be higher and the fitness variance in the population will be even smaller, leading the search to an exploitative phase. This rationale meets the reasoning behind the proper selection pressure evolution [40]. Additionally, the authors also limit the selection of the same solution by a maximum of two times per list, which also helps to prevent early convergence;
  * **Mating individuals**: after the individuals are organized in both structures, a mating operation will start where the first individual from structure A mates with the last individual from the structure B, the second individual from structure A mates with the second to last individual from structure B and so forth until both structures are fully mated [49]. The individuals in both structures were organized in descending order of fitness. This crossed mating between individuals of both structures enables the reproduction between high and low fitness individuals, gradually improving the fitness over the course of the evolution process in the genetic algorithm.

    This technique was compared to other standard selection methods used in genetic algorithms such as proportional selection, tournament selection and fitness ranking in three different phases of the evolution process (early, mid and convergence evolution stages). A number of selections for each individual in the population was registered, where the proposed method was compared individually with each standard method. The results suggest that the proposed selection method agrees with the reasoning behind the proper selec-

tion pressure evolution [40] where the selection pressure gradually increases throughout the evolution process. As for the standard methods, the selection pressure generally showed that it would decrease throughout the evolution process, which is not ideal as it can lead to premature convergence;

- **"Parent Selection Operators for Genetic Algorithms"** [50]: another method to improve the selection of parents during the execution of the genetic algorithm was proposed by Jebari [50] where different standard selection methods are selected, properly evaluated between generations and the best result from the best selection method is taken into account during the generation process. His technique can be explained in the following steps:
  - **Selection of standard selection methods**: a set of standard selection methods in genetic algorithms was chosen in order to be evaluated during the execution of the algorithm. These selection methods include the roulette wheel selection, stochastic universal sampling, linear and exponential ranking selection, tournament selection and truncation selection;
  - **Evaluation of standard selection methods**: for each generation during the execution of the algorithm, every selection method of the set was applied and results were obtained. Only the best performing selection method was taken into account for the generation, i.e., the selection that provided the best individual was the one that made impact for the recombination phase. The selected individual was evaluated with two objective criteria, one that measures the quality of solution regarding the fitness value (the best fitness value is chosen) and the other to measure the diversity of the individual according to the rest of the population, where the most diverse individual from the population is selected [50].

    The work analyzed this selection technique against each standard selection method. The comparison occurred using three different runs with three different test functions. The results suggest that the new selection procedure obtained more optimal parents than the standard selection methods, proving to be better a selection method to find optimal parents. However, this work wastes a lot of computational resources as each standard selection method is evaluated in each generation to decide the best one to use, which can increase execution times for the genetic algorithm.

- **Crossover and mutation optimization**:
  - **"Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach"** [51]: the work proposed by Hassanat et al. [51] presents two deterministic parameter control techniques that focus on the variance of crossover and mutation rates during the execution of a standard genetic algorithm. The authors created these two methods and compare them to other two standard parameter tuning techniques, being fifty-fifty crossover/mutation ratios and the 0.9 crossover and 0.03 mutation ratios.

The authors applied this under the Travelling Salesman Problem (TSP) to find optimal solutions and proposed two different deterministic parameter tuning techniques:

* The first technique is called Dynamic Increasing of Low Mutation/Dynamic Decreasing of High Crossover (ILM/DHC). This is a deterministic method where an increase and decrease rule is applied to the mutation and crossover rates respectively. The objective behind this technique is to gradually increase mutation rate and gradually decrease the crossover rate during the execution of the genetic algorithm;
* The second technique is called Dynamic Decreasing of High Mutation Ratio/-Dynamic Increasing of Low Crossover Ratio (DHM/ILC) and is the opposite of the first technique. In this case, crossover rate gradually increases and mutation rate gradually decreases.

Six different set of experiments were executed with varying population sizes (small with 25 and 50 individuals, moderate with 100 and 200 individuals, large with 300 and 400 individuals) on the two proposed standard methods and the two standard deterministic parameter tuning techniques. In each set, the genetic algorithm was executed 10 times. The results obtained concluded that:

* For the **small population sizes**, ILM/DHC proved to be the best technique, as it showed the lowest number of convergences for the solution in relation to the other three techniques. This means that the technique was quicker than the others in finding an optimal solution. This is often explained that in smaller population sizes, a higher mutation rate is often more beneficial than a higher crossover rate as it introduces more variety to a small population [51] (which may not have good individuals as the population is smaller);
* For the **moderate population sizes**, ILM/DHC proved to be the best technique for population size of 100 individuals and DHM/ILC proved to be best for population size 200 individuals. The second result can be explained according to the growing size of the populations. A higher population size can include individuals with high fitness and a higher mutation rate in the beginning is not needed for this situation, on the contrary, a higher crossover rate is needed to generate better offspring from large population sizes [51]. As the generations go on, then the focus can go into mutation as the majority of individuals should have high fitness in the population and crossover is way less impactful than applying a mutation on the later generation levels;
* For the **large population sizes**, DHM/ILC proved to be the best in relation to the other techniques because of the evidence showed in the last point, as the increase of the population size indicates a better performance for increasing crossover and decreasing mutation rates.

The standard methods in all experiments did not have better results than the deterministic approaches, proving that even the deterministic approaches present

better results than the conventional values in standard parameter tuning techniques;

– **"Implementation of evolutionary fuzzy systems"** [52]: work developed by Shi, Eberhart and Chen [52] presents an adaptive approach regarding the crossover and mutation rates adjustment. The authors present an implementation of an evolutionary fuzzy expert system, they defend the use of a genetic algorithm in order to help the fuzzy expert system adapt its membership functions and set of rules during its execution. The main theme of this work does not present relevance for the topic of crossover and mutation rate adjustment, however the authors in order to present arguments to defend their view on the theme implemented a fuzzy expert system that adjusts crossover and mutation rates. This was done to further demonstrate how the fuzzy set rules were evolved with a genetic algorithm that has fixed and adapted crossover and mutation rates. The fuzzy system is composed of different membership functions that directly use feedback obtained from the genetic algorithm execution to adjust the crossover and mutation rates. This system takes as inputs several relevant data from the generation, such as: population's best fitness, number of generations of unchanged best fitness and variance of population fitness. The membership functions used were the left triangle, triangle, and right triangle that take this feedback and apply a set of rules to it. The data goes through defuzzification and the fuzzy system returns adjusted crossover and mutation rates.

  The authors implemented an evolutionary fuzzy system with and without crossover and mutation rate adaptation in order to see its significance when classifying an iris dataset. For the experiments that implemented the adaptation of crossover and mutation rate, they verified that these achieved the solution with a lower number of generations than the fuzzy system with fixed crossover and mutation rate;

– **"Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms"** [53]: adaptive methods to adjust the crossover and mutation rates during the execution of a genetic algorithm were proposed by Srinivas and Patnaik [53]. The implemented mechanisms focus on preventing the premature convergence into a local optimum [53] by adjusting the crossover and mutation rates when a possible convergence behavior is observed. The adjustment process is explained as follows:

  * **Convergence detection**: according to the authors, the convergence point of a GA can be detected by analyzing the fitness variation of the population, more concretely, the difference between the maximum fitness value and the average fitness value of the population. Studying this variation can enable an easier detection of a convergence point, as the difference between these values should be minimal when the search is gearing towards an optimal region [53]. This difference is used in the adjusting mechanisms;
  * **Crossover and mutation adjustment**: the adjustment of the crossover and mutation rates takes feedback from the generation in respect to the fitness

variation of the individuals. The authors have considered initially the fitness variation as the main aspect to evaluate for the adjustment, however they soon realized that this would not be enough as it could cause disruptive behavior for good solutions as the crossover and mutation rates would be the same for all the solutions [53]. This situation could cause the loss of genetic information on these good solutions. The authors decided to also consider the fitness of the solutions into the adjustment formula alongside the fitness variation.

The authors compared the performance of the adaptive GA with a standard GA over 30 runs, for each algorithm, in terms of how much generations each algorithm would need in order to achieve a desired solution. The results suggest that the adaptive GA outperformed the standard GA achieving the desired solution quicker requiring less generations. The adaptive GA also presented a surprising behavior when achieving an optimal solution. This behavior is the increase of the maximum fitness value of the population throughout the generations. As the mechanism is now considering the fitness of the solutions, the chance to disrupt already good solutions is now lower and consecutively better individuals can be generated with the genetic variety of the good solutions [53]. The standard GA, when achieving an optimal solution, does not present this type of behavior and gets stuck in a local optimum;

– **"An Emperical Study on GAs "Without Parameters""** [47]: a new self-adaptive method for crossover rates was proposed and a self-adaptive method for mutation rates was implemented by the authors Bäck, Eiben and Vaart [47]. The implemented methods are explained according different objectives:

  * **Crossover evolution**: a self-adaptive crossover method where an initial crossover rate was randomly chosen in the interval $[0, 1]$ and encoded into the individuals' representation scheme. After an individual is selected in the selection process, a random number in interval $[0, 1]$ is chosen and compared with the encoded crossover rate of the individual to see its mating potential. This is performed for both parents and after this process, both parents are evaluated regarding their mating potential. Depending on these mating evaluations, different crossover operations will occur, which can also include required mutation operations [47];
  * **Mutation evolution**: the mutation method applied by the Bäck, Eiben and Vaart [47] consists in adapting mutation rates that are encoded in the individuals. An initial mutation rate is encoded in the individuals' representation being its value chosen randomly in the interval $[0.001, 0.25]$, this mutation rate will undergo mutation operations during the generations to be updated.

The authors made the comparison of a traditional genetic algorithm with a Self-Adaptive Mutation Genetic Algorithm (SAMGA), a Self-Adaptive Crossover Genetic Algorithm (SAXGA) and a genetic algorithm that uses both self-adaptive methods with the adaptive population size method (SAMXPGA). They also considered a genetic algorithm with adaptive population size, but for this analysis of

the authors results, only the self-adaptive and traditional genetic algorithms are considered. The results observed reveal that:

* SAMGA and SAXGA revealed to be worse in terms of execution time than the traditional genetic algorithm when trying to find the optimum for test functions. One reason the authors gave for this situation is that the methods focused too much in optimizing their respective parameters instead of finding the overall objective of the search, this being the optimum;
* SAMXPGA is the best genetic algorithm when compared to the traditional genetic algorithm, SAMGA and SAXGA where it obtained the fastest execution times and achieved the optimum faster. One reason the authors present to explain this behavior is the adaptive population size greatly helps with the search process, reducing the time spent in evaluating individuals while only maintaining the fittest individuals for a longer time during the generations.

– **"Intelligent Mutation Rate Control in Canonical Genetic Algorithms"** [54]: the work done by Bäck and Schütz [54] introduces a deterministic schedule and a self-adaptive method to control mutation rate during the execution of genetic algorithms.

The deterministic schedule introduces a "time-dependent" mutation rate which is independent of the objective function in genetic algorithms and considers other factors such as a generation counter and the maximum number of generations in the algorithm.

The self-adaptive method is focused on this work because this method can obtain additional advantages between the strategy parameters and the objective function values of a genetic algorithm. It can ease the adaptation of strategy parameters without any kind of forced control [54]. This self-adaptive method focuses on specifying a mutation rate p_m in the interval $]0, 1[$ with the use of a learning rate γ to control the adaptation speed during the genetic algorithm.

The authors executed self-adaptive genetic algorithms, excluding crossover operations, that implement this self-adaptive mutation control rate method using two different selection methods. The selection methods used were proportional selection and (μ, λ)-selection. The authors did not compare the use of the self-adaptive mechanism with traditional genetic algorithms, as they only used the mechanism within a self-adaptive genetic algorithm. The mechanism presented excellent results, where the mutation rates adjust themselves to enable the improvement of the objective function value during generations [54]. The mutation rate is adapted with respect to the convergence speed of the algorithm [54] proving the importance of the self-adaptive method for the overall genetic algorithm performance when searching for optimal solutions.

3.3 Automated Unit Test Generation

This section presents some works that achieved the intended purpose behind this dissertation, which is enabling an automated unit test generation process for units of code. All of these works are listed below:

- **"Automated Support for Unit Test Generation A Tutorial Book Chapter"** [28]: an automated unit test generation technique for Python programs was proposed by Fontes, Gay, Neto and Feldt [28]. This work delves upon a search-based unit test generation technique, where it employs a local search algorithm being the hill climber algorithm and a global search algorithm being the genetic algorithm. Test suites were generated automatically with the aid of the algorithms from a class context that represented a Body Mass Index (BMI) calculator.

  The authors used different setups to organize both algorithms and only presented results for the genetic algorithm. These results were obtained after the genetic algorithm was executed with 1000 generations and the authors observed that coverage fitness and test suite size increased over the generations while the average test length tend to decrease. Overall, the authors successfully achieved the end result, which was to automatically generate unit tests, however they verified that the test suite themselves were not good as some introduced redundancy [28];

- **"Pynguin: Automated Unit Test Generation for Python"** [55]: a framework for Python programming language, focused in generating unit tests automatically, was developed by Lukasczyk and Fraser [55]. This is framework is designed to automatically generate unit tests for Python modules and it creates a set of test cases by executing the code present in the modules. It utilizes coverage measurement, more specifically branch coverage, and a set of unit test generation algorithms that include genetic algorithms and other techniques. Given a Python module, the framework is executed via Command Line Interface (CLI) with a set of mutable configurations and generates the unit tests according to the Python module. The result of this generation is a file which contains a test suite with the generated test cases for the module;

- **"EvoSuite: Automatic Test Suite Generation for Object-Oriented Software"** [56]: a unit test generation tool developed by Fraser and Arcuri [56]. This tool can generate test cases with assertions based upon the Java bytecode of the Class Under Test (CUT) [56]. It is configurable and can be executed via CLI. For the generation of test cases, this tool utilizes a genetic algorithm and mutation-based assertion generation to provide unit tests with the assertions. It focuses on the search-based test generation technique, where the maximum branch coverage is sought after.

3.4 Conclusion

This chapter included very important researches for genetic algorithms and works for the automated generation of unit tests. Optimization of genetic operators in genetic algorithms

has immense potential for this work, as it can potentially enhance the performance of the algorithms in the generation of unit tests. The existing works on automated generation of unit tests present very useful insights and approaches when it comes to generate unit tests automatically, which can be enriching for the development of this work.

Chapter 4

Implementation Details

4.1 Introduction

This chapter introduces a thorough explanation about the implementation process of this work. The chapter is composed of the following sections:

- **Class Under Test**: the section 4.2 introduces the class that will be the target of the unit test generation process. Its structure and composing elements will be explained thoroughly;
- **Representation Scheme**: the section 4.3 presents the chosen representation scheme for the chromosomes regarding the structure of the class. Additionally, it also presents the methodology applied to implement the concept of fitness evaluation on the chosen representation scheme;
- **Technologies and Libraries**: the section 4.4 demonstrates the technologies and libraries used for the practical implementation of this dissertation, while also presenting a brief explanation of their use;
- **Materials**: the section 4.5 specifies the hardware and software/tools used for the implementation of this dissertation;
- **Implementation Structure and Execution Procedure**: the section 4.6 explains, in throughout manner, the implementation structure as well as the chosen procedure for the GAs execution for the benchmarks;
- **Implementation Challenges**: the section 4.7 mentions the principal challenges about the development of this dissertation;
- **Implementation Limitations**: the section 4.8 specifies the principal limitations regarding the implementation of this dissertation;
- **Threats to Validity**: the section 4.9 demonstrates the potential threats to validity under the experiments performed in this dissertation.

4.2 Class Under Test

In order to generate unit tests, a piece of code must be first decided upon. In literature terms, the target of the generation is often called as SUT. In this particular case, a class was designed around a calorie intake calculation class being considered as a CUT.

This class will be the target for the generation of unit tests. This CUT was designed with the purpose of providing a simple calorie intake calculation for a person that desires to lose, maintain or gain weight. It uses three equations in total and a set of attributes necessary to them.
A brief summary of the elements that compose this class is presented below:

- **Variables**:
  - `weight`: weight of the person in kilograms in the interval $[40, 210]$;
  - `height`: height of the person in centimeters in the interval $[140, 220]$;
  - `age`: age of the person in years in the interval $[10, 80]$;
  - `gender`: person's gender, either male or female[1];
  - `bodyfat`: body fat percentage of the person in the interval $[0, 0.3]$;
  - `amount_exercise`: amount of exercise the person does, either sedentary, light, moderate, very active or extremely active.
- **Methods**:
  - `mifflin_stjeor_equation`: predictive equation for Resting Energy Expenditure (REE) done by Mifflin, St Jeor et al. [57]. The REE represents the energy consumption of an individual at rest [57]. This equation takes into consideration different factors such as gender, age, weight, and height of a person. The gender factor is distinct when a person is male (value defined to be 5) or female (value defined to be -161) [57]. The equation proposed for the authors [57] is as follows

$$REE = (10 \times Weight) + (6.25 \times Height) - (5 \times Age) + Sex \tag{4.1}$$

    where `Weight` refers to the weight of the person in kilograms, `Height` refers to the height of the person in centimeters, `Age` refers to the age of the person in years and `Sex` refers to the person's gender;
  - `katch_mcardle_equation`: formula that calculates Resting Daily Energy Expenditure (RDEE) for an individual. The RDEE is also sometimes referred to as REE. This formula was created by McArdle and Katch et al. [58]. This formula takes into consideration an individual's lean body mass on the contrary of other equations that only consider an individual's age, sex, weight, and height. The formula done by the authors [58] is as follows

$$RDEE = 370 + 21.6 \times (1 - bodyfat) \times Weight \tag{4.2}$$

    where `bodyfat` refers to an individual's body fat percentage and `Weight` refers to the weight of the person in kilograms;

[1] According to calorie intake calculation equations requirements.

- `tdee_calculation`: equation that determines the Total Daily Energy Expenditure (TDEE) of an individual. The TDEE represents the energy consumption of an individual throughout an 24 hour period [59]. This equation uses the individual's Basal Metabolic Rate (BMR) and multiplies it by a set of multipliers according to the individual's exercise activity. The formula and respective activity multipliers are as follows

$$TDEE = BMR \times activityfactor \tag{4.3}$$

  where `BMR` refers to the basal metabolic rate of the individual and the `activityfactor` refers to the type of activity done by the individual, which will have different multipliers according to the activity [60]. These activity multipliers are:

  * **Sedentary**: individual that does little to no exercise = 1.2;
  * **Lightly Active**: individual that performs light exercise $1-3$ days a week = 1.375;
  * **Moderately Active**: individual that exercises moderately $3-5$ days a week = 1.55;
  * **Very Active**: individual that does heavy exercise $6-7$ days a week = 1.725;
  * **Extremely Active**: individual that performs very heavy exercise 2 times a day = 1.9.

- `determine_calorie_intake`: method that calculates the amount of calories needed to lose, maintain and gain weight based upon the calories' consumption values of the National Health Service (NHS) [61]. According to the NHS, if an individual wants to lose or gain weight, he should be below or above 600 calories of their TDEE value (maintenance calories) in each day [61] and if the individual wants to maintain weight, he needs to reach his maintenance calories each day.

The thresholds defined for the variables above are defined as standard values, except for the `gender` and `amount_exercise` variables, whose values are specifically tailored for the calorie intake calculation equations.

4.3 Representation Scheme

One of the major difficulties behind the application of genetic algorithms is to choose a representation scheme that reflects the problem to be tackled. The individuals need to be represented in a genotype format, being this their encoding form in respect to their phenotypes. As previously mentioned in section 2.3.3.3.3, the representation for the individuals needs to be specifically tailored to the problem context and there is not a definitive or "right" way to perform this transition from phenotype space to genotype space.
This work delves upon the automated generation of unit tests for pieces of code and as such the representation scheme for the individuals of the genetic algorithm was tailored to fit a

unit test structure. The CUT described in section 4.2 is composed of a set of variables and methods. If we view this from a unit test perspective, there needs to be a way to translate each of these elements to the genotype space in order to fulfill the objective of this work. The chosen procedure for the representation is divided into CUT metadata and respective individual structure according to it. The section 4.3.1 and section 4.3.2 explain in detail how the representation scheme was implemented.

4.3.1 Metadata

According to the composition of the CUT, it was chosen to translate the CUT into metadata for easier manipulation. The metadata from the class was extracted into a file of JavaScript Object Notation (JSON) format.
The JSON format enabled a proper organization of the CUT constructor and methods into different objects with different properties or key-value pairs. This distinct separation into different objects made it easier to access the necessary data to create the individuals of the genetic algorithm. The metadata of the CUT is divided into the following objects and key-value pairs:

- `constructor`: representing the constructor of the class and the inherent attributes associated to it. It is divided into `name` and `parameters` key-value pairs:
  - `name`: key that represents the name of the CUT;
  - `parameters`: key that contains each attribute of the constructor, being an array of objects where each object represents each constructor's attribute. Each object of the array has a corresponding `type` key and keys associated to a predefined range of values.
- `other_functions`: key that contains a set of objects, which are the CUT methods, comprising getters, setters, and other functions. Each object of the array has a corresponding `name` (name of the method) and `type` (if it is a getter, setter, or another function) keys. There are some objects that include an additional key that contains parameters, which corresponds to an array that includes an object with corresponding key-value pairs that represent the parameters of the methods.

The metadata file contains the range of values specified for each variable. However, **these limits were slightly changed in the metadata file** in order to **target possible invalid values** when the individuals are generated. The maximum and minimum range of values were slightly increased and decreased by 5 units or 5 tenths. Additionally, for attributes that required specific characters, it was added a "non-defined" character "N" to act as an invalid parameter value. This is made to target possible erroneous situations (values out of the pre-defined interval and invalid values) in the CUT. To exemplify, certain methods are composed of `If Else` and `Elif` statements that check the parameter values to verify if they are outside the pre-defined range of values. If they are, a `ValueError` exception is thrown, indicating that an error occurred. This serves to verify all possible outcomes in the branches that compose the CUT as previously explained in section 2.2.7.

4.3.2 Solution Structure

The individuals of the population have a pre-defined structure according to the data present in the metadata extracted from the CUT. This structure will be referred to as "solution" for the remainder of this work for easier reading, as each individual represents a possible solution for the problem at hand. Each solution is composed of a **test suite that encompasses a set of tests cases that contains a set of attributes and methods necessary for the evolution process**. Important to mention that the constitution of each test suite for the solutions is all made using the `list` data structure in Python.
In a brief description, the population is a nested list involving a list of test suites where each test suite has a list of test cases and where each test case has a list of methods from the CUT. After the explanation of the solution structure, a figure is presented with phenotype and genotype format side by side for easier understanding.
The population for the different genetic algorithms will be composed of a set of solutions, where each of them is composed by the following elements:

- **Test suite**: each solution has only one test suite, which contains a set of two or more test cases. Each test case is composed of a set of methods that must contain a constructor and can contain zero or more functions (according to CUT structure). These inherent methods also have a structure of their own:
  - **Method identifier**: each method has an identifier that determines which type of method was selected from the CUT. The constructor has the identifier of value -1 and the other functions have corresponding identifiers in the interval $[0, number\,of\,class\,attributes[$ according to their position index in the metadata file;
  - **Method parameters**: each method has a list that contains the parameters associated to it. These parameters differ in regard to the type of method that was chosen during the initial random generation of solutions.

- **Attributes**: a solution is composed of an additional set of attributes that are necessary for the genetic algorithm execution and respective optimizations. These attributes are:
  - **Encoded crossover rate**: each solution has a self-encoded crossover rate that is randomly generated within the interval $[0, 1]$ during the initial generation of the solution. It's used in the self-adaptive crossover rate optimization when the self-encoded crossover rate is compared to a random number in the interval $[0, 1]$ to decide which genetic operator (crossover, mutation or even both) to apply to the solution. This method was presented in the work **"An Emperical Study on GAs "Without Parameters""** [47] mentioned in section 3.2;
  - **Encoded mutation rate**: each solution has a self-encoded mutation rate that is randomly generated within the interval $[0, 1]$ during the initial generation of the solution. It's used in the self-adaptive mutation rate optimization, alongside other data, to calculate a new mutation rate for the solution, adjusting its mutation rate

across the genetic algorithm execution. This method was proposed by the work **"Intelligent Mutation Rate Control in Canonical Genetic Algorithms"** [54] mentioned in section 3.2;

- **Fitness score**: each solution has a fitness score value associated to it. This value is pre-defined at the beginning and updated across the genetic algorithm run either during the initial generation or through genetic operations;

- **Rank score**: a pre-defined rank score is attributed to a solution during the initial generation. This score is then updated when rank-based selection is used during the evolution process;

- **Mating chance**: a mating chance is calculated for each solution during the adaptive selection method execution. It's the probability for the solution to be inserted into the mating lists for further reproduction. This attribute is done solely for the adaptive method described in the work **"Adaptive selection routine for evolutionary algorithms"** [49] mentioned in section 3.2;

- **Maximum selections**: maximum number of selections of the solution to be inserted into the mating lists. Each solution can be selected up to two times in order to avoid early convergence. This was implemented in the adaptive selection method proposed by the work **"Adaptive selection routine for evolutionary algorithms"** [49] mentioned in section 3.2;

- **Lifetime value**: value that determines the remaining lifetime of a solution. This is used in the population optimization method to control the size of the population during the genetic algorithm execution. This attribute is mostly used to evaluate if the individual needs to be removed from the population according to the remaining lifetime it has. This method was presented by the work **"APOGA: An Adaptive Population Pool Size based Genetic Algorithm"** [48] mentioned in section 3.2.

Important to mention that the self-encoded rates were allocated into the individual's class representation, not at the genotype level. This decision is open for discussion as the representation itself, mentioned in section 2.3.3.4, is not specifically stating that it needs to be at a genotype level. Despite this subjective matter, this implementation choice still goes in line with the "representation" idea of an individual, as each solution represents an individual in the population.

In order to provide a better visualization of how each individual is composed at a genotype level, Figure 4.1 demonstrates an image side by side of both genotype and phenotype formats of an individual.

```python
from cut import *

def test_case_0():
    cut = calorie_intake_calc(99.5,183.07,60,'F',0.11,'M')
    cut.height = 165.79
    result_mifflin_stjeor_equation = cut.mifflin_stjeor_equation()
    cut.height = 178.34

def test_case_1():
    cut = calorie_intake_calc(97.19,143.89,69,'M',0.02,'V')
    result_tdee_calculation = cut.tdee_calculation()
    result_mifflin_stjeor_equation = cut.mifflin_stjeor_equation()
    result_katch_mcardle_equation = cut.katch_mcardle_equation()
    cut.weight = 63.55
```

```
[
    [
        [-1, [99.5, 183.07, 60, "F", 0.11, "M"]],
        [1, [165.79]],
        [6, []],
        [1, [178.34]]
    ],
    [
        [-1, [97.19, 143.89, 69, "M", 0.02, "V"]],
        [8, []],
        [6, []],
        [7, []],
        [0, [63.55]]
    ]
]
```

Figure 4.1: Example of the phenotype format (left side) and the genotype format (right side) of an individual.

On the left side of the image, the phenotype format (how a human visualizes it) is represented, and on the right side is the genotype format (how the algorithm understands it). As already mentioned in section 4.3.2, a solution is a nested list containing a set of test cases and inherent CUT methods. The genotype format is the genetic constitution of the individual during the algorithm execution. By visualizing Figure 4.1, the genotype of the individual is a `list` that contains a set of two test cases. In each of them, a `list` is also present where each element of it corresponds to a CUT method, which is composed of an identifier and a set of parameters. For example, for the first test case, from top to bottom, three methods were created which correspond to the constructor, the height setter and the Mifflin Stjeor equation respectively (height setter repeats itself at the end).
Using the search provided by the stochastic algorithms, this representation scheme is applied for all the created individuals, focusing upon the arrange and act phases mentioned in the unit test structure section 2.2.5. It is important to mention that, in each test case of an individual, the constructor is the first instruction to be generated. After the constructor is defined, the following CUT methods are generated. This generation process, including the initialization of parameters and the choosing of CUT methods, is all done randomly according to the metadata pre-defined range of values, obeying the stochastic nature of the genetic algorithms.

4.3.3 Fitness Evaluation

During the generations of the genetic algorithm, each individual must be evaluated according to its suitableness to a predefined goal. The implementation of this work focuses on generating unit tests with a maximization mindset, i.e, achieving unit tests that cover the maximum amount of code possible.
The implementation process of fitness evaluation is composed of two activities, being running the generated tests and evaluate them according to a desired code coverage metric. This is possible due to the use of two Python modules which are:

1. **`Pytest:`** testing framework that provides enough tools for the creation and execution of tests in Python [62]. It requires Python's version 3.7 or above in order to be used. The `Pytest` testing framework runs any file of the form `test_*.py` or `*_test.py` in the current directory and respective subdirectories [62].

This file annotation format is taken into account during the generations, where each individual's phenotype is created according to the genotype that was generated from the algorithm execution. The phenotype file name includes the form `test_*` for it to be detected and executed by `Pytest`. This tool is necessary to execute the generated unit tests first before the code coverage calculation can be applied;

2. **`Coverage` module**: it measures code coverage in Python, measuring which code was executed and analyses the source code that could have been executed [63]. It requires Python's version 3.8 or above in order to be used. The `Coverage` module executes a test suite and gathers data that can be visualized in different formats. The tool measures the coverage, by default, using the statement coverage metric, however it can be also set up to measure coverage using the branch coverage metric.

   As stated in the previous point, the `coverage` module needs a test runner to execute the tests first in order to have information about which parts of the code were executed. The tool works with `Pytest`, `Unittest` and `Nosetest` test runners. After the execution of the tests, the tool is able to calculate the code coverage according to the coverage metric chosen.

   During the execution of the test, the tool writes the data into an intermediate file, which is a SQLite database, that records which lines of the test were executed [63]. This file is then read for the calculation of the code coverage according to a specific coverage metric. The results are then available via CLI output, HyperText Markup Language (HTML), JSON, annotated source or Extensible Markup Language (XML). The results show the number of executed and missed statements (including which lines were missed) and the percentage of code coverage obtained from the execution.

The used coverage metric in this work was chosen to be branch coverage according to the CUT specification, which is heavily focused on branch conditions. Using this metric, each and every possible outcome present in the branch conditions can be tested. The general application of the concept of a fitness function was explained earlier, where the conjunction of two tools enable a proper analysis of code execution as well as to determine how much code coverage was obtained. The question now lingers on **when this evaluation is applied** during the genetic algorithm execution. The fitness evaluation process is always applied **whenever an individual is created**. In the genetic algorithm implementation, this includes the initial setup of the population as well as crossover and mutation operations, where a new individual is always created as its genetic information undergoes changes.

4.4 Technologies and Libraries

This work was implemented using the Python programming language, whose version was 3.8.10. The programming language choice is mostly justified by the sheer amount of works involving unit testing for non-dynamic languages such as Java for example. There is a low amount of solid researches and works that focus on the automated test generation in the Python programming language, as the work presented in section 3.3 for Python was one of

the very few that actually made a solid impact for this theme. Additionally, Python provides a great amount of useful and simple libraries and tools for the implementation of genetic algorithms, such as coverage modules, and its respective optimizations, such as fuzzy system libraries, done in this work.
The libraries and modules employed in this work are:

- `os`: module to use the operating system dependent functionality such as the reading and writing of files. This module was used to read and write files for the individual's metadata, the representation scheme of the chromosomes (genotype format) and phenotype format, for path verification and handling, as well as folder creation and removal for the results of this work;
- `json`: data format to store and exchange data, being written in JavaScript object notation. It's used in this work for specification and handling of metadata for the class under test. The data obtained from the generation process and the best generated unit tests genotype form are written in this format;
- `configparser`: module that implements basic configuration language. This was used to dynamically adjust configurations for the different genetic algorithms executions;
- `numpy`: library used for scientific computing, mostly for the treatment of multidimensional arrays. It's used to calculate the standard deviation of the results obtained from the generation process and for the implementation of the genetic algorithms;
- `random`: module that implements pseudo-random number generators for various distributions. This module is used across different parts of the implementation, as the genetic algorithms are of stochastic nature;
- `math`: module providing access to mathematical functions. This module is used for exponential calculation in the genetic algorithms optimization methods;
- `subprocess`: module that allows the spawning of new processes, connection to respective input/output/error pipes and obtainment of their return codes. The `subprocess` module is used to enable the coverage module to run in the background during the execution of the genetic algorithms;
- `skfuzzy`: tool that implements fuzzy logic, consisting in a collection of fuzzy logic algorithms. This is used to implement a fuzzy system used in the genetic algorithms' optimization;
- `matplotlib`: low level graph library used to create static, animated and interactive visualizations. This library is used to plot the results obtained from the generation process after the genetic algorithms executions;
- `copy`: module that provides shallow and deep copy operations for mutable collections. This is used to copy information from the nested data structure assigned to the individuals during the generation process;

- `shutil`: module that offers high-level operations on files or collection of files. It's used for the clean removal of directories during different executions of the script that contains different genetic algorithm configurations;

- `Enum`: module that implements a set of symbolic names bound to unique values. This module is used to group the configuration data into an enumeration for use during the genetic algorithm execution;

- `resource`: module that provides mechanisms to measure and control system resources utilized by a program. This module is used to obtain the start and end times of the genetic algorithms executions to calculate the elapsed time of the executions;

- `statistics`: module that calculates mathematical statistics of numeric data. This module is used to calculate the variance of population fitness needed as an input for the fuzzy system;

- `coverage`: tool that measures code coverage for Python programs. It monitors the program, identifies and evaluates which parts of the code were executed. This tool is used to calculate the fitness score of each individual in the population;

- `pytest`: testing framework used to write and execute tests for Python. It is used to run the test suites, created during the genetic algorithm execution, which aids the coverage tool in its code coverage assessment;

- `mutatest`: tool that measures the mutation score for Python programs. This tool is used to evaluate how well the best generated test suites, obtained from this dissertation experiments, can detect artificial faults.

4.5 Materials

The main objective behind this work falls upon the automated generation of unit tests for a target CUT. Moreover, this dissertation also aims to benchmark the different executed GAs to evaluate their performance and impact in generating representative unit tests. The execution of these GAs were all under the CPU and in order to not introduce any type of inconsistent or irregular results due to hardware changes, the whole experiment was performed under a singular virtual machine instance hosted in the Digital Ocean cloud computing platform. The virtual machine specifications are DO-Premium-AMD virtual CPU with 4 cores, 8 GB Random Access Memory (RAM), 160 GB Solid-State Drive (SSD) with Ubuntu 22.04 (LTS) x64 distribution. The used software/tools used in this dissertation include: Visual Studio Code to create all the Python code and its inherent Secure Shell (SSH) extension to connect remotely to the virtual machine and Git version control system [64] to save all the programming and research work.

4.6 Implementation Structure and Execution Procedure

The plan behind the GAs execution is explained in this section alongside another relevant elements associated to the implementation. This section presents the relevance of specifying a configuration file for the GAs execution in section 4.6.1, the overall GA structure and its inherent elements in section 4.6.2 as well as the execution strategy for the different GAs according to their genetic attributes optimizations in section 4.6.3.

4.6.1 Configuration File

A configuration file was created to hold a set of different attributes for easier execution of the different GAs. The configuration file is a Windows and MS-DOS initialization file that bears the `INI` format extension. This `INI` file has different set of configurations divided in five sections:

- **Metadata**: section that contains the field `metadata_location` to specify the location of the metadata file that holds the necessary CUT information;
- **Configurations for the GA**: section that contains a wide set of basic configurations for the GA execution. This section is divided into the following fields:
  - `max_number_functions`: this field is used to determine the maximum number of CUT functions an individual can have in its test cases. For the first benchmark, presented in section 5.2, this field was set with value 10. For the second benchmark, presented in section 5.3, this field was set with the value 4;
  - `max_number_test_cases`: this field is used to determine the maximum number of test cases an individual can have in its test suite composition. For the first benchmark, presented in section 5.2, this field was set with value 10. For the second benchmark, presented in section 5.3, this field was set with the value 4;
  - `tournament_size`: this field specifies the tournament size for the tournament selection method. The GA was designed with the standard two individuals in mind, which means that operations like crossover were designed for two individuals, so the used and recommended value for this field is 2;
  - `max_number_generations`: this field specifies the maximum number of generations by which the GA can run. For the first benchmark, presented in section 5.2, this field was set with the value 100. For the second benchmark, presented in section 5.3, this field was set with the value 1000;
  - `fitness_max_stagnation_period`: this field specifies the maximum stagnation period (number of generations) by which the GA can run without fitness improvement (stagnation of the best fitness score). For the first benchmark, presented in section 5.2, this field was set with the value 30. For the second benchmark, presented in section 5.3, this field was set with the value 100;

- `fitness_function_type`: this field specifies the type of fitness function used in the GA. This can range between two types of coverage according to the `Coverage` module limitations, as the only available coverage metrics are the branch coverage and statement coverage metrics. For both benchmarks, presented in sections 5.2 and 5.3, this field was set with `branch coverage`;
- `fitness_iteration_limit`: this field specifies the iteration limit by which the GA can run without fitness improvement. This field has a different purpose than the `fitness_max_stagnation_period` as it's used for the population size control optimization. For the first benchmark, presented in section 5.2, this field was set with the value 2. For the second benchmark, presented in section 5.3, this field was set with the value 5.

- **Configurations for genetic operators**: section that contains a wide set of configurations related to the genetic operators. This includes configurations for the population, selection, crossover and mutation operations. This section is divided into the following fields:

  - `population_size`: this field specifies the population size for the initial population creation during the GAs execution. For the first benchmark, presented in section 5.2, this field was defined with the value 25. For the second benchmark, presented in section 5.3, this field was defined with the value 55;
  - `population_control`: this field specifies if the population control optimization method, referring to the work "APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48], can be activated during the GAs execution. The value of this field was set to `True` during the GAs population benchmark and to `False` for the remainder of the first benchmark. For the second benchmark, this field was only set to `False` for the Traditional Genetic Algorithm (TGA) while for the other GAs this field was set to `True`;
  - `selection_type`: this field specifies the type of selection method to be used during the GAs execution. This selection type field can take one of the following values: `random`, `roulette_wheel`, `adaptive`, `rank` or `tournament`. The `adaptive` value applies the methodology of the research [49]. The standard value defined for this field is `tournament`;
  - `crossover_type`: this field specifies the type of crossover method to be used during the GAs execution. This crossover type field can take one of the following values: `deterministic`, `self-adaptive`, `adaptive`, or `uniform`. The `deterministic`, `self-adaptive` and `adaptive` values enable the methodologies of the researches [51], [47] and [52] respectively. The standard value defined for this field is `uniform`;
  - `crossover_rate`: this field specifies the crossover rate to be used during the crossover operations. This field was set with the value 0.50. This was assigned based on a standard range of rates for this genetic operator [42] which is $[0.45, 0.95]$;

- `crossover_rate_adjustment_type`: this field specifies the type of strategy to apply when the deterministic crossover method, proposed in "Choosing Mutation and Crossover Ratios for Genetic Algorithms — A Review with a New Dynamic Approach" [51], is applied during the GAs execution. This field can take one of the following values: `ilc` or `dhc`;
- `mutation_type`: this field specifies the type of mutation method to be used during the GAs execution. This mutation type field can take one of the following values: `add_test_case`, `delete_test_case`, `deterministic`, `adaptive`, `self-adaptive` or `change_parameters`. The `deterministic`, `adaptive`, `self-adaptive` values apply the methodologies of the researches [51], [52] and [54] respectively. The standard value defined for this field is `change_parameters`;
- `mutation_rate`: this field specifies the mutation rate to be used during the mutation operations. This field was set with the value 0.15. There is a high discrepancy in the literature related to the desired interval of mutation rates, as intervals such as $[0.001, 0.01]$ or $[0.001, 0.05]$ are suggested [42, 53]. This value was chosen to be a slightly higher than the standard values due to its low probability, however it was not increased too much in order to not disrupt the genetic information too often;
- `mutation_rate_adjustment_type`: this field specifies the type of strategy to apply when the deterministic mutation method, proposed in "Choosing Mutation and Crossover Ratios for Genetic Algorithms - A Review with a New Dynamic Approach" [51], is applied during the GAs execution. This field can take one of the following values: `ilm` or `dhm`.

- **Configurations for GAs optimizations**: section that contains a wide set of configurations for the GAs optimizations. This section is divided into the following fields:

  - `population_decrease_rate`: decrease rate used in the population control method, proposed in the research "APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48], during stagnation of fitness improvement. This field was set with the value 0.20 as per parameter setting description in the research;
  - `uniform_number_crossover`: control variable for the implementation of the uniform crossover operation to decide which part of the genotype would be exchanged between the parents. This field was set with the value 0.5;
  - `lt_max`: maximum lifetime for the individuals in the population. This is used during the RLT mechanic of the population control method proposed in the work "APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48]. For the first benchmark, presented in section 5.2, this field was set with the value 10. For the second benchmark, presented in section 5.3, this field was set with the value 20;
  - `lt_min`: minimum lifetime for the individuals in the population. This is used during the RLT mechanic of the population control method proposed in the work

"APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48]. For the first benchmark, presented in section 5.2, this field was set with the value 1. For the second benchmark, presented in section 5.3, this field was set with the value 4;

- `alpha`: variable used in the population growth operation of the population control method, proposed in the research "APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48], during fitness improvement. For the first benchmark, presented in section 5.2, this field was set with the value 0.7. For the second benchmark, presented in section 5.3, this field was set with the value 0.5;

- **File paths**: section that contains a wide set of file locations for the GA execution. This section is divided into the following fields:

  - `fuzzy_membership_functions`: path for the files that will store a visual representation of the membership functions that compose the fuzzy system described in research "Implementation of evolutionary fuzzy systems" [52];
  - `intermediate_test_suite`: path for the files that will contain new test suites each time an individual generation occurs. These tests suites are written in phenotype format, corresponding to the individual's genetic information;
  - `generations_stats`: path for the files that will store graphs relative to generation results after the GA execution;
  - `best_generated_test_suite`: path for the files that will store the best generated test suite in phenotype format after the GA execution;
  - `generation_data`: path for the files that will store the data obtained from the generations after the GA execution;
  - `benchmark`: path for the files that will store a visual representation of the GAs performance comparison on a diverse set of evaluation metrics;
  - `generations_stats_history`: path for the files that will store the generation stats history for the GAs (data from each generation) in `text` format.

- **Scripts**: section that contains configurations for the execution of additional scripts. This section is divided into the following fields:

  - `execution_script`: this field is used as a control variable during the GA execution. This field is set to `False` by default and it's only set to `True` during the `execution` script. This is done to create an initial population at the beginning of the `execution` script, as this population is intended to be the same across the multiple GAs executions to avoid any type of advantage or disadvantage for the GAs. In the end of the `execution` script, this field is set to `False`. In cases where only the `genetic algorithm` script is executed, the population is created during the `genetic algorithm` script run;

– **`execution_optimizations`**: this field is used as a control variable in order to determine which benchmark to execute. If it's set to `True` then the benchmark of the optimized GAs will be executed. If it's set to `False` then the benchmark for the optimizations of the GA genetic attributes will be executed instead.

A clearer explanation is needed regarding the `Scripts` section. The adopted implementation of this dissertation includes two major scripts. One that executes only one GA instance (referring to the `genetic algorithm` script), and one that executes the ten iterations of GA executions for each planned benchmark (referring to the `execution script`). The first one holds the necessary GA implementation and can be executed as it is, however in order to perform the two benchmarks on differently configured GAs a singular script that would be able to execute and obtain generation data from multiple GAs was needed and hence the `execution` script was implemented. After the end of the `execution script`, the GAs are compared regarding generation performance. This benchmark topic is explained thoroughly in section 4.6.3.

4.6.2 Genetic Algorithm Execution

This section is aimed at detailing the GA execution flow by providing a set of algorithm flowcharts that denote the essential steps during the execution. A description about each flowchart is presented, showcasing the focal points in each flowchart diagram. The GA flowchart diagrams are demonstrated in Figures 4.2, 4.3, 4.4, 4.5, 4.6 and A.4.
The flowchart represented in Figure 4.2 showcases the overall structure that composes the GA execution. It's composed of different execution phases which include the **`initialization`**, **`selection`**, **`crossover`**, **`mutation`** and **`population control`** phases. Each of these phases will be distinctly focused on the figures below. As for this overall structure, the GA starts off by initializing a set of needed variables alongside population creation and fitness assessment. After this first phase, the GA will undergo a selection process and depending on the verified condition of the decision nodes, the GA will follow different execution paths that may or may not follow traditional GA execution routes.

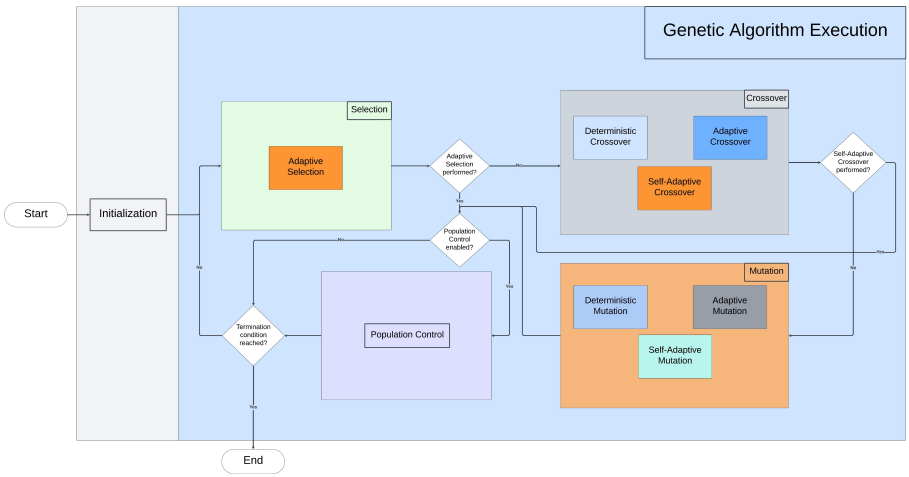


Figure 4.2: Genetic algorithm general execution flowchart.

There is an **important point that needs to be discussed regarding the selection of execution phases** during the GA run. Some implemented works, more concretely the adaptive selection routine proposed in research [49] and the self-adaptive crossover method proposed in research [47], cannot follow the standard GA genetic operations route as they are not compatible with the rest of the genetic operators.

The adaptive selection routine, as explained in section 3.2, performs an inherent crossover operation and the research itself does not mention any type of applied mutation method, so the `mutation` phase needs to be skipped when this adaptive selection method is applied. These are the reasons why the GA goes instantly to the `population control` phase instead of passing through the `crossover` and `mutation` phases when the adaptive selection method is applied.

The self-adaptive crossover method, as explained in section 3.2, performs different crossover operations that also include required mutation operations depending on the mating potential of the parents. These required mutations, according to my interpretation of the research, need to happen if certain criteria are met. This is not compatible with the `mutation` phase, which includes the probability check of the mutation rate and thus goes against the necessary occurrence of the mutation operation during the self-adaptive crossover method.

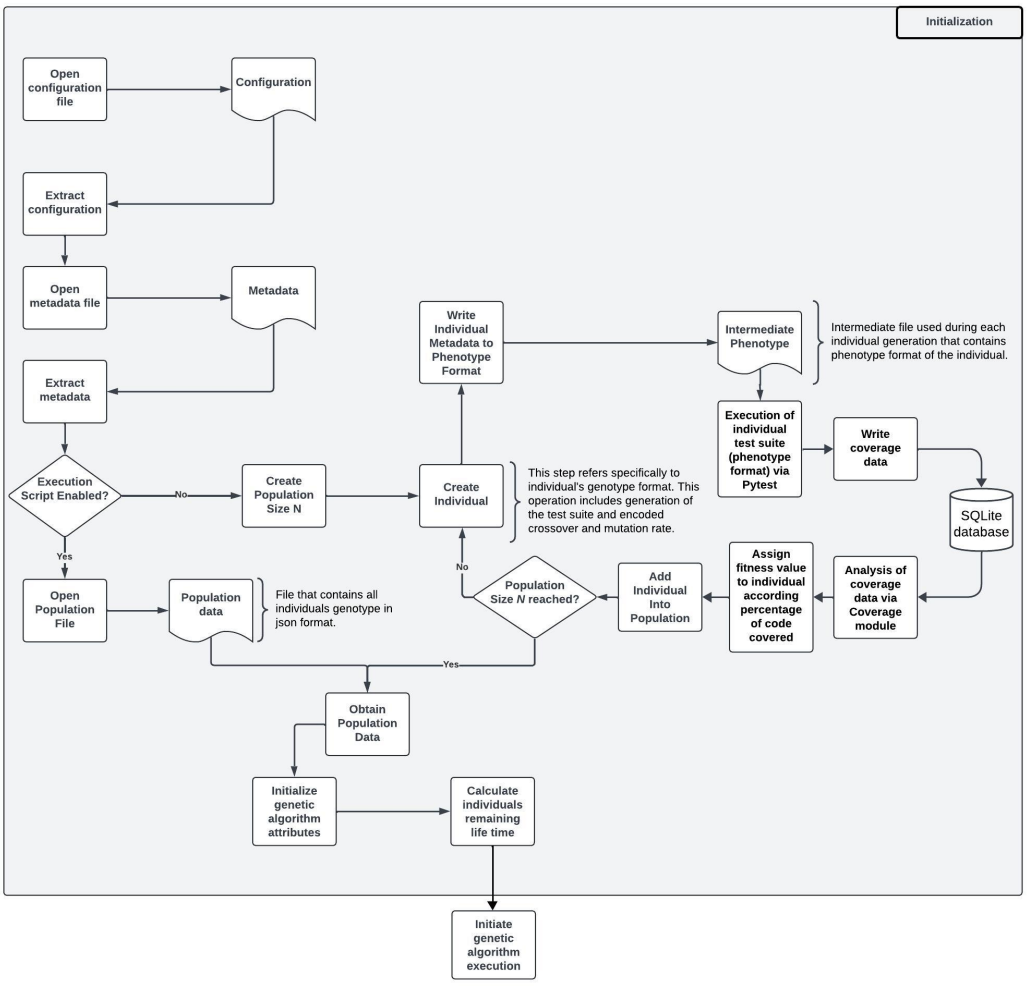


Figure 4.3: Genetic algorithm initialization flowchart.

Figure 4.3 presents the `initialization` phase of the GA. This is the first phase before the GA execution, where a set of GA attributes and additional files are initialized. A configuration file was created to hold a set of different attributes for easier execution of the different GAs.

In this initialization phase, the configuration file is read, and its configuration is "allocated" for the GA execution. The metadata file that holds the CUT metadata, described in section 4.3.1, is read and then a verification is made on the configuration to assess if the `execution_script` field is activated. If so, then a population was already created previously and its data will be obtained to be used during the GA execution. If not, then a new population is created, where each individual is generated randomly and attributed a fitness score with the `Coverage` module described in section 4.3.3. After the population creation, all individuals need to have their RLT calculated, according to the explanation provided in research [48]. This is a necessary step for the application of the `population_control` phase at the end of each GA generation.

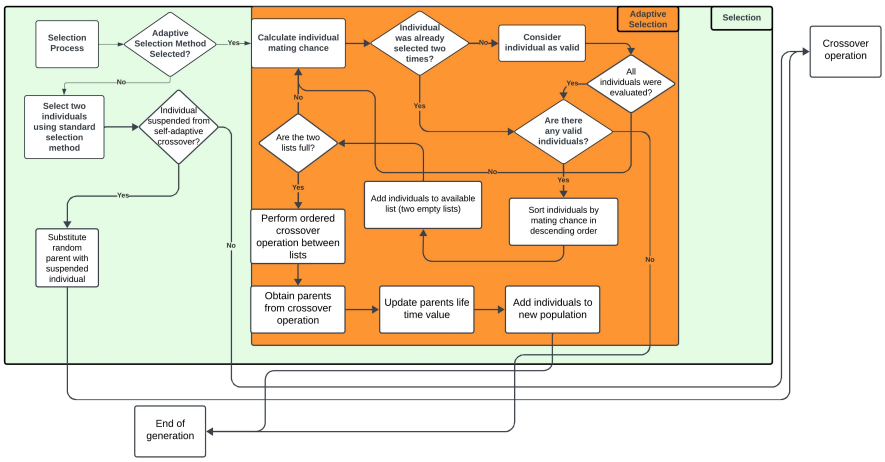


Figure 4.4: Genetic algorithm selection flowchart.

Figure 4.4 demonstrates the `selection` phase of the GA. This is the second phase to be executed after the `initialization` phase presented in Figure 4.3. This phase includes standard and adaptive selection methods. During this phase and subsequent phases, the optimizations for the genetic attributes start to be applied under certain conditions. The GA selection process has an initial verification to see if the **adaptive selection method** is specified in the **configuration file**. If that is the case, then the selection process will undergo the set of operations defined in the adaptive selection method proposed in research [49]. The **verification of the type of selection method is necessary** because of the restrictions in the adaptive selection method, as explained in the beginning of this section. In cases where the adaptive selection method is not selected, a standard selection method is applied instead. The main difference behind the adaptive and standard selection methods is that, at the end of the adaptive method, the GA advances into a new generation while the standard methods enable the GA to continue its genetic process by advancing into the `crossover` phase.

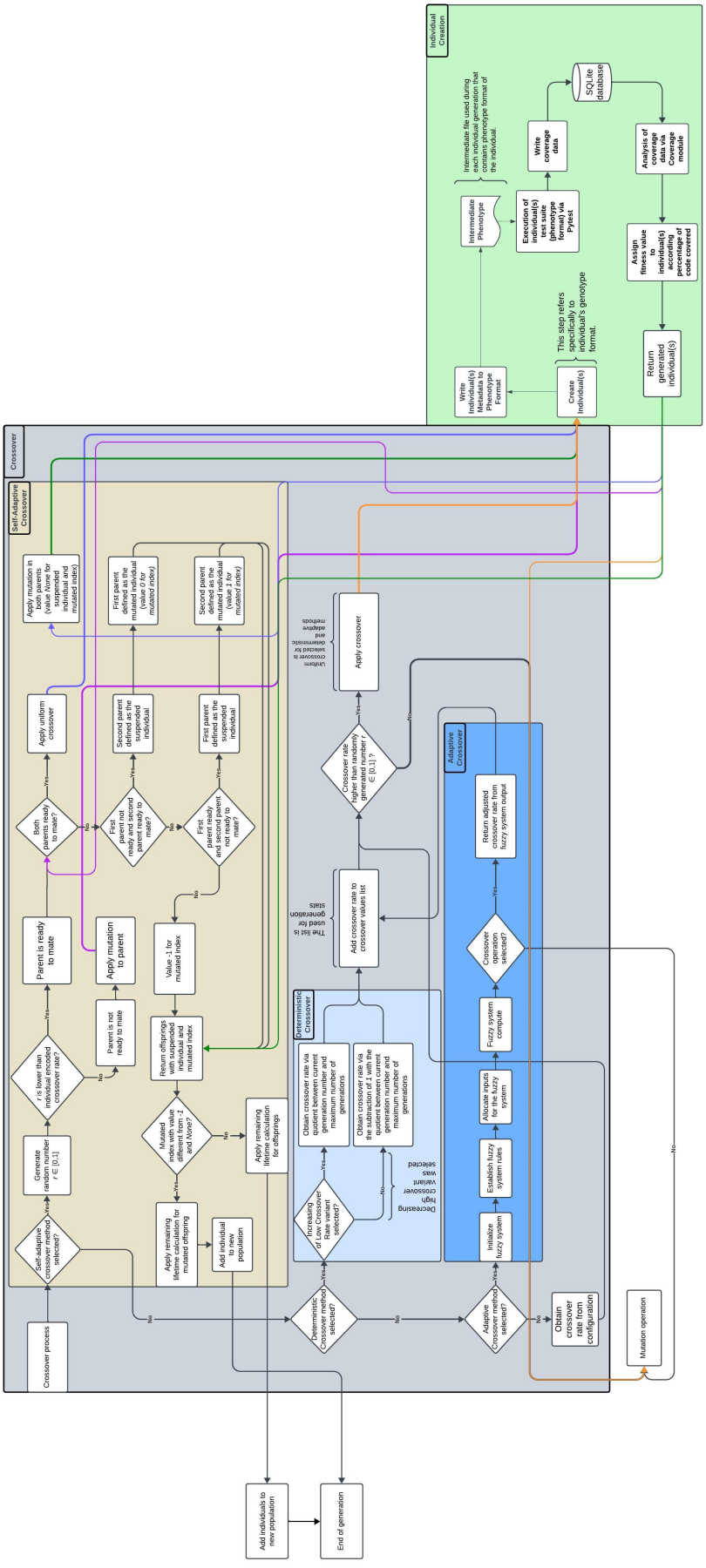


Figure 4.5: Genetic algorithm crossover flowchart.

Figure 4.5 depicts the `crossover` phase of the GA. This is the third phase to be executed after the `selection` phase if the adaptive selection method was not performed. The GA crossover process can be applied under different methodologies that include deterministic [51], adaptive [52], self-adaptive [47] and standard variations. The optimization aspect in this phase also comes with a restriction for the genetic operation order. The imposed restriction happens during the self-adaptive crossover method, where the GA can perform a predefined set of crossover operations in conjunction with mutation operations. The **verification of the type of crossover method is also necessary** in this phase, as the self-adaptive method advances the GA into a new generation at the end of the method, similar to what happens in the adaptive selection method. Earlier in this section, the rationale behind the skip of the `mutation` phase during the self-adaptive crossover method was presented. In situations where this self-adaptive crossover method is not defined in the configuration file, the GA follows the standard genetic operation order advancing into the `mutation` phase. The crossover operation itself is rather simple to explain in this implementation scope. The selected parents will exchange genetic information between them and as these changes are making new individuals, an individual creation process is applied. This process creates a pair of individuals that reflect the changes made in the parents and returns them with adjusted fitness scores. The new fitness scores for the offsprings are assigned via the conjunction of the `Pytest` framework and the `Coverage` module.

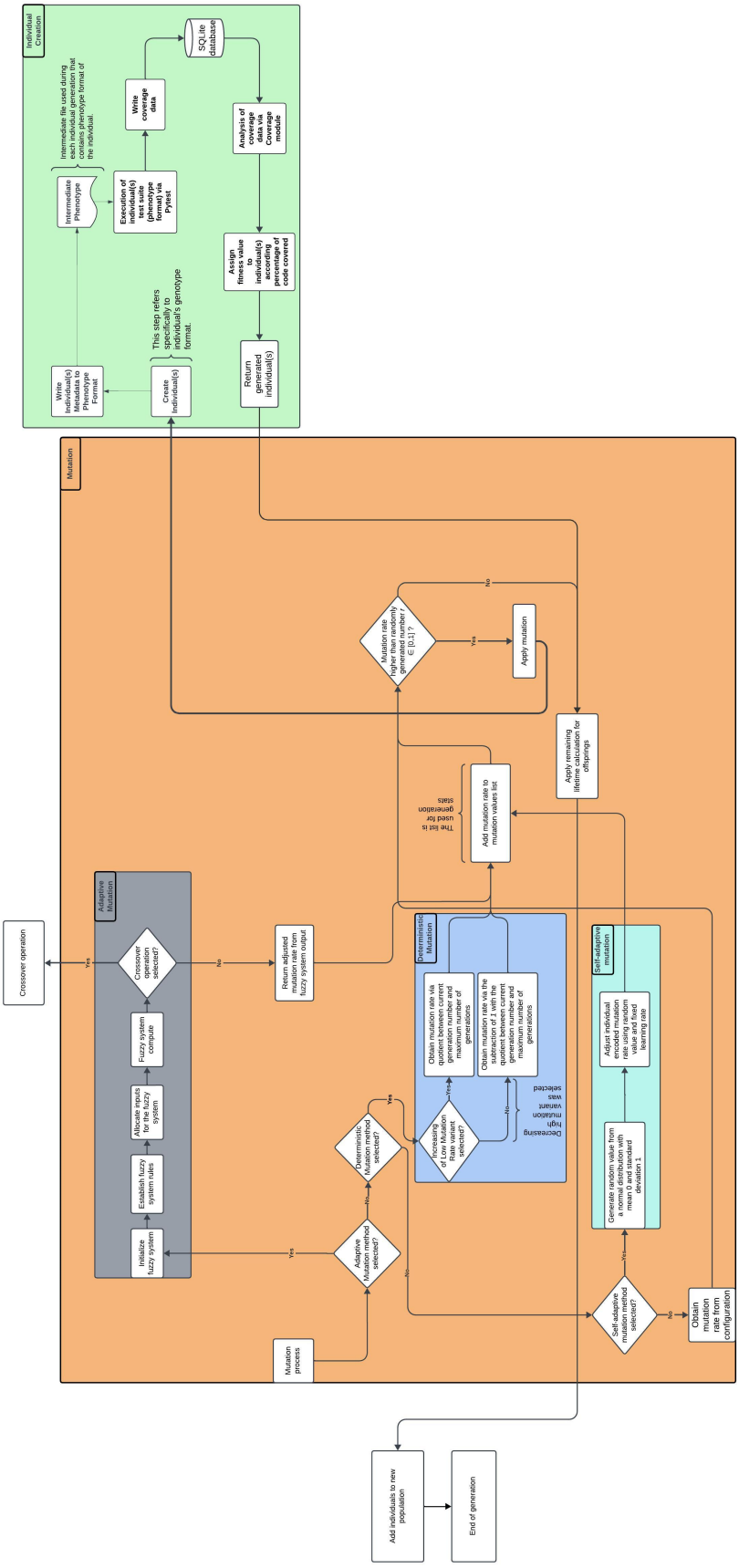


Figure 4.6: Genetic algorithm mutation flowchart.

Figure 4.6 represents the `mutation` phase of the GA. This is the fourth execution phase after the `crossover` phase if the self-adaptive crossover method was not applied. The GA mutation process can be performed with standard, deterministic [51], adaptive [52] and self-adaptive [54] methods. This phase does not impose any type of restrictions to the order of the genetic operations, as the `selection` and `crossover` phases do. The `mutation` phase shares some similarities to the `crossover` phase, as the deterministic and adaptive methods are from the same works that were also applied under the crossover process. Additionally, as the genetic information of the individuals is changed, these individuals will be replaced with new individuals that reflect those genetic changes. This is possible by an individual creation process that also happens during the `crossover` phase. In this case, the process will create individuals that reflect the mutations that were applied to the individuals. These new individuals will have adjusted fitness scores, according to the changed genes, via the conjunction of the `Pytest` framework and the `Coverage` module.

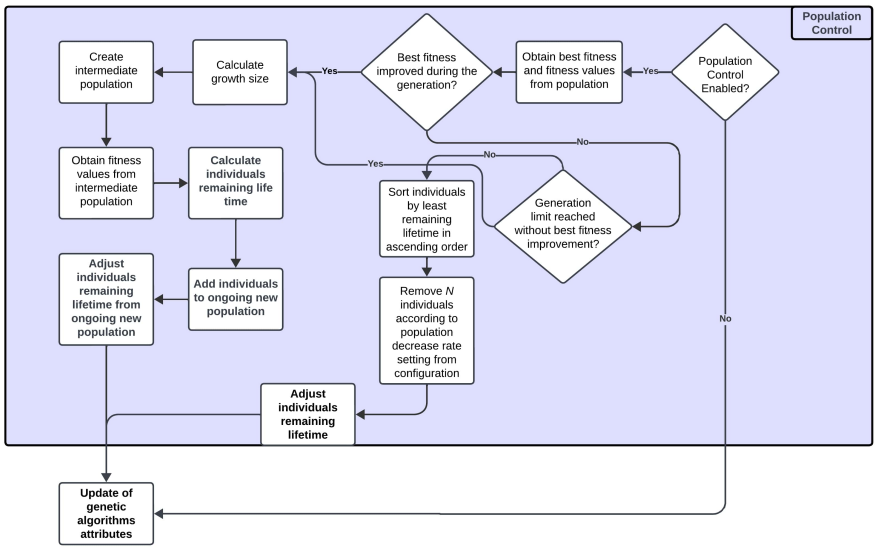


Figure 4.7: Genetic algorithm population control flowchart.

Figure 4.7 shows the `population control` phase of the GA. This is the final execution phase that follows up after the `mutation` phase. This phase can be executed earlier if the adaptive selection method or the self-adaptive crossover method is performed. This phase employs the population control mechanism proposed in the work [48] where the population size is adjusted across the GA run.

After this phase ends, a verification will be made to see if the GA reached any termination criteria. If this happens, the GA will stop executing and will return the data obtained from the generation in JSON format including **number of completed generations**, **population size**, **best fitness seen**, **best fitness scores in each generation**, **average crossover rate**, **average mutation rate** and **execution time**. Additionally, graphs will also be created to illustrate the generations' data in a visual representation. The phenotype and genotype format of the best individual generated during the GA run will also be obtainable. If the GA has not reached termination criteria, then the GA advances into a new generation

and the whole genetic process starts again with an updated population at the `selection` phase.

There is also an important detail to mention during the GA execution route. The new population that is being created cannot be lower in size when compared to the old population. This is a necessary restriction before the GA can proceed to the next generation. As the principal genetic operations (crossover and mutation) generate two offsprings each, totaling two final offspring at the end of these operations, it wouldn't make sense to go into a new generation with only two individuals. For this reason, the new population size can't be lower than the old population size. Old population is considered as the population of the previous generation or the initial population in the first generation of the GA. This specific detail can be viewed in Figure A.4 which represents the complete execution flow of the GAs that were executed during this work.

The reason why the above Figures 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 were simplified from the complete structure presented in Figure A.4 was to provide a general view on the GA execution flow as well as a more focused view on each execution phase of the GA. However, these simplified representations exclude interactions between the phases, so the Figure A.4 is indicated for a complete and detailed representation of the GA execution flow.

4.6.3 Execution Strategy

The GA execution flowchart, represented in Figure A.4, is composed of multiple researches that focused on the optimization of genetic attributes as well as standard operations used in literature. Given this vast amount of genetic operation choices, multiple routes could be taken to execute the GA, however this work aims to automatically create the best possible set of unit tests for a CUT. Considering this aspect, a good execution strategy would be to individually benchmark the available optimizations for each genetic attribute (population, selection, crossover and mutation) and then decide based on the results, which optimization should be used for a final GA regarding each genetic attribute. In this way, an optimized GA could be obtained and provide the best results possible. After the first benchmark, the optimized GA would be compared to a TGA to verify the validity of the optimizations. This was the initial line of thinking, however after the analysis of the researches presented in section 3.2 as well as the first benchmark results, the planned "final" optimized GA would be converted into four different GAs. Additionally, some configurations used for the parameters of the GA would also be adjusted according to the first benchmark results. The reasons as to why four different GAs were executed in the final benchmark as well as why the adjustment of parameters in the configuration was needed are explained thoroughly in section 5.2.

The benchmark themselves, in order to present scientific relevance, were decided to be applied under a decent amount of GAs executions where in both benchmarks the GAs were executed for ten iterations. The performance of an GA is properly evaluated when multiple GA instances, with the same configurations, are executed [5]. According to the description presented in the book "Introduction to Evolutionary Computing" [5] and the experiments performed by other authors in section 3.2 regarding this topic, it was decided to perform ten iterations of executions for the GAs with different configurations for the benchmarks.

4.7 Implementation Challenges

This section presents a set of implementation challenges that emerged during the development of this dissertation. After a proper identification and analysis of the encountered challenges, these were resolved without any major difficulties. The implementation challenges include **partial branch coverage** inconsistencies upon fitness assessment of a generated unit test, the lack of information or ambiguity when it comes to the researches that proposed optimizations for the stochastic nature of the GAs and incompatibility of the range of values defined in the inputs of the fuzzy system for the adaptive crossover and mutation methods.

The first challenge was detected upon the generation of a set of unit tests that presented a score close to the global optima, however it seemed that they could never achieve it. This situation was provoked by the exception detection imposed in the setters of the CUT attributes. The exceptions would stop the test case execution and the CUT methods whose branches covered the affected attributes could not be executed. This was verified in two methods of the CUT and was quickly resolved by removing the detection of the exception in the setters of the affected attributes. This was detected before the final results of the first benchmark, which consecutively forced a new execution of the first benchmark after this issue was fixed.

The second challenge was detected during the implementation of the optimizations proposed in various researches. These challenges were mostly around the understanding of complex mechanics behind the population control, adaptive selection and self-adaptive crossover optimizations. All of these works included mechanics that needed proper testing before being included into the main GA structure or needed various cross-references with other works because of the lack of information (missing proper descriptions on applied genetic operations) or ambiguity in the explanation. The most complex works were the ones presented by Pham, Castellani [49] and Bäck, Eiben and Vaart [47]. These demanded a different execution flow from the TGA route, as these works implemented inherent mechanics that conflicted with the normal GA execution route. This challenge was promptly resolved by adding additional conditions in the implementation of the GA execution flow to consider the situations provoked by these optimizations.

The third challenge was considered to be either an incompatibility behind the `skfuzzy` tool or an inconsistency in the work "Implementation of Evolutionary Fuzzy Systems". The work describes a range of values for the fuzzy system inputs with the corresponding range of values for the fuzzy membership functions. These range of values presented some inconsistencies, where some lower and upper limits of the input and output ranges were defined correctly with the corresponding membership function's lower and upper limits, but some were not. The inconsistent input was the "number of generations of unchanged best fitness" and the inconsistent output was the "mutation rate". The lower and upper limits of the range of values of the input and output were not corresponding to the lower and upper limits of the range of values of the corresponding membership functions. These values were set up as they were presented in the `skfuzzy` tool and upon some testing, some input and output values were out of the lower and upper limits which consecutively produced calculation errors in

the tool. This was resolved by using the lower and upper limits defined in the membership functions.

4.8 Implementation Limitations

Due to time constraints, it was not possible to fully optimize some aspects of the chosen theme's practical development. This section presents the main limitations behind this dissertation implementation:

- **Static metadata**: the metadata was defined statically. This static approach can be a drawback to this dissertation if the implementation needs to be adapted for simpler or more complex CUTs. The metadata was defined specifically for the chosen CUT and there is no defined methodology to extract metadata dynamically from a CUT;

- **Lack of assertion phases**: upon further analysis of a unit test structure and the complex problems this dissertation would face trying to properly determine correct assertions for the generated unit tests, the idea of implementing assertion phases in an automated form was discarded. This is due to the lack of proper advances in this research field as demonstrated in researches [65, 66] where there is a lack of substantial results when it comes to automatically generate test oracles (assertions) for SUTs or the existing methodologies for the generation of test oracles still need a much-needed improvement. The works [67, 68] reached a conclusion where the applied methodologies involving state-of-the-art unit test generation or test oracle generation tools for the Java programming language did not achieve good enough results. As stated in research [67], the generated unit tests did not detect the majority of the faults, where only a small amount of the individual test suites detected a fault [67]. Additionally, another work [68] that focuses on the automated generation of test oracles through the Test Oracle GenerAtion (TOGA) tool [69] verified that the vast majority of generated test oracles failed to compile, and of those that did, only a small portion of them were able to correctly identify faults. Trying to analyze this research field as well as trying to focus on the input generation for the tests themselves would take too much time, which was already limited by the time the implementation of this work could be done. Additionally, because of the lack of solid results regarding the generation of test oracles and the availability of tools for mostly non-dynamic programming languages, the focus of this work was directed to the generation of the arrange and act phases of the unit tests.

4.9 Threats to Validity

This section presents the threats to validity that can be detected upon the experiments performed in this dissertation. This section is divided into the construct, internal, external and conclusion validity sections 4.9.1, 4.9.2, 4.9.3 and 4.9.4 respectively.

4.9.1 Construct Validity

A threat to construct validity is the potential incompleteness of the literature review for the optimization of genetic operations. This aspect concerns the "generalisation of the experiment to concept or theory behind the experiment" [70]. A set of researches were chosen based upon the "relevance" criteria for a desired objective, i.e, some researches for the crossover and mutation optimization were chosen based upon the initial search results in academic websites. There is also a chance that works that implemented mechanisms related to the optimization objective were not considered because the title and abstract description did not mention the optimizations. This situation, however, is mitigated by including the study of several works for genetic operators optimizations in order to properly evaluate which optimization should be chosen for the GA genetic operators.

Another potential threat to construct validity is the fact that there isn't an automated generation of assert phases, as the initial objective behind this work was to fully generate a unit test structure. The reasons to why this was not generated were already mentioned in section 4.8 however this still constitutes a threat to construct validity as "the chosen perspectives...may not be representative or good for scenario-based reading" [70]. This, however, cannot be mitigated as there is a lack of assertion phase in the generated unit tests. In contrast, a huge effort was made to fully optimize the generation of the arrange and act phases of a unit test structure.

4.9.2 Internal Validity

A potential threat to internal validity is the subjective selection process of the GA parameters values. One of the major points behind this work is the benchmark analysis of the genetic operators optimizations. This threat can happen in both benchmarks. There isn't a correct way to properly decide which values the GA parameters should take before the executions. Parameters such as population size, maximum number of generations and maximum number of test cases can be crucial for the generation process. This subjective matter is considered to be a threat to internal validity. It's not possible to determine a GA behavior beforehand and consecutively, there isn't a way to properly define the necessary parameter's values to achieve good enough solutions. This is an existent problem in the stochastic algorithm area as already presented in section 2.3.3.4. The parameter optimization is a threat to the internal validity as it concerns "matters that may affect the independent variable with respect to causality, without the researchers knowledge" [70]. This situation was mitigated by considering an adjustment of the GA parameter values according to the results obtained in the first benchmark. This adjustment was decided based upon the GA behavior from the results graphs by correlating GA literature with the behavior demonstrated in the results of the first benchmark. The analyzed GA literature delves upon genetic operations' behavior, mostly being around the observation of the experiment results with the researches that proposed the optimizations. Despite this adjustment continues to be of a subjective matter, an attempt is made to optimize the overall genetic process.

4.9.3 External Validity

The CUT is statically created before the generation provided by the GAs. This is a threat to external validity, as it concerns the "generalisation of the experiment result to other environments" [70]. If a simpler or a more complex CUT replaces the previous CUT structure, the generation results will not be the same. The generated unit tests were evaluated according to how much branch coverage they could reach in the target CUT. If the CUT is changed to either having more or less conditional branches, the results themselves obtained in the current experiment will likely not be the same after the CUT changes. Additionally, the experiment could have been performed for statement coverage assessment. However, this would produce the same expected behavior, as less or more statements in a new CUT would affect the generation results. According to these observations, other environments will not produce the same results as demonstrated in this experiment. This is not an issue that can be mitigated, as the nature of the stochastic algorithms does not allow it.

4.9.4 Conclusion Validity

The results obtained from the GAs executions can be considered a threat to conclusion validity. A low amount of results can hide the presence of patterns in the data and may not reveal significant outcomes. This situation concerns the "statistical analysis of results and the composition of subjects" [70]. This was mitigated by running the different GAs for ten iterations each, where each one maintained the configuration they had in the first run. This was done for the first benchmark (optimizations of genetic attributes) and the second benchmark (optimized GAs). This execution strategy provides enough data to reveal potential patterns in the results and to demonstrate interesting behaviors by the GAs. Executing the GAs multiple times with the same environment accompanied by statistical measures for the results is a necessary process to have a good estimation of a GA performance [5].

4.10 Conclusion

This chapter delved upon the particularities about the adopted implementation regarding the theme of this dissertation. It presented important aspects such as the individuals' representation scheme, the individual's generation and evaluation in the genetic problem context and which technologies, libraries and materials were used to make this work possible. It also showcases the procedure for the GAs execution and which execution strategy was applied for the GAs while taking into account the benchmarking aspect. Implementation challenges and limitations that were found are also demonstrated, as well as possible threats to validity for this dissertation.

Chapter 5

Results Discussion

5.1 Introduction

This chapter demonstrates the results obtained from the execution of the differently configured GAs in both benchmarks and the corresponding analysis of the generation results. Additionally, an evaluation of one of the best generated test suites from the second benchmark is also performed. This chapter is divided into the following sections:

- **Genetic Algorithms Optimizations**: section 5.2 presents a more complex view about the GA optimizations research, the generation results for each GA attribute optimization and a corresponding analysis of the GAs performance in each genetic attribute. In this section, it is also decided the next set of GAs for the second benchmark;
- **Optimized Genetic Algorithms**: section 5.3 demonstrates the generation results from the optimized GAs and corresponding analysis. Additionally, the best generated unit test from the GAs is also evaluated in terms of unit test structure and quality.

5.2 Genetic Algorithms Optimizations

This section introduces the evaluation of the generations statistics obtained from the ten iterations of executions for the genetic algorithms with different configurations. The algorithms evaluated are the genetic algorithms that have different configurations for each genetic attribute, i.e, genetic algorithms with different configurations for the population, selection, crossover, and mutation operations.
These algorithms are always tested independently, only one optimization is applied per execution. In this way, there is no mixture of optimizations and a fair evaluation is guaranteed where only the optimization at hand is evaluated without any benefits that might come from other optimizations. The ten iterations of executions refer to ten separate runs of the genetic algorithm under a specific configuration. This part of the work executed, in total, 190 genetic algorithms where each of them had a pre-determined configuration for each genetic operation optimization (population, selection, crossover and mutation). The configuration used for the rest of the genetic attributes, that are not the target of optimization, is based upon standard procedures used in traditional genetic algorithms.
During the executions, certain genetic algorithm data was retrieved in order to provide a proper quality assessment of each optimization method applied. These data include: a set of the best fitness values registered during each generation, number of generations until each algorithm reached termination condition and time of execution of each genetic algorithm.

Each data registered has its own relevance for the performance evaluation. Registering the best fitness values from the generations provides a proper assessment on how effective the algorithm was on an objective level [5], i.e, how well the algorithm performed in achieving the best solution possible and how well the creation of the offsprings was during the run. Analyzing the number of generations and the time of execution needed for the genetic algorithm provides contextualization on the algorithm's efficiency [5], i.e, how quick the algorithm was to achieve the best solution possible.
The analysis of fitness values is divided into two points, where the MBF across all runs is evaluated and the mean of the best fitness values in each generation is calculated. The former is self-explanatory, however the latter needs more contextualization. The best fitness value per generation was registered in order to visualize how good the algorithm performs in each generation, obtaining the best individual generated in that generation (not out of all of them, as the MBF metric presents). After the genetic algorithm run is concluded, a mean of the best fitness values in each generation is calculated. This enables a proper view about the genetic algorithm behavior throughout the run and also serves to determine if the algorithm entered a local optimum during the search process. This data is then plotted into different graphs in order to provide enough contextualization about the different genetic algorithms' performance. This will serve as a basis to evaluate each algorithm and provide a conclusion about the generation process. As mentioned previously, this section will evaluate the genetic algorithm performance of each genetic attribute (population, selection, crossover and mutation) according to a set of different configurations, more concretely optimizations, applied to each attribute.

5.2.1 Population Optimization

This section focuses on the performance evaluation of two different genetic algorithms that delve upon the population attribute. The applied configurations for the population involve:

- **No population control**: the genetic algorithm with this configuration does not perform any kind of action towards the management of the population size. The predefined size for the population maintains constant throughout the execution of the genetic algorithm. The genetic algorithm with no population control will be defined as **"Population False"**;

- **Population control**: this genetic algorithm employs the population control mechanism proposed by the Rajakumar and George in the work "APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48]. It consists in managing the population size, increasing it or decreasing it with the RLT mechanism and feedback obtained from the genetic algorithm execution. The global scope of this mechanism was already explained in section 3.2, however some further explanation is needed for some aspects.

  The authors applied the mechanism for minimization problems, however according to the scope of this dissertation, where the automated generation of unit tests is sought after, it requires changing this mechanism to be applied on **maximization problems**. The generation of unit tests is focused on finding tests that maximize code coverage, and

as such, this heavily influences the **RLT mathematical formulation**. The changed formula is presented in equation 5.1 [44]

$$RLT(i) = \begin{cases} MinLT + n\frac{fitness(i)-MinFit}{AvgFit-MinFit}, \text{if } AvgFit \geq fitness(i) \\ \\ \frac{1}{2}(MinLT + MaxLT) + n\frac{fitness(i)-AvgFit}{MaxFit-AvgFit}, \text{if } AvgFit < fitness(i) \end{cases} \tag{5.1}$$

where $n = \frac{MaxLT-MinLT}{2}$ and $MaxLT$ and $MinLT$ are the maximum and minimum lifetimes of the individual, respectively. The measures $AvgFit$, $MaxFit$, $MinFit$ represent the average, maximum and minimum fitness of the population respectively. This formulation, targeted for maximization problems, was proposed by the authors Arabas, Michalewicz and Mulawka [44]. The RLT calculation was also proposed by these authors. Additionally, the formula used to grow the population size is presented in equation 5.2

$$G = \alpha \times (I^{max} - I) \times \frac{F^{newbest} - F^{oldbest}}{F^{initialbest}} \tag{5.2}$$

where α is a random value in the interval $[0, 1]$, I^{max}, I, $F^{newbest}$, $F^{oldbest}$, $F^{initialbest}$ and G represent the maximum number of generations, current generation number, the best fitness value of current generation, the best fitness value of previous generation, initial best fitness value and growth size respectively. The method proposed by the authors also decreases the population by a predefined rate $D\%$ when there is no improvement of the best fitness value in each generation before some x amount of generations.

Different works presented in section 3.2 used the RLT mathematical formulation, such as [47, 48]. The proposed method in the work **"APOGA: An Adaptive Population Pool Size based Genetic Algorithm"** was the one selected out of the works [44, 45, 47].

The `PRoFIGA` algorithm, proposed in the work [45], obtained better results than the `GAVaPS` [44] in terms of speed and efficiency when generating individuals. The `APGA` algorithm [47] was the better algorithm when compared to the `PRoFIGA` algorithm. The reason the **`APOGA` algorithm was chosen**, for this work, instead of the `APGA` one was because of the **additional features that can enable more control over the population size**, such as the **growth** or **decrease** of population size **according to the fitness variation during the genetic algorithm execution**. The `APGA` algorithm only grows and decreases the population according to the RLT formulation, not considering any type of additional feedback obtained during the execution of the genetic algorithm.

The genetic algorithm with population control will be defined as **"Population True"**. The values used for $MinLT$, $MaxLT$, the decrease rate $D\%$ and the maximum number

of generations for best fitness stagnation were 1, 10, 0.2 and 2 respectively. The α value was set statically with the value 0.7. The authors also pre-defined a value for the α variable.

The Figure 5.1 and Table 5.1 present the results obtained from the executions of the genetic algorithms for the population attribute. The analysis of the generation results for each population attribute is presented below regarding each evaluation metric:

- **Mean best fitness**: the population method's MBF metric, previously discussed in section 2.3.3.5, can be visualized in Figure 5.1a and the specific scores can be visualized in Table 5.1a.

  The analysis of Figure 5.1a suggests that the **"Population True"** algorithm (population control optimization), obtained **better MBF** than the "Population False" algorithm (static population size). The anaylsis of the scores in Table 5.1a demonstrate that the "Population True" algorithm obtained a MBF value of 0.91490 and the "Population False" algorithm obtained a MBF value of 0.84761.

  Additionally, for each population algorithm, the standard deviation of the data was calculated and presented in both Figure 5.1a and Table 5.1a. The **"Population True"** algorithm, who has a standard deviation value of 0.02324, **is more robust** than the "Population False" algorithm, who has a standard deviation value of 0.02336 despite the difference being very minimal;

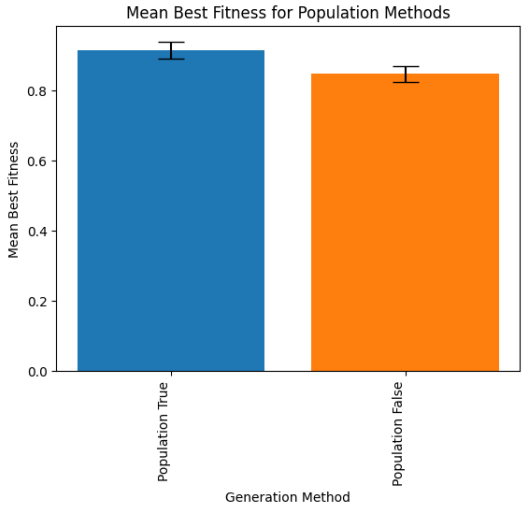
- **Mean of the best fitness values per generations**: apart from the MBF metric, it was decided to also retrieve data about the best fitness score per generations for each population algorithm. The analysis of Figure 5.1b suggests that the **"Population True"** algorithm **was superior** when compared to the "Population False" algorithm. The mean of the best fitness scores per generations for each method can be visualized in Table 5.1b where 0.90510 and 0.79238 scores correspond to the "Population True" and "Population False" algorithms respectively. The fitness scores also have a respective standard deviation value, being 0.01881 and 0.04969 for the "Population True" and "Population False" algorithms respectively. The **former method seems to be more robust** than the latter;

- **Mean number of generations**: as for the number of generations, each method obtained distinct values. The "Population True" algorithm was executed, in average, for about 36[1] generations. The "Population False" algorithm was executed, in average, for about 40[1] generations. By visualizing Figure 5.1c and Table 5.1c the **"Population True"** algorithm **executed fewer generations** about 36[1] generations when compared to the "Population False" algorithm which executed around 40[1] generations;

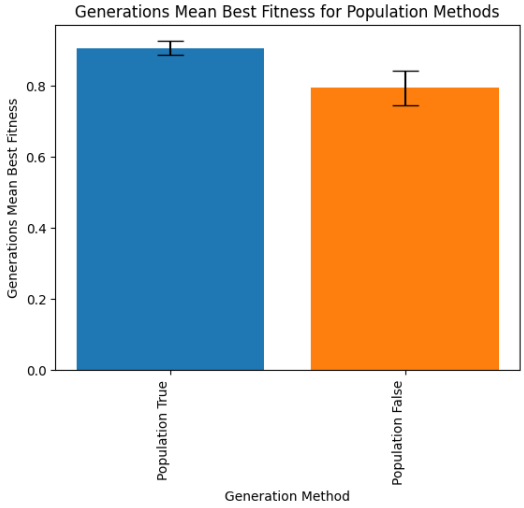
- **Average execution time**: both algorithms were also evaluated in terms of execution time. The **"Population True"** algorithm was **vastly superior** relatively to the

[1] Value rounded to the nearest integer.

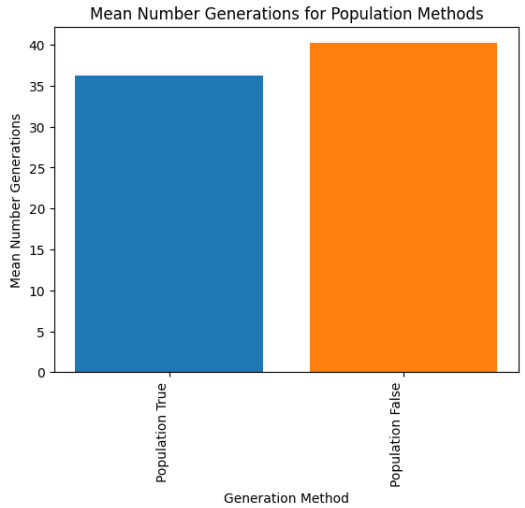
"Population False" algorithm, being the **fastest algorithm to execute**. The difference is notable in both visual representation, shown in Figure 5.1d, and in numerical representation, shown in Table 5.1d. The "Population True" algorithm was executed, in average, for about 467[1] seconds, whereas the "Population False" algorithm was executed, in average, for about 889[1] seconds.



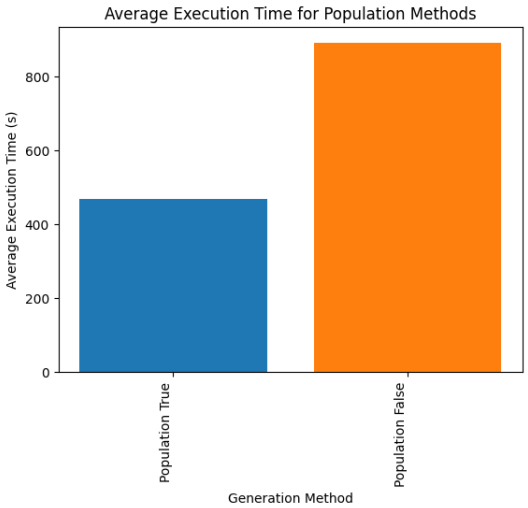
(a) Mean best fitness value for each population method with respective standard deviation.



(b) Mean of the best fitness values per generations for each population method with respective standard deviation.



(c) Mean number of generations taken by each population method.



(d) Average execution time (in seconds) for each population method.

Figure 5.1: Histograms of generation statistics for each population method execution.

The results presented above generally suggest that the **population control** algorithm, adapted from the work "APOGA: An Adaptive Population Pool Size Based Genetic Algorithm" [48], was the best algorithm in all the performance metrics.

A simple explanation can be described about the possible behavior performed by the **"Population True"** algorithm. The algorithm grows and decreases the population size according to the RLT mechanic (attribution of a lifetime expectancy to an individual according to

	Population True	Population False
MBF	**0.91490**	0.84761
Stdev	**0.02324**	0.02336

(a) Mean best fitness scores for each population method.

	Population True	Population False
Generations Mean Best Fitness	**0.90510**	0.79238
Stdev	**0.01881**	0.04969

(b) Mean of the best fitness scores per generations for each population method.

	Population True	Population False
Mean Number of Generations	**36.2**	40.2

(c) Mean number of generations taken by each population method.

	Population True	Population False
Average Execution Time	**466.93199s**	889.04933s

(d) Average execution time (in seconds) for each population method.

Table 5.1: Generation statistics scores for each population method.

its fitness score) and additionally performs these operations according to feedback from the population (the best fitness score variation).
The reason why the results presented in Figures 5.1a, 5.1b, 5.1d and Tables 5.1a, 5.1b, 5.1d are so noticeable is because of the impact of the initial decreasing operation that happens because of low RLT values. If an initial population is composed mostly of low fitness individuals, the algorithm will allocate the lowest RLT value possible to them and decrease the RLT score in each generation. All the individuals that reach RLT values below one are instantly removed from the population pool [48]. For the executions of this algorithm, the minimal RLT value was one, so a drastic reduction in population size in the initial generations is to be expected. In this specific situation, the population will be reduced in terms of size, but will contain stronger individuals. This situation, however, can bring good and bad outcomes:

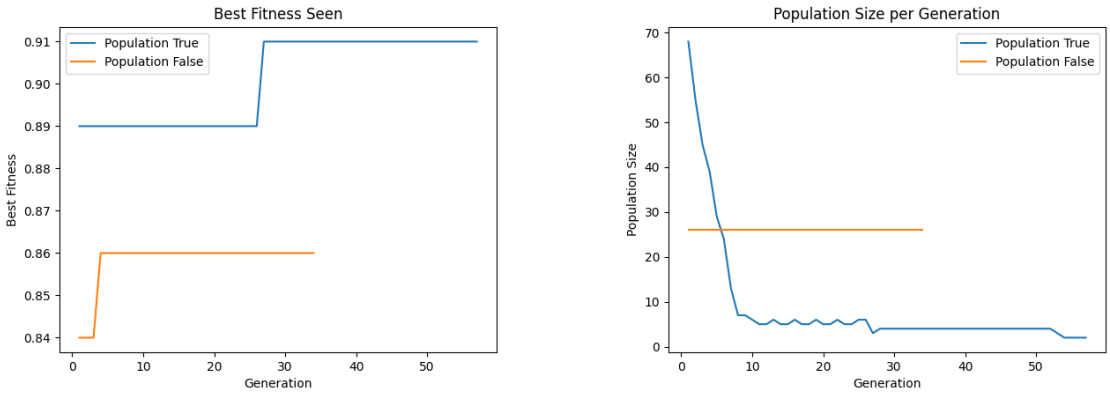
- **High fitness population**: this brings forth the initial objective behind these genetic algorithms for maximization problems. A higher percentage of high fitness individuals is always desired as the probability of finding a good enough solution increases;

- **Lack of genetic variety**: having a strong population pool may not be always the best situation possible. If these individuals are of the same structure, i.e, composed by the same genetic information, there is a low chance that new individuals generated from these high fitness individuals will bring forth any unseen genetic information. This affects immensely the quality of the solutions, making the genetic algorithm fall under a local optima, similar to what happens in the roulette wheel selection described in the section 2.3.3.3.4 where “predominant individuals...will introduce bias to the initial search...This leads to premature convergence and loss of diversity”.

The vast reduction in population size provokes the fast execution times, as fewer fitness evaluations are applied. The population size decreases until the best seen fitness score improves during the generations or the best seen fitness score does not change until x amount of generations [48]. The reason why this algorithm achieves such high fitness scores can be explained by two phenomenons:

- **Initial high fitness scores**: in the initial generations, the majority of the individuals could have high fitness scores. During the rest of the genetic algorithm execution, the algorithm may not achieve better fitness scores, which can lead to search stagnation. This is probably what happened by visualizing Figure 5.1c and Table 5.1c where, in average, the number of generations, across all executions, is not that different as the run limit in the configuration was defined with a maximum amount of 30 generations. The number of generations of both algorithms were above the limit, which means that during some executions, a higher fitness individual was created, which consecutively enabled the execution to go longer. The second phenomenon that provoked this situation is explained in the next bullet point;
- **Growth impact**: the mechanism, implemented by this algorithm, grows the population when a new best fitness score is obtained or when the best fitness score is not improved during x amount of generations [48]. Another fact to add up is that in the first generation, the best fitness individual is the first individual in the population (initialization). The growth mechanic can be activated more often in the first generation, which will introduce a high amount of individuals (depending on the initial population size and the α variable value in the growth formula). This sudden growth can introduce high fitness individuals, potentially limiting the search very early.

In order to further enhance the explanation of the two last bullet points, the Figure 5.2 shows an iteration example out of the ten execution iterations for the "Population True" algorithm. Only one iteration is shown, as it is impossible to perform a mean of these variations, as each algorithm iteration finished executing with different number of generations (imposed by the termination criteria).
The Figure 5.2a demonstrates the best fitness variation of the population across the generations, while Figure 5.2b presents the population size variation across the generations. The presented behavior in both figures goes in line with the explanation in the last two bullet points. For the "Population True" algorithm, the population starts with a high fitness value of 89% and maintains it until the 26^{th} generation. In this generation, a higher fitness individual is achieved due to the growth mechanic embedded in the algorithm and consecutively the execution goes longer. The growth occurs at generation 25 as shown in Figure 5.2b and consecutively a higher fitness individual is created at the same instant as shown in Figure 5.2a. Additionally, a high reduction in population size is also shown between generations 1 and 7 showcasing the shrink operation of the adaptive method as well as various growth attempts to achieve a higher fitness score occurring between generations. This method, however, shows in Figure 5.2b that the **growth and shrink operations need more tuning** as there is a **drastic reduction and consecutively a minimal growth** between generations 10 and



(a) Best fitness value variation for each population method.

(b) Population size variation for each population method.

Figure 5.2: Impact of the adaptive population size method.

25, **restricting too much the availability of genetic variety** in the population. One interesting fact to also point out is that the population size was initially set up to be 25 individuals, however due to the growth mechanic the population size increased by 43 individuals totaling around 68 individuals in the first generation.
Based upon the above observations, the **"Population True"** method that employs the population size adaptation, proposed by the work "APOGA: An Adaptive Population Pool Size based Genetic Algorithm" [48], **will be selected to apply for the final algorithm configuration**. This decision comes from its notorious advantage regarding execution time (being the fastest to execute) and population size variance impact. This size variation is better than static population size, as it can add new genetic information to the population (growth) or remove unfit individuals (shrink) from the population, vastly reducing the algorithm execution time.

5.2.2 Selection Optimization

This section delves upon the performance evaluation of a set of selection methods applied to genetic algorithms. These methods are composed of four different standard selection methods and one optimization method applied to this genetic operator. The configurations used for the selection operator are presented:

- **Standard selection methods**: this set includes standard selection methods used in traditional genetic algorithms. The chosen methods were already introduced in section 2.3.3.3.4 being the random, roulette wheel, rank and tournament selection methods;
- **Adaptive selection scheme**: a proposal about an adaptive selection scheme, presented by the work "Adaptive selection routine for evolutionary algorithms" [49]. The general scope of this work was already explained in section 3.2, consisting of a selection scheme that organizes individuals between two structures (operation aided with a noise perturbation) with a specific mating order. This selection scheme aims to control the selection pressure during the generations [49]. This adaptive selection scheme, however,

needs further explanation regarding the inherent mechanics. This scheme is composed of the following phases:

– **Individual selection**: the chosen individuals need to be subjected to a noise perturbation, calculated by the following equation 5.3 [49]

$$m(i) = f(i) + (f_{max} - \mu_f) \cdot rand \tag{5.3}$$

where f_{max} and μ_f correspond to the maximum and average fitness of the population, and $rand$ is a random real number in the interval $[0, 1]$. The mating chance and the fitness of the individual is represented by $m(i)$ and $f(i)$ respectively. This equation is used to calculate the individual's mating chance [49], and it is applied to every individual. This mating chance adjusts the selection pressure by using an estimation of the fitness variation throughout the generations.

In early generations, the fitness variation is likely to be large between the individuals, which can induce early convergence [49]. This undesired situation is controlled by the mating chance equation, represented in equation 5.3, with the **subtraction between the maximum and average fitness**, restricting these high fitness individuals from dominating the population pool. As the genetic algorithm advances towards the final generations, according to the selection pressure rationale [40], the population diversity should gradually reduce. In this situation, the search should be more on the exploitative side because the population is mostly composed of already high fitness individuals. This stage of the algorithm is also taken into account in the equation 5.3 where the difference $f_{max} - \upsilon_f$ is low, and therefore the mating chance won't have any impact at all. Additionally, the authors also limit the selection of the same individuals up to two times to also avoid any chance of early convergence [49].

After the mating chance calculation, the individuals are ranked according to the mating chance and the best half is inserted into a temporary list. The above operation is repeated once more, where the best half is inserted into another temporary list. In total, there are two different lists that contain individuals with high mating chance values. The next step is how the mating procedure influences the selection pressure. This step is explained in the bullet point below;

– **Mating order impact**: upon the organization of individuals into lists, an ordered mating operation is performed. The order in the mating between individuals is justified by the authors as essential for the "gradual improvement of the average population fitness, preventing...the population...premature convergence" [49]. The individuals from both lists are mated between each other in opposite directions, i.e, the first individual from the first list mates with the last individual from the second list, the second individual from the first list mates with the second to last individual from the second list, and so on. Ordering the mating operation in this form, enables the mating between high and low fitness individuals, contributing

to the prevention of early convergence [49].

This selection method was the one selected from the works proposed in section 3.2. The other work proposed by Jebari [50] was not considered due to the heavy computational effort needed, as each selection operator presented by the author would be executed in each generation of the genetic algorithm run. Additionally, it would add redundancy for the general execution for the selection attribute, as this dissertation is already comparing a set of standard selection methods individually. Despite an extensive search, no additional relevant works for the optimization of the selection operator were found, and consequently this part of the work is not as comprehensive as the rest of the genetic operators optimizations.
The Figure 5.3 and Table 5.2 demonstrate the generation results obtained from a series of executions for the selection attribute optimization. These runs were evaluated according to different topics:

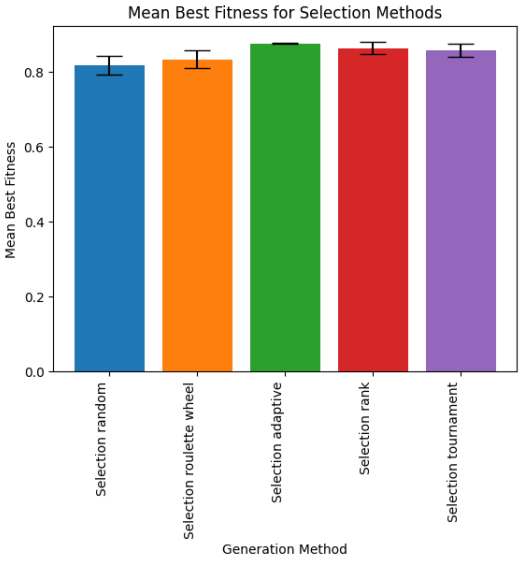
- **Mean best fitness**: the selection methods MBF metric can be visualized in Figure 5.3a and the specific scores in Table 5.2a. The analysis of Figure 5.3a suggests that the **adaptive selection method was the better alternative** when compared to the other applied methods. The Table 5.2a presents the scores obtained for each method relative to the MBF metric, as well as the respective standard deviation of the recorded values. The analysis of scores indicates that the **adaptive selection method obtained the highest MBF** with a score of 0.87661 and the lowest standard deviation value with a score of 0.00104 indicating it is the **most robust method**;

- **Mean of the best fitness values per generations**: the selection methods mean of the best fitness values per generations can be visualized in Figure 5.3b and the respective scores in Table 5.2b. Figure 5.3b indicates that the **adaptive selection method was better** than his counterparts in achieving a **higher mean of the best fitness scores per generations**, as well as **being the most robust**. Scores for each selection method can be visualized in Table 5.2b where the **adaptive selection method** obtained the **highest mean of the best fitness scores per generations** with a score of 0.86845. In respect to the **standard deviation**, the **adaptive selection method** outperformed the other methods by a good margin, obtaining a value of 0.00823;

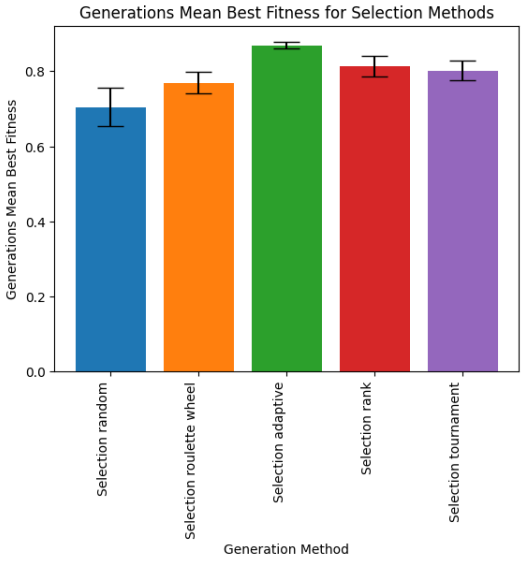
- **Mean number of generations**: the mean number of generations needed for each selection method was registered. The visualization of Figure 5.3c shows similar values for the random, adaptive, rank and tournament selection methods however by visualizing Table 5.2c it is possible to conclude that the **random selection method** was the **best one**, totaling 34^1 generations. This number of generations taken by the random selection method is pretty irrelevant, as the other methods, except for the roulette wheel selection method, also took around 34^1 generations to achieve a solution;

- **Average execution time**: the average execution time was recorded for each selection method. The execution time, in seconds, per each selection method is demonstrated in Figure 5.3d where all the methods, except for the adaptive selection method, took very similar times to execute. The visualization of Table 5.2d suggests that the **random**

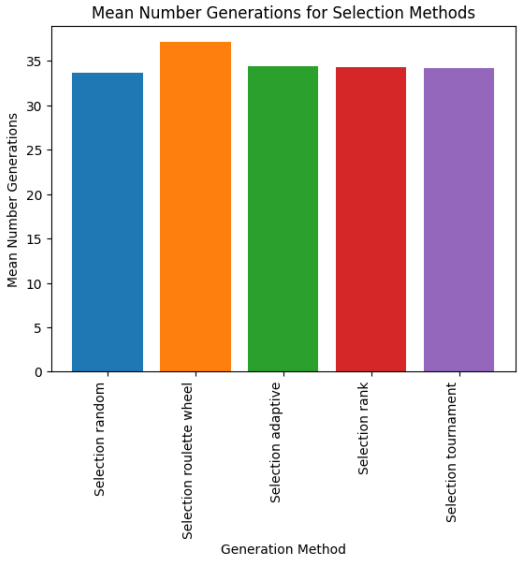
selection method was the one that **took the least time** to finish the search, taking about 705[1] seconds.



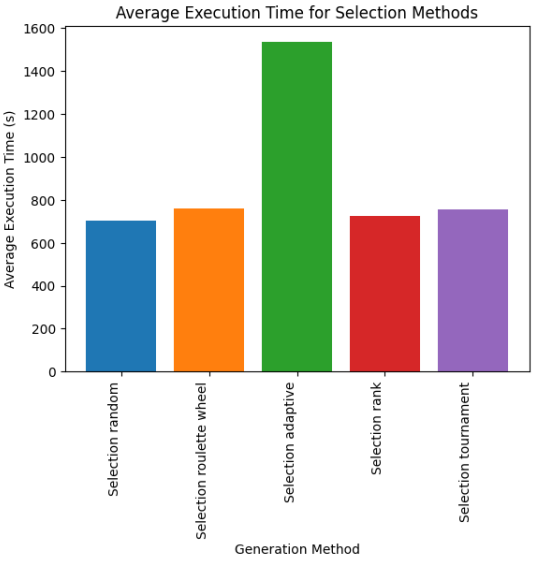
(a) Mean best fitness value for each selection method with respective standard deviation.



(b) Mean of the best fitness values per generations for each selection method with respective standard deviation.



(c) Mean number of generations taken by each selection method.



(d) Average execution time (in seconds) for each selection method.

Figure 5.3: Histograms of generation statistics for each selection method execution.

The results presented in Figure 5.3 and Table 5.2 suggest that the **adaptive selection method**, based on the work "Adaptive selection routine for evolutionary algorithms" [49], was the better algorithm overall. This conclusion mostly comes from the fitness results obtained from the executions.

The **adaptive selection method** obtained **the best MBF value** out of all the other methods, however, as one can see in Table 5.2a, the difference to the rank and tournament

	Random	Roulette Wheel	Adaptive	Rank	Tournament
MBF	0.81834	0.83462	**0.87661**	0.86384	0.85844
Stdev	0.02467	0.02385	**0.00104**	0.01601	0.01781

(a) Mean best fitness scores for each selection method.

	Random	Roulette Wheel	Adaptive	Rank	Tournament
Generations Mean Best Fitness	0.70400	0.76883	**0.86845**	0.81267	0.80126
Stdev	0.05146	0.02882	**0.00823**	0.02694	0.02557

(b) Mean of the best fitness scores per generations for each selection method.

	Random	Roulette Wheel	Adaptive	Rank	Tournament
Mean Number of Generations	**33.7**	37.1	34.4	34.3	34.2

(c) Mean number of generations taken by each selection method.

	Random	Roulette Wheel	Adaptive	Rank	Tournament
Average Execution Time	**704.51696s**	757.53112s	1531.18063s	724.44612s	754.96541s

(d) Average execution time (in seconds) for each selection method.

Table 5.2: Generation statistics scores for each selection method.

selection methods is small, only two hundredths of a decimal point. These scores, generally, suggest that almost all methods are somewhat effective, however this can be deceiving as the initial population might contain individuals of high fitness which translates into a higher chance for them to be selected, and they keep this MBF metric at such a high value. In this case, the Table 5.2b is of great value, as it proves how consistent an algorithm is in selecting good individuals to generate offsprings from. By observing the results in Table 5.2b one can see that it disputes the results obtained in Table 5.2a as the random and roulette wheel selection methods didn't obtain, in average, good fitness values throughout the generations however they still obtained decent MBF scores probably due to high fitness individuals in the initial population. The random and roulette wheel selection methods show the most discrepancy when comparing to the MBF results in table 5.3a while the adaptive selection method maintains, in average, close values to its MBF score.

The **adaptive method** proved to be the **best regarding the mean of the best fitness scores per generations**, obtaining a score of 0.86845. In terms of **algorithm robustness**, the **adaptive method is the best out of all of them**, which indicates it **had the best consistency when selecting good individuals** as the variation of data is very minimal. One interesting result can be observed in Tables 5.2c and 5.2d where the **number of generations for the adaptive selection method**, 34^1 generations in average, is **practically the same as the rank selection method** however the **latter method took considerably less time to execute** when compared to the former method. This situation is due to an interesting behavior presented by the adaptive method, proposed by Pham and Castellani [49], when in presence of populations of odd size. This behavior can be observed in Figure 5.4.

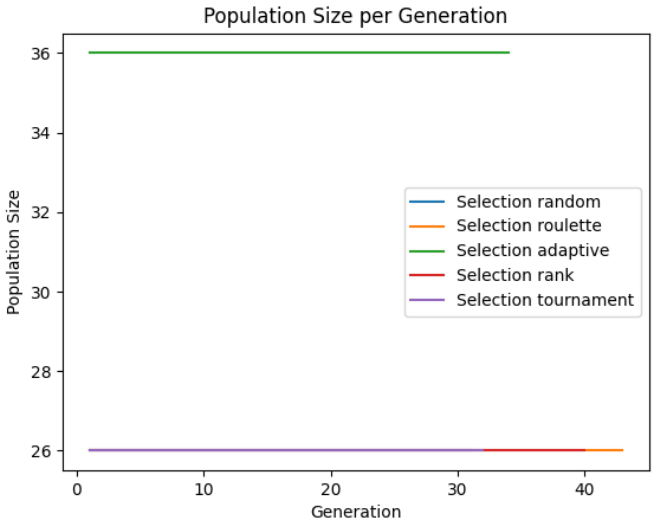


Figure 5.4: Population size variation for each selection method.

The population size in the configuration was of 25 individuals and analyzing Figure 5.4, who demonstrates the results after the first generation onwards, it suggests that the population of the adaptive selection method increased to 36 individuals as the other methods remained as 26. The additional individual in the other methods is explained by an implementation choice, as the **new population size created in each generation can't be lower than the old population size**. As the population size is an odd number and the offspring addition to the new population occurs in pairs, an additional pair is added when the new population size reaches 24 individuals because this number is not lower than the size of the original population. After the new population size is higher than the old population size, the current generation ends and the genetic process continues with the new population. This line of thought goes inline with the behavior of the adaptive selection method as well. The two lists formed by the adaptive method have half the size of the population, i.e, each list is composed of 12 individuals (round to the nearest unit). The adaptive selection method will mate the individuals from both lists in a specific order, generating 12 offsprings to add them into the new population. An additional iteration is performed when the new population size reaches 24 individuals, adding another 12 individuals into the new population. In the end of the first generation, the new population is constituted by 36 individuals. As the new number of individuals is an even number, for future generations, the adaptive selection method will not cause any type of increase as it can be seen in Figure 5.4.

Resuming the point above, as the number of individuals increased per generation, so the time of execution is expected to increase as well, which explains the discrepancy between the adaptive and rank selection methods in Tables 5.2c and 5.2d regarding the mean number of generations and average execution time for the two selection methods.

Based upon the above observations, the **adaptive selection method**, proposed by the work "Adaptive selection routine for evolutionary algorithms" [49], was the best selection method out of all ten iterations of executions. This is justified by the MBF score it obtained, as well as the **robustness** of the method and the **number of generations** it took to achieve a

solution. This selection method when faced with populations of odd size increases the initial population size and adds more genetic information to the population, which is a significant advantage to consider. Another point to mention, regarding this population increase, is that the adaptive selection method took longer than the rank selection method because of the additional individuals it had. If this population growth didn't happen, it might have had similar or even lesser execution time compared to the rank selection method.
Despite being the best method for the selection operation, unfortunately it **cannot be chosen for the final algorithm configuration**. The work where this method was proposed [49], does not mention any type of mutation method applied onto the genetic process. Additionally, the crossover operation is embedded into the selection scheme, restricting the application of any type of crossover method. The **rank selection method will be chosen** instead as it was the second-best method when compared to the other selection methods in terms of MBF score, the mean best fitness score per generations with respective standard deviation values and time of execution. This adaptive selection method, however, will be used in the final benchmark alongside the optimized GA and a standard GA.

5.2.3 Crossover Optimization

This section demonstrates an evaluation of the generation results obtained from the GAs focused on the crossover genetic operator. The executed GAs include a standard method as well as optimizations applied to the recombination of individuals. The crossover methods used for this evaluation were:

- **Standard method**: the uniform crossover method was the chosen standard method. This method was already explained thoroughly in section 2.3.3.3.5;

- **Deterministic crossover rate adjustment**: a deterministic crossover rate adjustment method was proposed in the work "Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach" [51]. This method consists in dynamically adjusting the crossover rate throughout the generations according to deterministic rules [51]. The essence of the crossover rate adjustment was already explained in section 3.2, however there are still mechanics to explain regarding the dynamic rate adjustment.

  The work implemented two different instances of crossover rate adjustment. These instances are deterministic rules that change the rate according to the current generation number of the GA and the maximum number of generations defined for it. The crossover rate can be adjusted either incrementally or decrementally throughout the generations. The two deterministic crossover rate adjustment techniques are the Dynamic Increasing of Low Crossover Ratio (ILC) and Dynamic Decreasing of High Crossover Ratio (DHC). The ILC technique is applied with the following equation 5.4 [51]

$$CR = \frac{LG}{Gn} \tag{5.4}$$

where CR is the adjusted crossover rate, LG is the number of generation level (current generation number) and Gn is the maximum number of generations. This technique aims to gradually increase the crossover rate from 0% to 100% during the execution of the GA. The DHC technique is represented by the following formula 5.5 [51]

$$CR = 1 - (\frac{LG}{Gn}) \tag{5.5}$$

where CR is the adjusted crossover rate, LG is the number of generation level (current generation number) and Gn is the maximum number of generations. The formula variables are exactly the same as the ILC technique, however the purpose behind it changes, as the DHC technique aims to gradually decrease the crossover rate from 100% to 0% during the execution of the GA.

The authors justify the implementation of these two techniques according to the importance of parameter setting in GAs to achieve optimality [51]. These two formulas are highly dependent on the generation level during the execution and as they do not evaluate any type of feedback from the generation process, they are considered as deterministic methodologies. One important point to mention is that **these mechanics may not be fully explored** as the configuration for the GAs maximum generations for fitness stagnation is set to 30 generations. The GAs that stagnate early or even from the beginning, regarding the best fitness improvement, won't be able to fully utilize these mechanics as the generation level might not surpass even half of the maximum number of generations which was set to 100 generations.

An extensive search was made in order to gather another relevant works that also presented deterministic approaches to adjust crossover rates, however no other relevant works on this topic were found and consecutively this is the only deterministic crossover method presented in section 3.2 as well as the only deterministic crossover method applied in this work;

- **Adaptive crossover rate adjustment**: the work "Implementation of Evolutionary Fuzzy Systems" proposed by Shi, Eberhart and Chen [52] presented an adaptive methodology to adjust crossover rates using a fuzzy system. The original purpose of this work was not to propose a fuzzy system for the genetic operators rates adjustment, however, the authors presented a fuzzy system variant that enables an adaptive adjustment of the crossover rates that evaluates feedback from the GA generations and outputs the adjusted rate accordingly. The general scope of this implementation, alongside the fuzzy system structure, was already explained in section 3.2, however, there is still some room for an enhanced explanation regarding the system's concept and constitution.

  A fuzzy expert system is of a fuzzy logic nature, which include fuzzy sets that represent the domain with different degrees of membership. These degrees of membership are determined with membership functions associated to the fuzzy sets. These functions

return an inference about a specific input variable, determining if the variable is of a certain nature or degree within the range of $[0, 1]$. This system include fuzzy sets that represent the domain with different degrees of membership. Each input variable is assigned a specific fuzzy set using membership functions in a process called fuzzification. Following this process, the input variables are mapped into linguistic values, resulting in an output fuzzy set [71]. These values are determined using approximate reasoning (inference) with expert knowledge (set of rules determined by experts). This process enables a correct approximation of which variable is more likely to be part of a specific fuzzy set. The output of the fuzzy expert system is then obtained by a process called defuzzification which assigns representative crisp data to the resultant output fuzzy set [71].

The inference engine used in the authors work is the Mamdani inference system [72] where the output provided by each rule is a fuzzy set. The Mamdani fuzzy rule in a fuzzy expert system [72] is defined by equation 5.6

$$If\ x_1\ is\ S_1,\ and\ x_2\ is\ S_2, ...,\ x_k\ is\ S_k,\ then\ y_1\ is\ T_1, ...,\ and\ y_k\ is\ T_k \tag{5.6}$$

where y_k is an inferred output (consequent) associated to a fuzzy output set T_k (after approximate reasoning) and each x_k is an input variable (antecedent) associated to a fuzzy set S_k (before approximate reasoning).

The input variables used in this system are feedback obtained from the GA generations such as: population's best fitness in the range $[0, 1]$, number of generations of unchanged best fitness in the range $[0, 12]$ and variance of population fitness in the range $[0.0, 0.2]$. The output variables of this system are the adjusted crossover and mutation rates. The output crossover rate is in the range $[0.48, 0.83]$. The input and output variables are associated to **low**, **medium** and **high** fuzzy sets according to specific intervals from the range of values mentioned above for each variable.

The implemented fuzzy system takes the input variables, where each of them is assigned a specific fuzzy set. After this process, the variables are mapped onto output fuzzy sets that are followed by a defuzzification process that assigns crisp data to the resultant output fuzzy set, i.e, the adjusted crossover rate is returned;

- **Self-adaptive crossover**: a self-adaptive crossover rate adjustment method, proposed by Bäck, Eiben and Vaart [47], was implemented in this work. It employs a new mechanism to control the recombination process by analyzing self-encoded crossover rates in the individuals and deciding which individuals should mate or advance according to these rates. The general view about this research was already explained in section 3.2, however the explanation about the recombination process, according to the self-encoded rates, needs to be elaborated.

  The self-encoded crossover rate is randomly chosen in the interval $[0, 1]$ and encoded into the individuals' representation scheme. After an individual is selected in the selection

process, a random number r in the range $[0, 1]$ is compared with the encoded crossover rate p_c of the individual. After this point, three instances, regarding the ordering of the recombination, are explored:

- If r is lower than p_c then the individual is ready for the crossover operation. If both parents have their p_c higher than r then a new child is created by crossover being it followed by the next in order set of genetic operators (mutation and addition to population) [47];
- If r is higher than p_c then the individual will not undergo crossover and will suffer a mutation to create one child. This child will undergo a mutation process and survivor selection immediately [47];
- If both parents do not have their p_c value higher than r then two children are created with mutation only. If one parent has its p_c value higher than r and the other parent does not, then the latter suffers mutation to create a child which is inserted into the population immediately [47]. The other parent is kept and in the next selection process only one parent is picked.

The self-adaptive method generally consists in using the encoded crossover rate of every individual and compare it to a random value to determine if the individuals will undergo specific genetic operations. These genetic operations are then decided based upon in the individuals' potential to mate.

The **adaptive method** proposed in the work "Implementation of evolutionary fuzzy systems" [52] **was the chosen methodology to implement** instead of the adaptive method proposed in the work "Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms" [53]. This decision comes from the additional generation's feedback that the research [52] considers. The work proposed by Srinivas and Patnaik [53] only consider the fitness variation during the generations, focusing largely on convergence behavior during the GA execution. The work by Shi, Eberhart and Chen [52] also considers this aspect as well as two additional aspects such as the best fitness score and the number of generations without the best fitness improvement. These additional considerations can make a solid impact on the overall crossover rate adjustment.
The self-adaptive crossover methodologies were also scarce during the investigation, as there were a variety of works that only presented adaptive strategies on adjusting the crossover rate or on the whole recombination process without any type of proposal for self-adaptation. There were few works that presented self-adaptation methodologies, however a major part of them could not be reproduced by ambiguous information or lack of it regarding the development process.
The Figure 5.5 and Table 5.3 present the results obtained from the different GAs in the ten execution iterations regarding the crossover operation. These executions were analyzed under different metrics:

- **Mean best fitness**: the results obtained for the mean best fitness can be visualized in Figure 5.5a and Table 5.3a. The analysis of Figure 5.5a shows that the self-adaptive

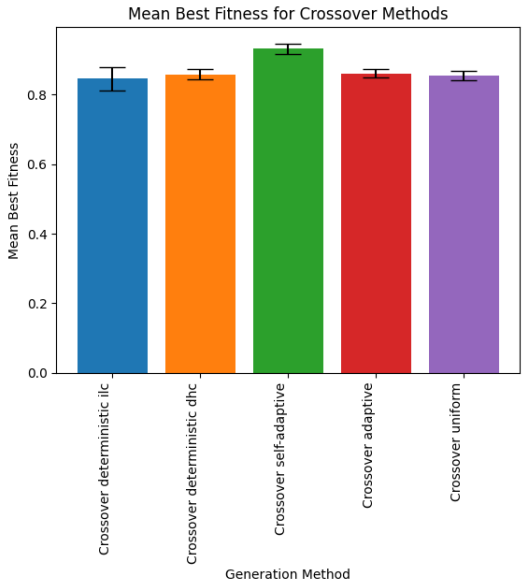
method was the best one out of all the other methods. The scores each method obtained can be visualized in Table 5.3a where it can be observed that the **self-adaptive crossover method** obtained a mean best fitness score of 0.93163 completely **surpassing the remaining methods**. As for the **robustness**, the **adaptive method was the best one** but with a very slight margin when compared to the deterministic DHC, self-adaptive and uniform crossover methods, differing by only thousandths of a decimal point;

- **Mean of the best fitness values per generations**: the mean of the best fitness values per generations can be observed for all methods in the Figure 5.5b and in Table 5.3b. The analysis of Figure 5.5b suggests that the self-adaptive crossover method was the best method when obtaining the best fitness solutions in the generations. The scores can be visualized in Table 5.3b where it can be noticed that the **self-adaptive crossover was the best method** achieving a mean of the best fitness values per generations of 0.89710, however it was not the most robust one as the **uniform crossover obtained the lowest standard deviation value** of 0.01941;

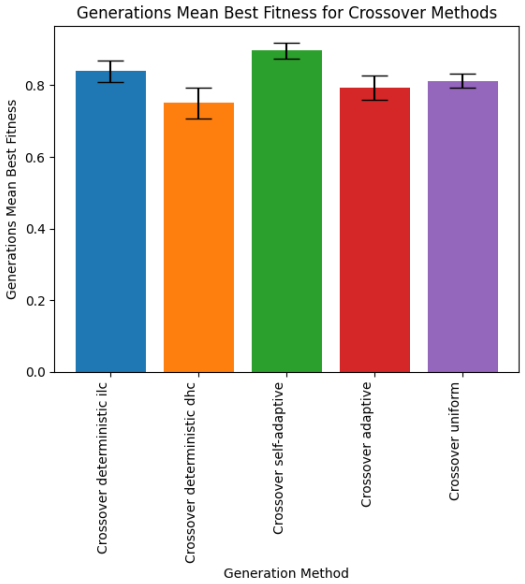
- **Mean number of generations**: the mean number of generations of each method can be observed in Figure 5.5c and Table 5.3c. The Figure 5.5c presents that the **adaptive crossover** was the method that **took the least amount of generations** in all ten execution iterations, needing about 34^1 generations;

- **Average execution time**: the average execution time for each method is represented in Figure 5.5d and Table 5.3d. The Figure 5.5d shows that the **deterministic ILC crossover** was the method that **took the least amount of time to execute**, averaging an execution time of 715^1 seconds.

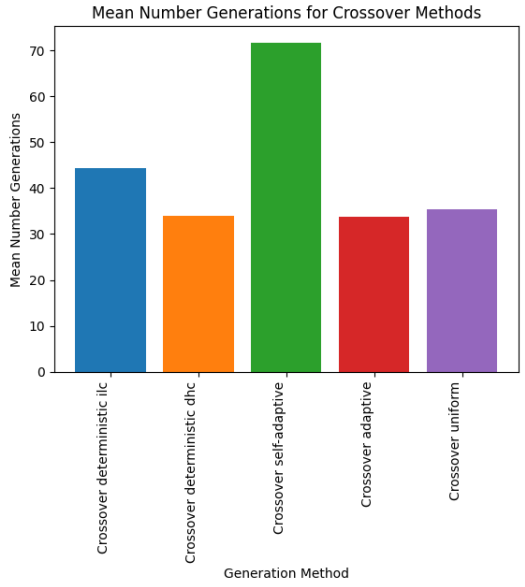
The results presented in Figure 5.5 and Table 5.3 generally demonstrate that the **self-adaptive method was the best algorithm** of all the tested crossover methods in terms of population and search quality.
The **self-adaptive** method obtained the **best MBF value** in respect to the other crossover methods. This high MBF can also be attributed to the low variance of high fitness values in each generation by this method. The observation of the fitness scores obtained in Tables 5.3a and 5.3b demonstrates that these values, in addition to being high, are not that separated from each other as the crossover methods deterministic DHC and adaptive are. These results proximity reveals itself to be an excellent sign that the search is not getting stuck at local optima. The reason for reaching this conclusion stems from the variation of the best fitness values per generations for the self-adaptive method, as this method obtained a high variation of high fitness values close to the global optima which translates into fewer situations where the GA was stuck in local optima. As for the other methods, many of them persisted in lower fitness values during a large number of generations, this means that the associated GAs got stuck at a local optimum [53]. This situation is represented in Figure 5.6, which demonstrates an iteration example associated to the crossover methods MBF and the best fitness values variations during the generations.



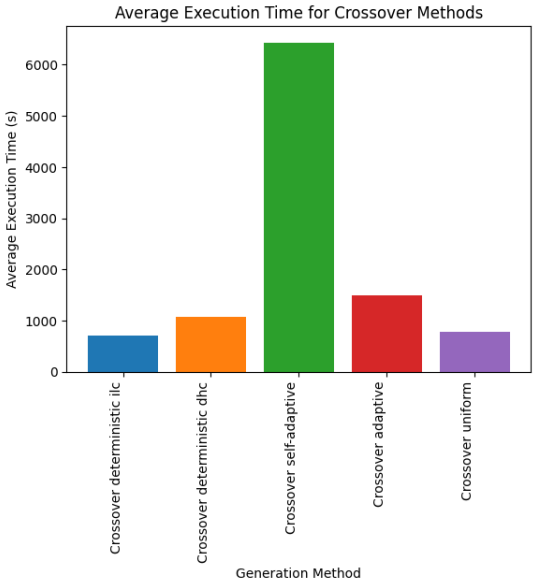
(a) Mean best fitness value for each crossover method with respective standard deviation.



(b) Mean of the best fitness values per generations for each crossover method with respective standard deviation.



(c) Mean number of generations taken by each crossover method.



(d) Average execution time (in seconds) for each crossover method.

Figure 5.5: Histograms of generation statistics for each crossover method execution.

	Deterministic ILC	Deterministic DHC	Self-Adaptive	Adaptive	Uniform
MBF	0.84480	0.85812	**0.93163**	0.86054	0.85430
Stdev	0.03305	0.01391	0.01463	**0.01173**	0.01229

(a) Mean best fitness scores for each crossover method.

	Deterministic ILC	Deterministic DHC	Self-Adaptive	Adaptive	Uniform
Generations Mean Best Fitness	0.83927	0.75049	**0.89710**	0.79259	0.81208
Stdev	0.03023	0.04360	0.02226	0.03453	**0.01941**

(b) Mean of the best fitness scores per generations for each crossover method.

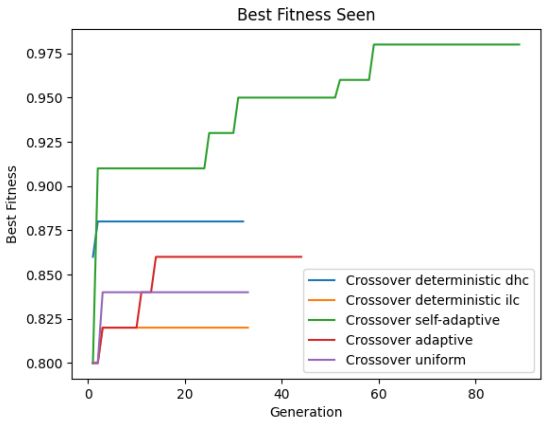
	Deterministic ILC	Deterministic DHC	Self-Adaptive	Adaptive	Uniform
Mean Number of Generations	44.3	33.9	71.7	**33.8**	35.4

(c) Mean number of generations taken by each crossover method.

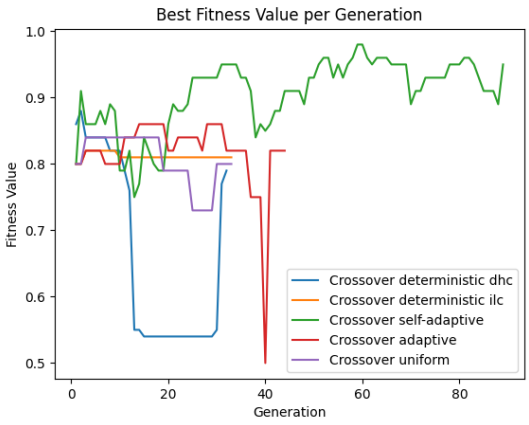
	Deterministic ILC	Deterministic DHC	Self-Adaptive	Adaptive	Uniform
Average Execution Time	**715.34581s**	1068.76897s	6432.50857s	1487.68248s	775.63365s

(d) Average execution time (in seconds) for each crossover method.

Table 5.3: Generation statistics scores for each crossover method.



(a) Best fitness value variation for each crossover method.



(b) Best fitness value per generation for each crossover method.

Figure 5.6: Individuals fitness variation comparison for each crossover method.

The analysis of Figure 5.6a suggests that the self-adaptive crossover, throughout the generations, was always able to obtain better individuals until it reached a plateau at the 60^{th} generation. The search performed by the other methods stagnated very early. The only method that was able to perform a wider search was the adaptive crossover method, however, it wasn't anything significant. The performance obtained by the other methods, except the self-adaptive crossover method, demonstrated signs of early convergence into local optima. This is even more obvious when observing Figure 5.6b that demonstrates the best fitness values, obtained by each method, in each generation. All crossover methods, excluding the self-adaptive method, exhibited early convergence behavior where initially they were able to obtain better individuals in each generation but rapidly declined after a few generations and ended up to enter spaces where a local optima was found. A direct example of a presence of a local optima can be visualized for the deterministic DHC crossover method, where the search stagnated between generations 12 and 29. The main objective during the generations is to achieve an individual that has the highest branch coverage possible, so the global optima value is 100% branch coverage. The self-adaptive crossover method was the best algorithm when it comes to explore the space as this can be viewed for the recurrent best fitness variation in each generation demonstrated in Figure 5.6b. This method achieved a high fitness solution around the 30^{th} generation, however it was able to achieve better solutions successively as the best solution was found at the 60^{th} generation. This algorithm rarely got stuck in local optima, as it can be seen for the variation shown in Figure 5.6b.
The behavior demonstrated by the self-adaptive crossover method was limited by one factor. This factor corresponds to the constitution of the CUT more concretely on the branch conditions present in two CUT setters. The setters for the `gender` and `amount_exercise` variables were limiting the branch evaluation on the methods `mifflin_stjeor_equation` and `tdee_calculation`. This is due to an `If` condition that stopped the phenotype program execution with an exception. The condition was present in both setters and during the generation of the individuals, one condition on both methods was not fully covered, i.e, it was only partially covered as the evaluated condition was only evaluated as `True`. In order to make this explanation more elucidative, an example of the affected condition is shown in Listing 5.1 for the `gender` variable.

```
1 def mifflin_stjeor_equation(self):
2     if self.gender.upper() == 'F':
3         return (10 * self.weight) + (6.25 * self.height) - (5 * self.age) - 161
4     elif self.gender.upper() == 'M':
5         return (10 * self.weight) + (6.25 * self.height) - (5 * self.age) + 5
6     else:
7         raise ValueError('Gender is not specified correctly')
```

Listing 5.1: Partial branch coverage example.

The `Elif` condition presented in line four of the listing 5.1 was never evaluated as `False`. This is due to the `If` condition present in the setter for the `gender` variable. The setter can be visualized in Listing 5.2.

```
1 def gender(self, gender: str):
2     if gender.upper() not in ['M','F']:
3         raise ValueError('Gender is not specified correctly')
4     self.__gender = gender
```

Listing 5.2: Setter for the `gender` variable.

The `If` condition throws a `ValueError` exception whenever the individual gender is not masculine or feminine. In situations where the gender is defined as "non-defined", the `If` condition is evaluated as `True` throwing a `ValueError` exception and consecutively stopping the phenotype program associated to the individual. Stopping the execution of the program in the setter would mean that the condition on line four of Listing 5.1 would never be evaluated as `False`. It does not have an opportunity to be evaluated as `False` because the setter is implicitly called in the constructor, which is the first instruction for each test case. In the CUT, each attribute is defined with the `property` and `*.setter` decorators.

This was detected by the stagnation of the 98% fitness score presented in Figure 5.6a and after the analysis of this result, this error was corrected in both instances of the CUT setters that presented this inconsistency by removing the `If` condition. Acknowledging this error calls **attention to the fact that the self-adaptive crossover method was probably able to reach the global optima**, which in turn is another point to prove **its superiority to its counterparts**.

Based upon the above observations, the **self-adaptive crossover method**, proposed in the work "An Emperical Study on GAs "Without Parameters"" [47], **cannot be inserted into the final algorithm configuration as the mutation optimization cannot be applied in conjunction with this optimization method**. The authors behind this work mention that in specific situations during the execution of the self-adaptive crossover method, certain individuals need to undergo mutations according to their mating potential. According to my interpretation, the stated mutation operation is always applied in those situations and I considered this fact in the implementation. The authors also proposed a GA that implemented the conjunction of both self-adaptive crossover and mutation methods however as the self-adaptive mutation method returns a rate it means that the operation is not guaranteed to happen and because of my interpretation for their explanation of the self-adaptive crossover method, I have decided to implement the self-adaptive crossover method with guaranteed mutation. The mutation methods used during the execution of this crossover optimization method were standard mutation methods (addition, removal and replacement of genes). The **adaptive crossover method will be the one applied** instead, as it was the second-best method to achieve a high MBF score as well as its capability of not being stuck so early in local optima as the rest of the other methods as demonstrated in Figure 5.6a. Additionally, this method does not seem to be stuck as much as the other methods during the search from the analysis of Figure 5.6b. Despite the incompatibility of the self-adaptive crossover, this method, similar to what happened in the results evaluation of the adaptive selection method in section 5.2.2, will be executed alongside the optimized GA, adaptive selection GA and a standard GA in the final benchmark.

5.2.4 Mutation Optimization

This section presents the analysis of the results obtained for the GAs centered around the mutation genetic operator. The executed GAs have been set up with different mutation methods, including standard and optimized mutation methods. The applied methodologies are presented as follows:

- **Standard methods**: several standard mutation methods consisting in addition, removal and replacement of genetic information were applied during the GA executions. These methods, according to the unit test theme of this work, include addition and removal of test cases and replacement of instructions inside the test cases;

- **Deterministic mutation rate adjustment**: a deterministic mutation rate adjustment mechanism was proposed in the work "Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach" [51]. The proposed mechanism dynamically adjusts the mutation rate during the GA generations using a deterministic rule that takes into account generation levels [51]. The general explanation was already given in section 3.2 and the core of the mechanism was already explained in section 5.2.3 under the "Deterministic crossover rate adjustment" bullet point. In this bullet point, the changes to the formula will be presented. The mechanism is both the same for the crossover and mutation operators, only differing on the nomenclature and on which genetic operator is chosen for the formula.

  The work implemented two different instances of mutation rate adjustment. These instances are deterministic rules that change the rate according to the current generation number of the GA and the maximum number of generations defined for it. The mutation rate can be adjusted either incrementally or decrementally throughout the generations. The two deterministic mutation rate adjustment techniques are the Dynamic Increasing of Low Mutation Ratio (ILM) and Dynamic Decreasing of High Mutation Ratio (DHM). The ILM technique is applied with the following equation 5.7 [51]

$$MR = \frac{LG}{Gn} \tag{5.7}$$

  where MR is the adjusted mutation rate, LG is the number of generation level (current generation number) and Gn is the maximum number of generations. This technique aims to gradually increase the mutation rate from 0% to 100% during the execution of the GA. The DHM technique is represented by the following equation 5.8 [51]

$$MR = 1 - (\frac{LG}{Gn}) \tag{5.8}$$

  where MR is the adjusted mutation rate, LG is the number of generation level (current generation number) and Gn is the maximum number of generations. The formula

variables are exactly the same as the ILM technique, however the purpose behind it changes, as the DHM technique aims to gradually decrease the mutation rate from 100% to 0% during the execution of the GA.

The research about alternative deterministic mutation methods ended up in the same situation as the deterministic crossover methods, already explained in section 5.2.3, as no other relevant works were found for this topic;

- **Adaptive mutation rate adjustment**: an adaptive method, consisting in the mutation rate adjustment throughout the run of the GA, was proposed by Shi, Eberhart and Chen [52]. This work implemented a fuzzy system that adapts the mutation rate of GA according to feedback obtained from the generations. A general description about this work was already presented in section 3.2 and the mechanism behind the fuzzy system was thoroughly explained in section 5.2.3 under the "Adaptive crossover rate adjustment" bullet point. The only additional information that can be given is the range of values of the adjusted mutation rate obtained from the output of the fuzzy system which is $[0.005, 0.1]$;

- **Self-adaptive mutation rate adjustment**: a self-adaptive mutation rate adjustment was proposed in the work "Intelligent Mutation Rate Control in Canonical Genetic Algorithms" [54] consisting in using a mechanism that adjusts the individual's encoded mutation rate throughout the GA run. The scope of the mechanism was already explained in section 3.2 however the formulation of the mechanism himself needs to be presented. The equation used to adjust the encoded mutation rate of the individuals is represented in equation 5.9 [54]

$$p' = (1 + \frac{1-p}{p} \cdot \exp(-\gamma \cdot N(0,1)))^{-1} \tag{5.9}$$

where p' is the adjusted encoded mutation rate, p is the encoded mutation rate, γ is a constant, $N(0,1)$ is a random value from a normal distribution with mean 0 and standard deviation 1. The learning rate γ specified in the equation 5.9 is used to control the convergence speed during the GA run [54]. The authors used the value of $\gamma = 0.22$ and this was exactly the same value used during the executions of this self-adaptive mutation method.

The Figure 5.7 and Table 5.4 demonstrate the results from the multiple mutation methods executions. The analysis of these results are composed of different evaluation metrics:

- **Mean best fitness**: the MBF values were calculated for each mutation method. The results presented in Figure 5.7a and Table 5.4a indicate that the **add test case mutation method** was the **best out of all algorithms**, as it obtained the **highest MBF** score of 0.95131 out of all the other mutation methods. The data standard deviation of the **add test case mutation method** is also the lowest, with a value of 0.00562 being the **most robust** mutation method;

- **Mean of the best fitness values per generations**: the mean of the best fitness values per generations was calculated for each mutation method. Figure 5.7b and Table 5.4b suggest that the **add test case mutation method** was the **best method** to achieve **high fitness values** during the generations, obtaining a fitness score of 0.94961. Additionally, it is the **most robust method** as its data standard deviation scored a value of 0.00608;
- **Mean number of generations**: the mean number of generations that each mutation method took was calculated and displayed in Figure 5.7c and Table 5.4c. The **delete test case** was the method that **took the least amount of generations**, needing about 34[1] generations to find a solution;
- **Average execution time**: the average time of execution was calculated for each method. The results are presented in Figure 5.7d and Table 5.4d and these suggest that the **delete test case mutation method** took the **least amount of time to execute**, averaging an execution time of 574[1] seconds.

	Add Test Case	Delete Test Case	Deterministic ILM	Deterministic DHM	Adaptive	Self-Adaptive	Change Parameters
MBF	**0.95131**	0.85595	0.87281	0.92833	0.88509	0.90207	0.85855
Stdev	**0.00562**	0.01859	0.01785	0.01266	0.02116	0.02525	0.01560

(a) Mean best fitness scores for each mutation method.

	Add Test Case	Delete Test Case	Deterministic ILM	Deterministic DHM	Adaptive	Self-Adaptive	Change Parameters
Generations Mean Best Fitness	**0.94961**	0.81175	0.83781	0.88327	0.85393	0.86968	0.79673
Stdev	**0.00608**	0.02375	0.01994	0.02205	0.02817	0.02613	0.02908

(b) Mean of the best fitness scores per generations for each mutation method.

	Add Test Case	Delete Test Case	Deterministic ILM	Deterministic DHM	Adaptive	Self-Adaptive	Change Parameters
Mean Number of Generations	64.6	**33.7**	46.6	54.3	56.7	58.6	35.4

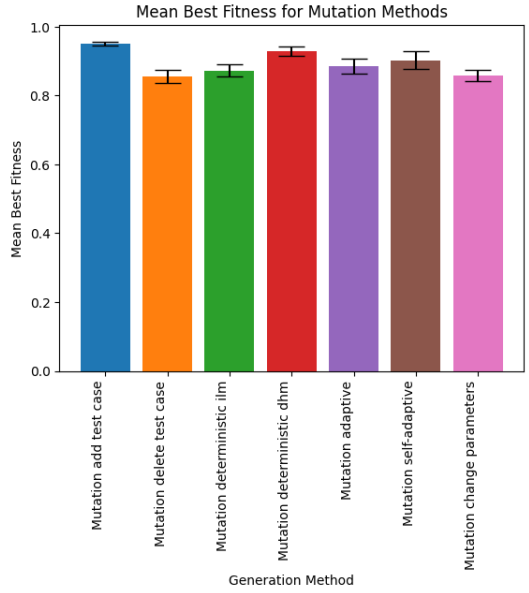
(c) Mean number of generations taken by each mutation method.

	Add Test Case	Delete Test Case	Deterministic ILM	Deterministic DHM	Adaptive	Self-Adaptive	Change Parameters
Average Execution Time	1918.88236s	**573.74788s**	1245.48242s	2524.48207s	2943.48482s	1952.71141s	729.79732s

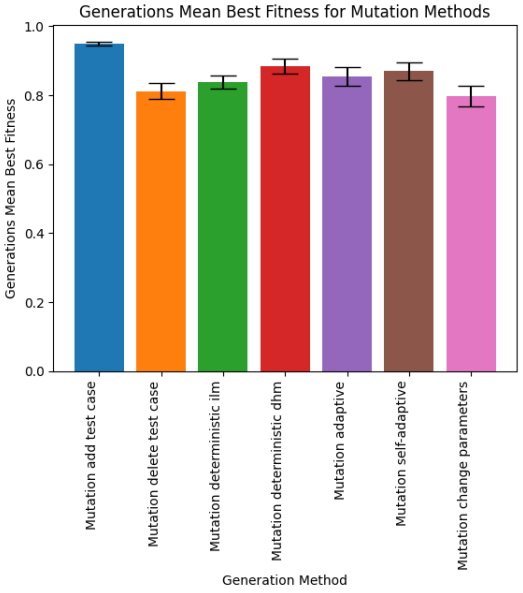
(d) Average execution time (in seconds) for each mutation method.

Table 5.4: Generation statistics scores for each mutation method.

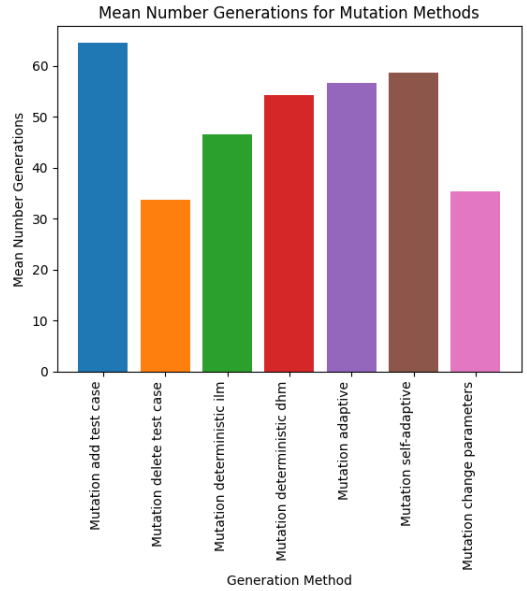
The results presented in Figure 5.7 and Table 5.4 suggest that the **add test case mutation method** was the **best configuration for the mutation operator**, excelling in obtaining high MBF scores and obtaining high fitness score values during the generations, as well as having high robustness. Despite these excellent results, one must take into account that **this method might not be appropriate**. The reasons for this concern are listed below:



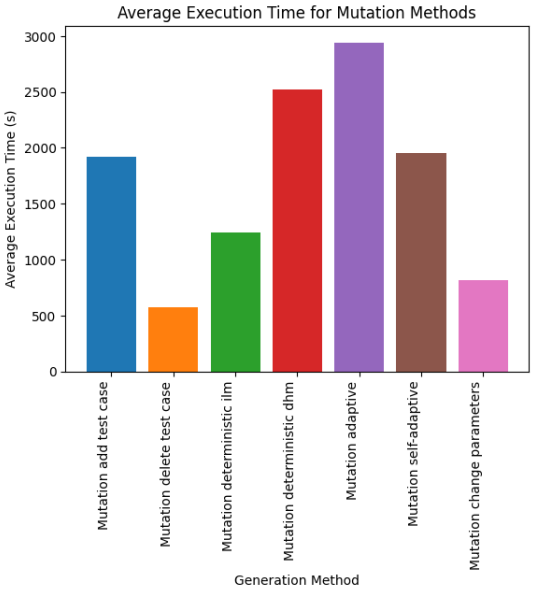
(a) Mean best fitness value for each mutation method with respective standard deviation.



(b) Mean of the best fitness values per generations for each mutation method with respective standard deviation.



(c) Mean number of generations taken by each mutation method.

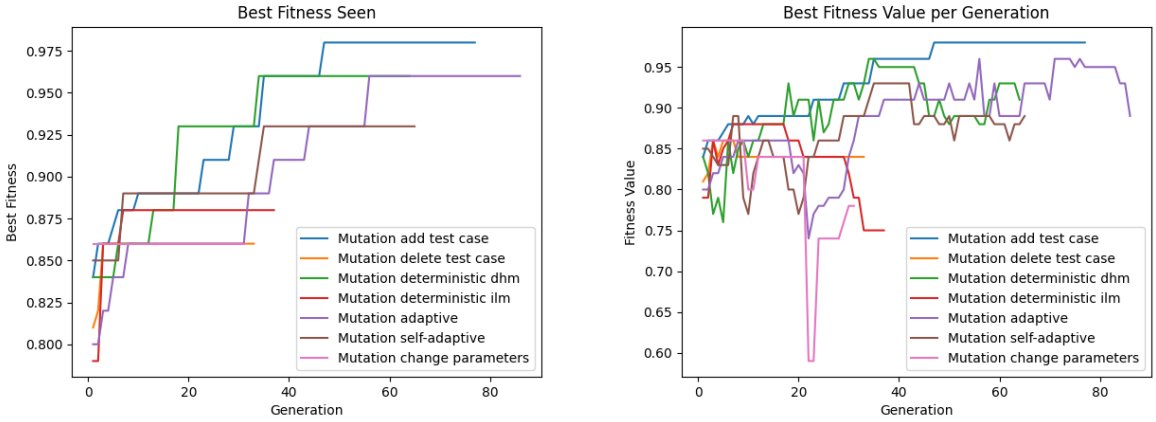


(d) Average execution time (in seconds) for each mutation method.

Figure 5.7: Histograms of generation statistics for each mutation method execution.

- **Fixed mutation rate**: the used mutation method had a fixed mutation rate throughout the run, as the method used was a standard one. This can be detrimental to the overall search process, as it can hinder the exploitation phase of a GA. In the final phases of a GA run, an exploitation activity is advised as there is a high chance of high fitness individuals dominating the population pool and a **search around these high fitness solutions vicinity should be sought after** [5]. A fixed mutation rate does not consider this selection pressure rationale, being even worse at lower mutation rate values;

- **No generation feedback**: not taking feedback during the GA hinders the search process efficiency and effectiveness. The GA falls under the evolutionary computing paradigm and consecutively it's classified as a stochastic search method [5]. This stochastic nature can be detrimental to the overall search process as there are no mechanisms to guide the search as it's all based upon randomness. The search can be led into premature convergence without a proper guidance during the exploration and exploitation phases of a GA [5]. Feedback such as population fitness variation, generation level and best fitness variation are perfect examples to guide the search towards optimal solutions. Simple mechanisms such as adapting mutation rate according to this feedback can properly adapt the mutation rate to optimize the search during the GA generations. This adaptation also falls under the selection pressure rationale [5], as properly adjusting the mutation rate in the final phases of the search can be essential to obtain high fitness solutions.



(a) Best fitness value variation for each mutation method.

(b) Best fitness value per generation for each mutation method.

Figure 5.8: Individuals fitness variation comparison for each mutation method.

An iteration example was taken from the results in order to further enhance the above conclusion. The execution iteration is represented in Figure 5.8. Figures 5.8a and 5.8b demonstrate the MBF score variation and best fitness value variation per generation respectively. The analysis of the performance of the **add test case mutation method** suggests that it was the best performing method out of all the other mutation methods, however one must analyze

its behavior during the best fitness values variation per generation. In comparison to parameter control methods, such as the adaptive and self-adaptive mutation methods, the add test case mutation method falls more under local optima than the previously mentioned methods. The analysis of Figure 5.8b indicates that throughout the run, the add test case method had various instances of local optima during generations $11 - 21$, $22 - 27$, $28 - 33$, $34 - 45$ while also having low variation of the best fitness value during the generations. As the add test case mutation method falls under suboptimal areas, the search becomes stagnated during a few generations because the method is not capable of exploring the space well enough as the parameter control methods do. This can be seen by the lack of best fitness value variation when the algorithm finds a better solution each time, demonstrated in Figure 5.8b. The adaptive and self-adaptive methods show a higher variation of fitness per generation, as the search does not stagnate as much because the search space is being properly explored. Figure 5.8b demonstrates that there are more peaks and descents in the fitness values for the parameter control methods compared to the add test case mutation method.

Based on the previous observations, the **add test case mutation method** will not be chosen for the final algorithm configuration. The **self-adaptive mutation method will be the one chosen for the final algorithm configuration** because of its high MBF scores and high mean of the best fitness scores per generations, while also being generally robust. This decision needs additional explanation, as the deterministic DHM mutation method was the second-best method according to its fitness scores. Tables 5.4a and 5.4b show that the deterministic DHM mutation method has higher fitness scores overall when compared to the self-adaptive method, however this is due to an **initial good population**. According to the selection pressure concept, that states that when good solutions are found, **more solutions must be sought after around their vicinity** in order to find better solutions. This phenomena should happen in later stages of an GA execution, as the population is essentially composed of high fitness individuals. This situation is similar to what happened for this method as it can be seen in Figure 5.8b that the initial best fitness value was 0.84 which is already a high fitness score and as the deterministic DHM gradually decreases the mutation rate across the generations, it's normal to expect such high fitness scores initially. In cases where the initial population does not have such high fitness values, this method would not have this much impact, being rather detrimental even as high mutation rate for low fitness population would not add any relevant genetic information. The adaptive mutation method was not considered due to its slightly worse performance than the self-adaptive mutation method.

According to the generation results from the genetic attributes in this section, four GAs will be executed for the final benchmark of this work. These GAs include a traditional GA, denominated as TGA, an adaptive selection focused GA, denominated as Adaptive Selection Genetic Algorithm (ASGA), a self-adaptive crossover focused GA, denominated as Self Adaptation of Crossover Genetic Algorithm (SACGA) and an optimized GA, denominated as Optimized Genetic Algorithm (OGA). The TGA will include a standard configuration. The **adaptive selection GA will include the population control optimization as well as the adaptive selection method**, because of its incompatibility with the rest of the ge-

netic operators (crossover and mutation). The **self-adaptive crossover GA will include the population size optimization method as well as the self-adaptive crossover method** since its incompatibility with the adaptive selection method and the mutation operator. The **optimized GA** will include all the best compatible optimizations for each genetic operator which are the population size optimization, the rank selection method, the adaptive crossover method and the self-adaptive mutation method.

5.3 Optimized Genetic Algorithms

This section introduces the evaluation of the resulting GAs from the first benchmark analysis. These GAs were configured according to the best performing GA in each genetic attribute evaluation, as previously demonstrated in section 5.2. The resulting GAs will now be evaluated according to the same performance measures, however they will also be evaluated regarding their ability to reach a desirable fitness score. This last evaluation metric is referring to the SR metric which determines the percentage of algorithms runs where the GA was able to reach a desired fitness score.

This second benchmark is performed with a higher limit for the maximum number of generations relatively to the first benchmark. The data retrieved is the same as what was retrieved in the first benchmark. In this comparison, four GAs will be executed and compared between each other with a set of performance measures. These GAs include the TGA, ASGA, SACGA and OGA. After the performance evaluation of the configured GAs, the best generated test suite (set of unit tests) of the best performing GA is evaluated according to unit test structure and quality. This part of the work executed, in total, 40 genetic algorithms where ten runs were executed for each of the four configured GAs.

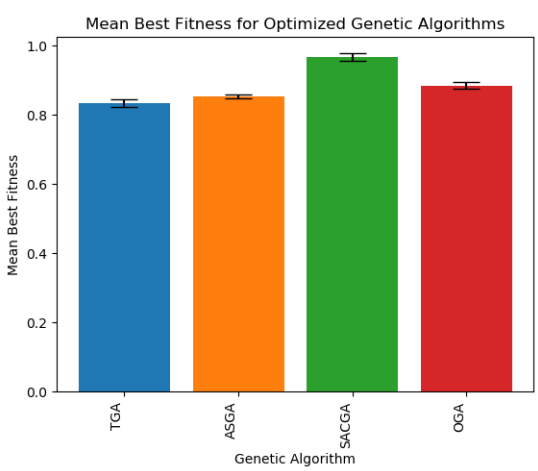
There are some key differences regarding some parameter values in the configuration file in this second benchmark. The analysis of the results from the first benchmark also contributed to perform an assessment on the potential impact of the GAs initial configuration. The changed parameters include: the **maximum number of functions** each test case could have (represented by `max_number_functions`), the **maximum number of test cases** the test suite could have (represented by `max_number_test_cases`), the **maximum number of generations** for the GA run (represented by `max_number_generations`), the **maximum number of generations without the best fitness improvement** (represented by `fitness_max_stagnation_period`), **initial population size** (represented by `population_size`), **maximum lifetime for the individuals** (represented by `lt_max`), **minimum lifetime for the individuals** (represented by `lt_min`) and **population size growth variable** (represented by `alpha`). The changed values can be observed in section 4.6.1 under the “Configurations for the GA“ and “Configurations for GAs optimizations” bullet points.

Important to mention that, in order to support the results of the benchmarks, only a few iteration examples from the executions are shown. This is similar to what happened in the first benchmark, as each GA finished executing with different number of generations and as such, it’s not possible to perform a mean of all ten runs between each GA for a desired

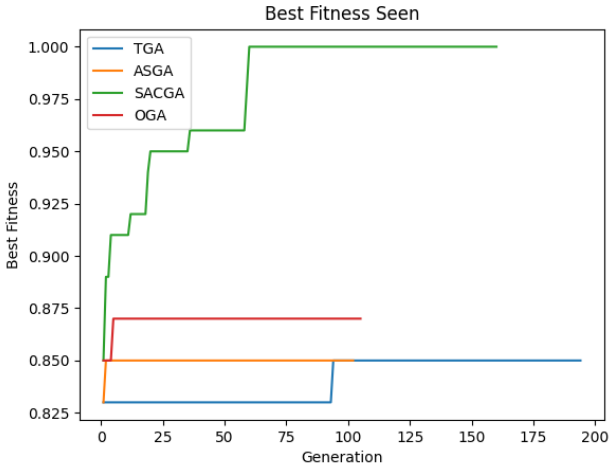
performance metric.

5.3.1 Mean Best Fitness

This section presents the MBF performance measure which evaluates the mean best fitness out of a set of genetic algorithms executions. For each configured GA, the corresponding MBF results are demonstrated in Figure 5.9 and in Table 5.5.



(a) Mean best fitness value for each genetic algorithm with respective standard deviation.



(b) Best fitness value variation for each genetic algorithm.

Figure 5.9: Mean best fitness results from the optimized genetic algorithms runs.

	TGA	ASGA	SACGA	OGA
MBF	0.83348	0.85278	**0.96540**	0.88395
Stdev	0.01074	**0.00652**	0.01094	0.00966

Table 5.5: Mean best fitness scores for each genetic algorithm.

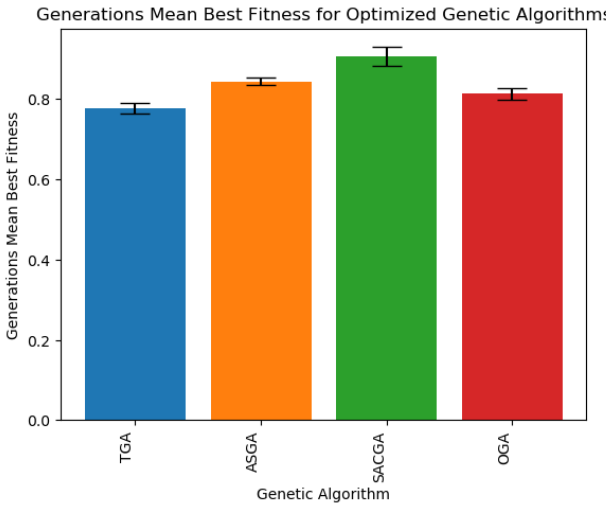
The analysis of Figure 5.9a and the fitness scores in Table 5.5 demonstrate that the **SACGA** was the **best GA in obtaining the best individuals**, scoring a mean of 0.96540 for the MBF performance measure. In accordance to the MBF standard deviation, the SACGA is the worst performing GA of all, being the **ASGA the most robust** with a standard deviation value of 0.00652.

The SACGA outperformed every GA in the second benchmark when it comes to the MBF performance measure. It was even able to reach the global optima, as demonstrated in Figure 5.9b which represents an iteration example out of all ten iterations of executions. It can also be observed that the SACGA algorithm was able to get out of multiple local optimal areas as opposed to the rest of the GAs which despite some best fitness improvements were not able to get out of local optima. The fact that the SACGA was able to get out of these local optima areas was probably due to the conjunction of the crossover and mutation operations according to the individual's mating potential. Additionally, there is a chance that a high fitness individual was kept to be selected during the next generation selection process according to the mechanism explained in section 5.2.3 under the "Self-adaptive crossover"

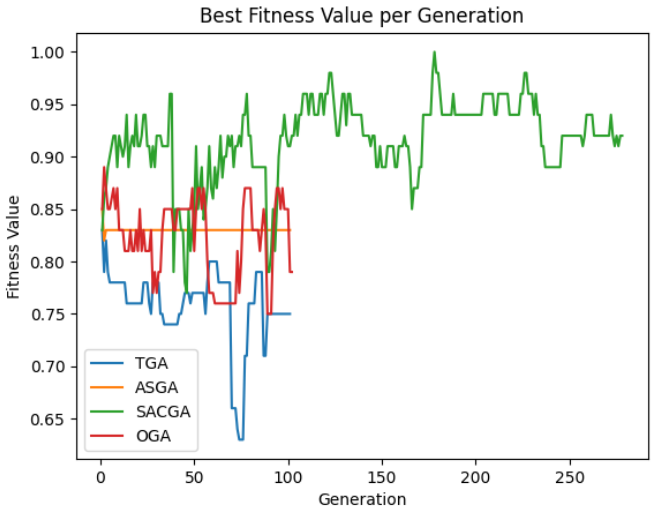
bullet point. This self-adaptive crossover mechanism, associated with the population control optimization that gradually grows and shrinks the population based upon the RLT mechanic associated to the individual's fitness scores, is what probably contributed to conglomerate high fitness individuals. The fact that the SACGA was not stuck in local optima for long when the population control optimization is applied is another important fact to consider. The crossover order decision with respective application of mutation is what led this GA to not get stuck under local optima even in situations where the population was probably composed of high fitness individuals because of the population control optimization.

5.3.2 Mean of the Best Fitness Values per Generations

This section presents the mean of the best fitness scores per generations performance measure, which evaluates the best fitness variations during each generation for each configured GA. The results are shown in Figure 5.10 and in Table 5.6.



(a) Mean of the best fitness values per generations for each genetic algorithm.



(b) Best fitness value variation per generation for each genetic algorithm.

Figure 5.10: Mean of the best fitness values per generations results from the optimized genetic algorithm runs.

	TGA	ASGA	SACGA	OGA
Generations Mean Best Fitness	0.77688	0.84369	**0.90520**	0.81231
Stdev	0.01248	**0.00874**	0.02316	0.01452

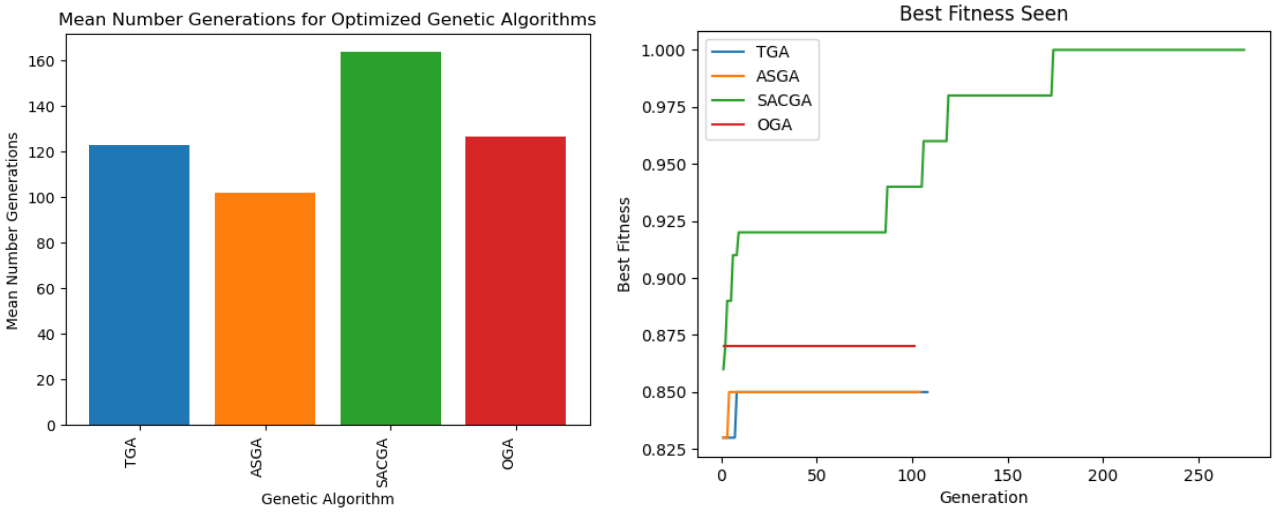
Table 5.6: Mean of the best fitness scores per generations for each genetic algorithm.

The analysis of Figure 5.10a and the fitness scores in Table 5.6 suggest that the **SACGA** was the **best algorithm when it comes to explore the search space** during the generations. It obtained the **best fitness values across each generation** of the ten GA runs, scoring a mean of 0.90520. The **standard deviation** of this GA was the worst out of all the other GAs where the **ASGA was the most robust** with a standard deviation value of 0.00874.

The SACGA was the best algorithm when it comes to genetic variety during the generations. This can be deduced by the behavior presented by the SACGA in Figure 5.10b where it presented the most fitness variance per generation. This demonstrates that the search space was evenly explored where the best fitness score was constantly changing throughout the generations, which means that the SACGA was not stagnating, in terms of the best fitness improvement, as the other GAs were. The GA that demonstrated the highest stagnation regarding the best fitness improvement was the ASGA algorithm. The TGA presented more signs of stagnation when compared to the OGA that showed more variance overall. Even tough the OGA was the second best GA, it did not reach the levels of fitness variance presented by the SACGA. Additionally, this high variance of fitness is what led the SACGA to execute longer as the generation number limit, imposed by the `fitness_max_stagnation_period` field in the configuration file, was reset because of the best fitness improvement that happened in the 36^{th}, 121^{st} and 177^{th} generations.

5.3.3 Mean Number Generations

This section demonstrates the mean number of generations that all the GAs took across all ten runs. This is a much-needed performance measure as it can evaluate how quick the algorithm achieved a solution [5] not in terms of time per se but in terms of genetic convergence speed. The results for this performance measure are represented in Figure 5.11 and in Table 5.6.



(a) Mean number of generations taken by each genetic algorithm.

(b) Best fitness improvement across the generations for each genetic algorithm.

Figure 5.11: Mean number of generations results from the optimized genetic algorithms runs.

	TGA	ASGA	SACGA	OGA
Mean Number of Generations	122.7	**102.2**	163.6	126.6

Table 5.7: Mean number of generations taken by each genetic algorithm.

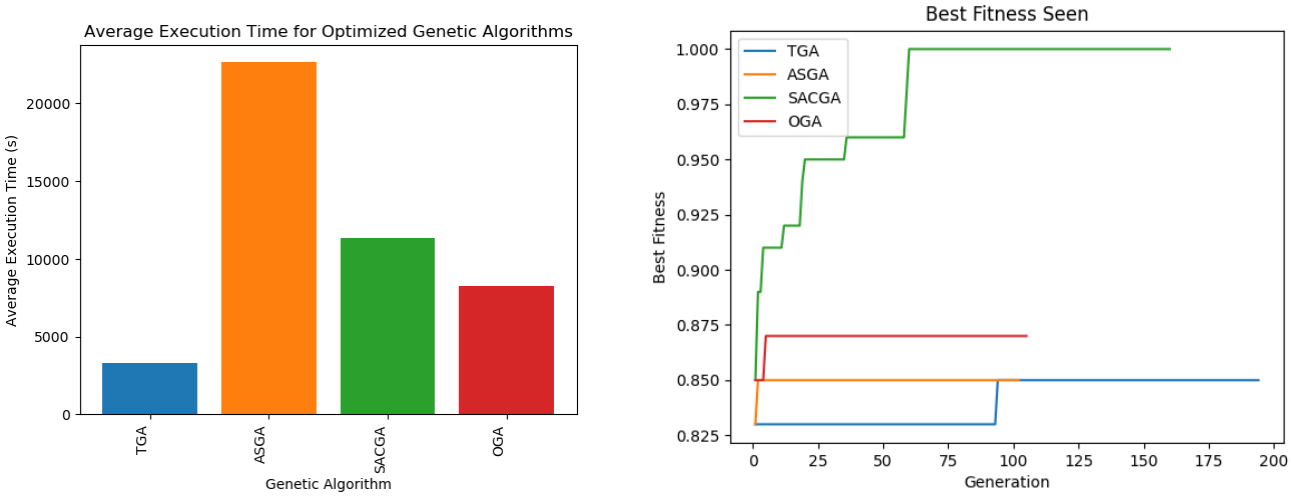
The analysis of the Figure 5.11a and the values in the Table 5.7 indicate that the **ASGA**

took the least amount of generations to achieve a solution which took about 102[1] generations. However, a low amount of generations does not necessarily mean a good outcome. This is verified by the best fitness improvement of the GAs represented in Figure 5.11b where the ASGA quickly converged into a local optima without any type of improvement and ended the search early. In fact, regarding the other GAs, except for the SACGA, they presented a similar behavior. This can be due to several reasons, however one of them could be the lack of more generations for the search. Even the SACGA, which was the best performing algorithm, stagnated in the best fitness improvement between the 9^{th} and 86^{th}. However, due to the time (number of generations) the SACGA still had left, it was able to find a better solution and thus continue the search. This generation limit reset upon finding a better individual gives the algorithm the necessary time to explore the space which can contribute to find even better solutions. This phenomena can be verified in the successive best fitness improvement after the 86^{th} generation until the GA ended the run in the 273^{rd} generation.

Giving the GA enough time to explore is a great factor to consider. Even though the ideal situation is to find the best solution in the least amount of generations possible, there are some scenarios, like the one presented in Figure 5.11, where, if given enough time, the GA can find optimal solutions.

5.3.4 Execution Time

This section presents a similar objective to the performance measure demonstrated in section 5.3.3 which consists in evaluating how much time the GAs took to explore the search space. This performance measure is the measurement of the GAs execution time. This time measurement of the GAs is made on the CPU time previously explained in section 2.3.3.5. The GAs average execution time are represented in Figure 5.12 and Table 5.8.



(a) Average execution time (in seconds) for each genetic algorithm.

(b) Best fitness seen across the generations for each genetic algorithm.

Figure 5.12: Average execution time results from the optimized genetic algorithms.

	TGA	ASGA	SACGA	OGA
Average Execution Time	**3329.28479s**	22582.18567s	11298.25754s	8216.38832s

Table 5.8: Average execution time (in seconds) for each genetic algorithm.

According to the results presented in Figure 5.12a and Table 5.8, the **TGA took the least amount of time to execute**, however this can be deceiving. Less amount of time to execute does not equal to less number of generations taken. This can be observed by the difference between the TGA and ASGA execution times in Table 5.8 and respective number of generations taken in Table 5.7. The TGA took the least amount of time but took a higher number of generations during the run when compared to ASGA. Additionally, taking less time to execute does not mean good performance at all times, similarly to the explanation given to the mean number of generations in section 5.3.3. The analysis of Figure 5.12, demonstrating an iteration example, shows the best fitness stagnation for TGA where the algorithm fell under local optima areas despite generally being the fastest to execute in CPU time. The SACGA shows impressive results, being able to reach global optima as presented in Figure 5.12b as well as better fitness scores across all ten runs represented in Tables 5.5 and 5.6.

One major point of discussion relatively to the Figure 5.12a falls upon the high execution time of ASGA. This phenomena was already explained in section 5.2.2 where the adaptive selection method, present in ASGA, grows the population whenever the population size is odd because of the size of the mating lists. This growth in population size can be observed in Figure 5.13, where ASGA shows the highest population size across the generations being this the reason for the high execution time. The `Coverage` module that calculates each individual's coverage in conjunction with the crossover operation will take more time to complete if the population size is high.

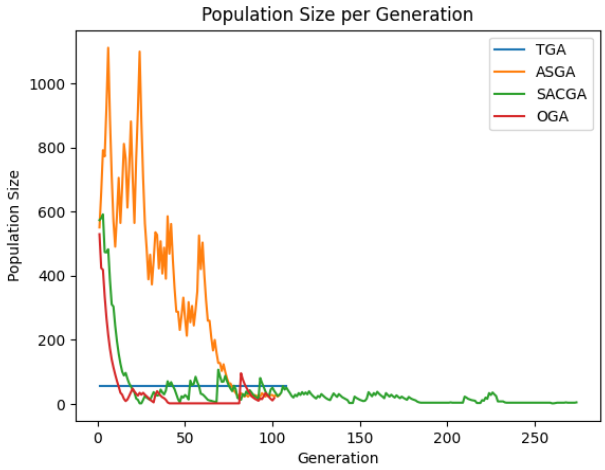


Figure 5.13: Population size variation for each genetic algorithm.

5.3.5 Success Rate

This section presents an important performance measure that evaluates an GA effectiveness in achieving a desired solution. This measure consists in the percentage of GA runs where an objective was met. This objective corresponds to a fitness score obtained from the mean of the results of the first benchmark for the MBF metric represented in Tables 5.1a, 5.2a, 5.3a and 5.4a. The mean of these fitness scores is 0.87 (rounded to the nearest hundredth). The percentage of the runs that obtained an individual with a fitness score equal or higher to 0.87 were calculated and the results are shown in Table 5.9.

	TGA	ASGA	SACGA	OGA
Success Rate	0%	10%	**100%**	**100%**

Table 5.9: Success rate for each genetic algorithm.

The **SACGA** and **OGA** are tied where each algorithm **achieved a desired solution in all ten runs**. The ASGA algorithm only got one successful run, whereas the TGA did not have any successful runs. Even though the great results shown by SACGA and OGA, the best performing GA is SACGA because of its ability to reach global optima. The **same calculation** regarding the SR performance measure **was performed for a** 1.0 **fitness score** representing a global optima. The **SACGA had a success rate of** 30% meaning the GA had three successful runs out of ten runs. The OGA had a success rate of 0% meaning the GA did not have any run where the global optima was reached.
Based upon the observations presented in sections 5.3.1, 5.3.2, 5.3.3, 5.3.4 and 5.3.5 the SACGA was the superior GA obtaining, in average, high fitness values while also being the most effective in obtaining desired solutions. Additionally, it was the only GA that was able to obtain global optima, which already proves its superiority to the other GAs in generating representative unit tests.

5.3.6 Generated Unit Tests

The best generated test suites from the SACGA algorithm runs were extracted. These test suites, which are considered as global optima from the analysis of SR performance measure, were evaluated according to some unit test quality metrics. A subset of one of the best generated test suites is represented in Listing A.2. The subset of the test suite can be evaluated in terms of quality under different topics:

- **Code coverage**: a potential way to evaluate the quality of a unit test relies upon the objective assessment of their design. One of the choices goes in accordance to the code coverage of a target SUT. The best generated test suites from all ten iterations of SACGA obtained 0.96540 mean fitness score, which means the generated test suites covered almost all the possible branch conditions of the CUT. It was already mentioned that this GA obtained three instances of global optima, where all the possible branches of the CUT were traversed by the test suite. This could indicate that the SACGA produced representative and effective test suites to thoroughly test the CUT, however as previously mentioned in section 2.2.7, a higher code coverage value does

not fully represent code quality. This statement does not dismiss the fact that a higher code coverage test suite is indeed better when compared to a lower one, however, this measurement cannot be solely used to determine the quality of a test suite. A simple example of this situation would be a test suite that covers all the possible branches but contains a good amount of test cases that are redundant. This fact is observed in the subset of one of the best generated test suites and explained in the next bullet point;

- **Redundancy**: this problem was expected to happen according to the stochastic nature of the GAs. The generation of the individual's information is still random despite the optimizations applied. The purpose behind these mechanics was to guide the search towards better search spaces, as the optimizations themselves focused on the search guidance and fitness improvement. This does not dismiss the possibility of redundancy to appear in the generated test suites as already mentioned in the previous bullet point, as test suites with a high amount of test cases can contain redundant method calls. This is exactly what happened during the generations.

  Some redundancy method calls can be observed in Listing A.2 in lines 32, 44 and 46. The methods in these lines were previously called in their respective test case without any need to be repeated. These redundant calls do not add any sort of value to the generated unit tests as they increase the length of the tests themselves, make them difficult to read and affect negatively the performance of the GA itself as the same genetic information hinders the search process for better solutions;

- **Maintainability**: another point to consider regarding the generated unit tests is the maintainability aspect. One must evaluate if the tests are easily perceived by the reader and if they are simple to run. The second consideration does not impose too much weight in this discussion, as the target CUT has a simple structure with a few branch conditions. The first consideration is the most important one, as the size of a test directly impacts its readability [1]. The fewer the test size, the easier it is to read it [1].

  Observing the generated test suite example in Listing A.2 it can be said that it's quite high considering that not all test cases are listed. The best generated test suite example shown only presents 9 out of 20 test cases. This makes the test suite a lot harder to read and in addition, there is also redundant method calls which make it even harder to understand;

- **Mutation testing**: a good quality attribute about the unit tests rely on their ability to detect potential faults. This topic delves upon the mutation analysis theme already thoroughly explained in section 2.2.6.6. The three best generated test suites, obtained from the SACGA runs, were evaluated in terms of mutation score. The average mutation score, calculated by the `mutatest` tool, for all three test suites is 41%. These mutation scores were calculated with `mutatest` "full mode", which is a run mode that forces the tool to apply mutations on all possible combinations in the test suite. This mutation score shows room for improvement, however, the principal objective behind this work was not to focus on the generation of unit tests for mutation testing purposes

but rather to generate representative unit tests regarding the chosen SUT. There are some researches that generate mutation-driven unit tests in order to target the maximum amount of mutation score possible [17]. A possible explanation about the low percentage of detected defects for this work is due to the lack of assertion phases. Test oracles are an important factor that provide an identification of the misbehavior present in the mutant program version compared to its normal version [17]. One simple example for this is a specified test oracle that verifies an expected outcome on a mutant that affected a conditional operator. This test oracle verifies the outcome provided by the faulty program with the original program, properly identifying the presence of a fault;

- **No assertion phase**: this fact was already mentioned and thoroughly explained in section 4.8. The lack of assertion phases directly impacts the quality of the unit tests themselves. This phase is responsible for the validation of the unit tests to determine their overall correctness [1] according to the SUT context. The fact that unit tests do not include this phase makes it's very difficult to determine whether a test passes or fails. The behavior of the unit test is not verified, but instead if its execution occurred without any errors. This fact however is not entirely incorrect, as there can be unit tests setups to target possible erroneous values to raise exceptions. This is precisely one of the projections made for this dissertation, where value ranges that target erroneous values for the CUT attributes and methods were established in the metadata, as explained in section 4.3.1.

The best generated test suite example shown in Listing A.2 exemplifies a set of unit tests that were able to achieve 100% branch coverage for the calorie intake CUT. Several points can be made according to the generated tests:

- **Execution of CUT attributes and methods**: all the attributes and methods of the CUT are represented, including getters, setters, and the equations related to the calorie intake calculation. The generated test cases start by initializing the CUT constructor and perform method calls in sequence either by calling attributes directly using the getters and setters or either by performing calorie intake calculations using the predefined equations;

- **Representation of erroneous situations**: all possible exceptions defined in the CUT were also explored. Erroneous parameter values including values out of the specified range and non-defined values are present. Several examples of this can be found in lines 4 and 42 for the `bodyfat` attribute, lines 20 and 49 for the `weight` attribute, line 24 for the `height` attribute and line 37 for the `amount_exercise` attribute. The `age` and `gender` attributes are also covered, but are not shown in Listing A.2;

- **Redundancy method calls**: some redundant method calls can also be observed in `test_case_11()` and `test_case_13()` where the methods `katch_mcardle_equation()` and `determine_calorie_intake()` are repeated after the initial call, meaning they were calls without any sort of value whatsoever.

5.4 Conclusion

This section delved upon the discussion of results obtained in both benchmarks, where each GA involved in the benchmarks was evaluated according to a set of genetic algorithm performance measures. The first benchmark evaluated the GA attributes' performance with optimizations applied to them, whereas the second benchmark evaluated a set of optimized GAs to decide the best performing GA. The results of the first benchmark resulted in the specification of four different GAs according to each genetic attribute evaluation. The results of the second benchmark concluded that the GA that implemented the self-adaptive crossover and the population size optimizations was the best out of all the other three GAs in obtaining the best fitness values. This GA was even able to reach global optima in three runs out of the ten runs performed. Additionally, after the second benchmark results, a subset of one of the best generated test suites from the GA was represented and evaluated according to different unit test quality metrics while also including a brief description about the generated test suite. The results obtained from all these experiments concluded that optimizations make a significant impact on the GAs performance and the generated test suites that come from these optimizations provide a good start for an automated generation process for unit testing despite still having some room for improvement.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Unit testing is one of the most important aspects during a software product lifecycle. This is an activity that can be performed both manually or automatically, however reducing costs and improving development speed, during a software life cycle, are very appealing aspects for developers and the automation of this testing process can make this possible.
Unit test generation techniques such as search-based test generation enable the automated generation of unit tests. One example of a search-based test generation algorithm is a genetic algorithm. This algorithm can aid the generation of unit tests by searching a wide space of solutions and return the best one after a search budget has been reached. This space of solutions is basically a set of unit tests where the genetic algorithm, through genetic operators, will return the best unit test seen for a given objective.
This work explored the possibilities within the search-based test generation realm to propose an automated generation of unit tests. The in-depth investigation in this field, focusing on how the genetic algorithms can aid in automated unit test generation, as well as the consideration of optimizations for the genetic attributes, led this work to obtain interesting conclusions.
The applied benchmarks were able to properly distinguish each GA performance, noting their strengths and weaknesses. The first benchmark was able to detail the best set of configurations for each genetic attribute in order to configure the best set of GAs for the second benchmark. The second benchmark was able to provide valuable insights about the performance of the resulting GAs from the first benchmark, as well as to provide representative test suites that were able to reach global optima. These sets of unit tests also provided interesting results about each GA behavior, being also target of evaluation in terms of unit test structure and quality. This dissertation was able to reach its principal objective, which was the proposal of a viable unit test generation approach. This automated unit test generation process, however, still has room for improvement as problems such as lack of assertion phases, presence of redundant method calls, high test case length and somewhat low mutation score are some aspects that still need to be improved upon.

6.2 Future Work

The future work of this dissertation should be focused on revising and improving some implementation aspects. One of the aspects is the static metadata creation for the target CUT. This metadata definition should be dynamic in order to facilitate the range of this work for different SUT structures. Another associated limitation is the lack of assertion phases, where

the generated unit tests do not include a behavioral assessment. The unit tests behavior is not verified, and consecutively it's harder to determine if the tests should pass or fail. An attempt to add an automated generation of test oracles through either tools or novel approaches must be sought after. Additionally, a parameter tuning technique for the initial GA configuration, more concretely for the GA parameters defined in the configuration file, should be studied. In this way, an ideal configuration for the complexity of the SUT can be defined initially, giving the GA a good starting point for the search.

Bibliography

[1] V. Khorikov, *Unit Testing: Principles, Practices and Patterns*, ser. ISBN: 9781617296277. Manning Publications, 2020. [Online]. Available: https://www.manning.com/books/unit-testing xix, xxi, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 108, 109, 119

[2] S. McConnell, *Code complete*. Microsoft Press, 2016. xix, 8, 9

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213000563 xix, 18, 19

[4] A. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999. xix, 29

[5] A. Eiben and J. Smith, *Introduction to evolutionary computing*. Springer-Verlag, 2015. xix, 22, 23, 25, 26, 27, 28, 31, 32, 33, 68, 72, 74, 99, 104, 120, 121

[6] RST. Actionable ways to achieve faster time to market (ttm) and release products before your competitors. Last accessed on September 28, 2023. [Online]. Available: https://www.rst.software/blog/actionable-ways-to-achieve-faster-time-to-market-ttm-and-release-products-before-your-competitors 1

[7] C. H. House and R. L. Pric. The return map: Tracking product teams. Last accessed on September 28, 2023. [Online]. Available: https://hbr.org/1991/01/the-return-map-tracking-product-teams 1

[8] G. Gurung, R. Shah, and D. Jaiswal, "Software development life cycle models-a comparative study," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pp. 30–37, 07 2020. 1

[9] T. Kurmaku, E. P. Enoiu, and M. Kumrija, "Human-based test design versus automated test generation: A literature review and meta-analysis," in *15th Innovations in Software Engineering Conference*, ser. ISEC 2022. Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3511430.3511433 1, 17, 18

[10] M. Magalhães, A. Morgado, H. Jesus, and N. Pombo, "Unlocking the potential of dynamic languages: An exploration of automated unit test generation techniques," in *2023 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2023, pp. 122–126. 2

[11] A. S. for Quality. What is software quality? Last accessed on October 4, 2023. [Online]. Available: https://asq.org/quality-resources/software-quality 8

[12] I. O. for Standardization. Iso 25010. Last accessed on October 4, 2023. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 8

[13] M. N. Aziz, "Properties of Good Unit Tests for Software Quality Assurance," *International Journal of Science and Business*, vol. 4, no. 5, Jun. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3884317 8

[14] S. Ducasse, D. Pollet, A. Bergel, and D. Cassou, "Reusing and composing tests with traits," in *Objects, Components, Models and Patterns*, M. Oriol and B. Meyer, Eds. Springer Berlin Heidelberg, 2009, pp. 252–271. 9

[15] M. Cneude. What is software entropy and how to manage it? Last accessed on October 4, 2023. [Online]. Available: https://thevaluable.dev/fighting-software-entropy/ 9

[16] J. C. M. de Campos, "Search-based unit test generation for evolving software," Ph.D. dissertation, University of Sheffield, November 2017. [Online]. Available: https://etheses.whiterose.ac.uk/20271/ 14, 18

[17] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," vol. 38, 07 2010, pp. 147–158. 14, 109

[18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. 14

[19] S. Bose. Code coverage vs test coverage. Last accessed on October 6, 2023. [Online]. Available: https://www.browserstack.com/guide/code-coverage-vs-test-coverage 15

[20] S. T. Help. Black box testing: An in-depth tutorial with examples and techniques. Last accessed on October 9, 2023. [Online]. Available: https://www.softwaretestinghelp.com/black-box-testing/ 17

[21] S. T. Help. White box testing: A complete guide with techniques, examples, & tools. Last accessed on October 11, 2023. [Online]. Available: https://www.softwaretestinghelp.com/white-box-testing-techniques-with-example/ 17

[22] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli, "On the effectiveness of manual and automatic unit test generation: Ten years later," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 121–125. 18

[23] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, "How do automatically generated unit tests influence software maintenance?" in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 250–261. 18

[24] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 201–211. 18

[25] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 215–222, 1976. [Online]. Available: https://api.semanticscholar.org/CorpusID:2940045 18

[26] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584901001896 20

[27] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163. 20

[28] A. Fontes, G. Gay, F. G. de Oliveira Neto, and R. Feldt, "Automated support for unit test generation: A tutorial book chapter," *CoRR*, vol. abs/2110.13575, 2021. [Online]. Available: https://arxiv.org/abs/2110.13575 20, 31, 44

[29] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Software Testing, Verification and Reliability*, vol. 29, no. 4-5, p. e1701, 2019, e1701 stvr.1701. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701 20, 21

[30] S. Balan. Metaheuristics in optimization: Algorithmic perspective. Last accessed on October 14, 2023. [Online]. Available: https://www.informs.org/Publications/OR-MS-Tomorrow/Metaheuristics-in-Optimization-Algorithmic-Perspective 21

[31] V. Chahar, S. Katoch, and S. Chauhan, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools and Applications*, vol. 80, 02 2021. 21, 23, 24

[32] D. Rocke, "Genetic algorithms + data structures = evolution programs by z. michalewicz," *Journal of the American Statistical Association*, vol. 95, pp. 347–348, 03 2000. 21, 22

[33] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test., Verif. Reliab.*, vol. 14, pp. 105–156, 06 2004. 22, 25

[34] D. Bhandari, C. Murthy, and S. Pal, "Variance as a stopping criterion for genetic algorithms with elitist model," *Fundamenta Informaticae*, vol. 120, pp. 145–164, 04 2012. 22, 28

[35] A. V. Kumar, "Encoding schemes in genetic algorithm," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:44159121 23

[36] S. N. Sivanandam and S. N. Deepa, *Introduction to genetic algorithms*. Springer, 2008. 24, 25, 27, 28, 30, 119

[37] Y. Fang and J. Li, "A review of tournament selection in genetic programming," in *Advances in Computation and Intelligence*, Z. Cai, C. Hu, Z. Kang, and Y. Liu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 181–192. 25

[38] H. Xie, M. Zhang, P. Andreae, and M. Johnson, "An analysis of multi-sampled issue and no-replacement tournament selection," 07 2008, pp. 1323–1330. 26

[39] A. Fialho, "Adaptive operator selection for optimization," 12 2010. 29, 30

[40] A. Eiben, M. Schut, and A. Wilde, "Is self-adaptation of selection pressure and population size possible? – a case study," 01 2006, pp. 900–909. 31, 38, 39, 81

[41] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, 1986. 31

[42] E.-G. Talbi, *Metaheuristics: From design to implementation.* John Wiley & Sons, 2009. 32, 33, 58, 59

[43] E. Marchiori and C. Rossi, "A flipping genetic algorithm for hard 3-sat problems," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, ser. GECCO'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 393–400. 32

[44] J. Arabas, Z. Michalewicz, and J. Mulawka, "Gavaps-a genetic algorithm with varying population size," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, 1994, pp. 73–78 vol.1. 35, 36, 75

[45] A. Eiben, E. Marchiori, and V. Valko, "Evolutionary algorithms with on-the-fly population size adjustment," 09 2004, pp. 41–50. 36, 75

[46] G. Harik and F. Lobo, "A parameter-less genetic algorithm," 01 1999, p. 258–265. 36

[47] T. Bäck, A. E. Eiben, and N. A. L. van der Vaart, "An emperical study on gas "without parameters"," in *Parallel Problem Solving from Nature PPSN VI*, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 315–324. 36, 37, 42, 51, 58, 62, 65, 69, 75, 88, 89, 94

[48] B. Rajakumar and A. George, "Apoga: An adaptive population pool size based genetic algorithm," *AASRI Procedia*, vol. 4, pp. 288–296, 2013, 2013 AASRI Conference on Intelligent Systems and Control. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212671613000449 37, 52, 58, 59, 60, 63, 67, 74, 75, 77, 78, 79, 80

[49] D. Pham and M. Castellani, "Adaptive selection routine for evolutionary algorithms," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 224, pp. 623–633, 01 2010. 38, 52, 58, 62, 63, 69, 80, 81, 82, 83, 84, 85, 86

[50] K. Jebari, "Parent selection operators for genetic algorithms," *International Journal of Engineering Research & Technology*, vol. 12, pp. 1141–1145, 11 2013. 39, 82

[51] A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri, and V. B. S. Prasath, "Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach," *Information*, vol. 10, no. 12, 2019. [Online]. Available: https://www.mdpi.com/2078-2489/10/12/390 39, 40, 58, 59, 65, 67, 86, 87, 95

[52] Y. Shi, R. Eberhart, and Y. Chen, "Implementation of evolutionary fuzzy systems," *IEEE Transactions on Fuzzy Systems*, vol. 7, no. 2, pp. 109–119, 1999. 41, 58, 59, 60, 65, 67, 87, 89, 96

[53] M. Srinivas and L. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994. 41, 42, 59, 89, 90

[54] T. Bäck and M. Schütz, "Intelligent mutation rate control in canonical genetic algorithms." 06 1996, pp. 158–167. 43, 52, 59, 67, 96

[55] S. Lukasczyk and G. Fraser, "Pynguin: automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. ACM, May 2022. [Online]. Available: http://dx.doi.org/10.1145/3510454.3516829 44

[56] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179 44

[57] M. D. Mifflin, S. T. S. Jeor, L. A. Hill, B. J. Scott, S. A. Daugherty, and Y. O. Koh, "A new predictive equation for resting energy expenditure in healthy individuals." *The American journal of clinical nutrition*, vol. 51 2, pp. 241–7, 1990. [Online]. Available: https://api.semanticscholar.org/CorpusID:45348378 48

[58] W. D. McArdle, F. I. Katch, and V. L. Katch, *Exercise physiology: Nutrition, energy, and human performance*. Lippincott Williams & Wilkins, 2010. 48

[59] R. MacPherson, "Energy expenditure: Tdee definition and calculator," https://www.verywellfit.com/what-is-energy-expenditure-3496103, October 2022, (Accessed on 04/11/2024). 49

[60] SteelFit, "What is my tdee (total daily energy expenditure)?" https://steelfitusa.com/blogs/health-and-wellness/calculate-tdee, (Accessed on 04/11/2024). 49

[61] N. H. Service, "Calorie counting - better health - nhs," https://www.nhs.uk/better-health/lose-weight/calorie-counting/, (Accessed on 04/11/2024). 49

[62] H. Krekel and P. dev team, "Full pytest documentation — pytest documentation," https://docs.pytest.org/en/7.1.x/contents.html, (Accessed on 04/22/2024). 53

[63] N. Batchelder, "Coverage.py — coverage.py 7.4.4 documentation," https://coverage.readthedocs.io/en/latest/index.html, (Accessed on 04/22/2024). 54

[64] "Git," https://git-scm.com/, (Accessed on 06/25/2024). 56

[65] A. Fontes and G. Gay, "Using machine learning to generate test oracles: A systematic literature review," *CoRR*, vol. abs/2107.00906, 2021. [Online]. Available: https://arxiv.org/abs/2107.00906 70

[66] A. Duque-Torres, C. Klammer, D. Pfahl, S. Fischer, and R. Ramler, "Towards automatic generation of amplified regression test oracles," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Sep. 2023. [Online]. Available: http://dx.doi.org/10.1109/SEAA60479.2023.00058 70

[67] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 201–211. 70

[68] D. C. Bento, "Fault revealing test oracles, are we there yet? evaluating the effectiveness of automatically generated test oracles on manually-written and automatically generated unit tests," Master's thesis, FACULDADE DE CIÊNCIAS DA UNIVERSIDADE DE LISBOA, 2022. 70

[69] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: a neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. ACM, May 2022. [Online]. Available: http://dx.doi.org/10.1145/3510003.3510141 70

[70] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Germany: Springer, 2012. 71, 72

[71] R. Czabanski, M. Jezewski, and J. Leski, *Introduction to Fuzzy Systems*. Cham: Springer International Publishing, 2017, pp. 23–43. [Online]. Available: https://doi.org/10.1007/978-3-319-59614-3_2 88

[72] E. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International Journal of Man-Machine Studies*, vol. 7, no. 1, pp. 1–13, 1975. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737375800022 88

Appendix A

Annexes

A.1 Dissimilarity between London and Classical Approaches

	Isolation of	Unit Test	Test doubles for
London School	Units	Is a class	All but immutable dependencies
Classical School	Unit tests	Is a class or set of classes	Shared dependencies

Table A.1: Isolation take between London and Classical school approaches [1].

A.2 Roulette Wheel Selection Pseudocode

Algorithm 1 Pseudocode for roulette wheel selection algorithm [36]

```
Calculate the sum of fitness scores of each individual in the population. Let it be S.
for N amount of times do
    Pick a random number r between 0 and S
    for P amount of times (according population size P) do
        Accumulative sum of fitness scores of each individual during the loop. Let it be AS
        if AS ≥ r then
            Select the current individual in the loop for recombination
            Break current loop
    Continue loop until N amount of times is reached
```

A.3 Rank Selection Pseudocode

Algorithm 2 Pseudocode for rank selection algorithm [36]

```
Sort population in ascending order by fitness score to consider ranking assignment (within
the interval [1, ..., N] from the weakest rank 1 to the strongest rank N)
while Pair of parents has not yet been selected do
    Random selection of a pair of individuals from the population.
    Obtain the ranking from both of them. Let the rank of the first individual be RF and
    the rank of the second individual be RS
    if RF > RS then
        Select the first individual from the pair
    else
        Select the second individual from the pair
    Continue cycle until a pair of parents is selected
```

A.4 Tournament Selection Pseudocode

Algorithm 3 Pseudocode for tournament selection algorithm [5]

```
1: while Number of parents are not selected do
2:     Select randomly k individuals, with or without replacement
3:     Compare fitness scores from the selected individuals and choose the best one to be a
       parent
4:     Continue cycle until two parents are selected
```

A.5 One-Point Crossover Representation

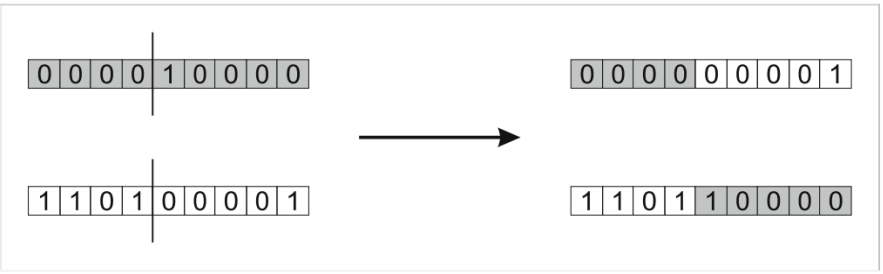


Figure A.1: One-Point crossover operation representation [5].

A.6 N-Point Crossover Representation

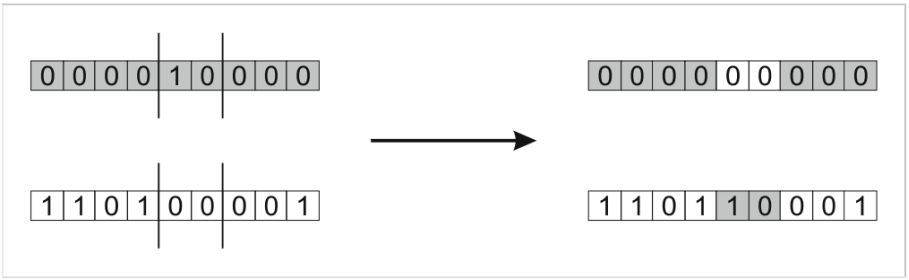


Figure A.2: N-Point crossover operation representation [5].

A.7 Uniform Crossover Representation

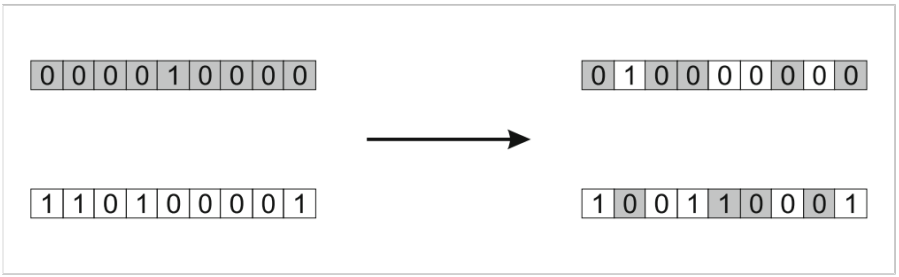


Figure A.3: Uniform crossover operation representation [5].

A.8 Mutation Bitwise Representation

Chromosome	1 0 **1** **0** 0 0 0 **1** 0
Mutated chromosome	1 0 **0** **1** 0 0 0 **0** 0

Table A.2: Mutation bitwise process.

A.9 Mutation Interchanging Representation

Chromosome	1 **0** 1 1 0 **1** 0 1
Mutated chromosome	1 **1** 1 1 0 **0** 0 1

Table A.3: Mutation interchanging process.

A.10 Mutation Reversing Representation

Chromosome	1 0 1 1 0 1 **0** **1**
Mutated chromosome	1 0 1 1 0 1 **1** **0**

Table A.4: Mutation reversing process.

A.11 Test in Java programming Language

```
1 public bool lowerThanTen = false;
2
3 public bool lowerNumber(int number)
4 {
5     boolean result = number < 10;
6     lowerThanTen = result;
7     return result;
8 }
9
```

```
public void test(){
    boolean result = lowerNumber(10);
    assertEqual(false, result);
}

public String integerToString(int number){
    return Integer.toString(number);
}

public void testString(){
    String message = integerToString(10);
    assertEqual("10", message);
}
```

Listing A.1: Test example in Java programming language.

A.12 Subset of the Best Generated Test Suite

```
from cut import *

def test_case_0():
  cut = calorie_intake_calc(122.74,156.97,76,'M',0.32,'V')
  cut.height = 165.67
  result_tdee_calculation = cut.tdee_calculation()
  cut.age = 84
  cut.age = 21

def test_case_5():
  cut = calorie_intake_calc(134.35,177.29,11,'M',-0.4,'L')
  cut.weight = 57.79
  result_katch_mcardle_equation = cut.katch_mcardle_equation()
  result_mifflin_stjeor_equation = cut.mifflin_stjeor_equation()
  cut.weight = 51.86

def test_case_8():
  cut = calorie_intake_calc(200.53,218.11,77,'N',0.22,'L')
  result_tdee_calculation = cut.tdee_calculation()
  cut.weight = 35.08
  result_katch_mcardle_equation = cut.katch_mcardle_equation()

def test_case_9():
  cut = calorie_intake_calc(127.2,139.08,19,'M',0.52,'N')
  result_katch_mcardle_equation = cut.katch_mcardle_equation()
  cut.height = 148.69

def test_case_11():
  cut = calorie_intake_calc(101.38,170.44,61,'N',-0.22,'E')
  cut.gender = 'F'
```

```
31    result_katch_mcardle_equation = cut.katch_mcardle_equation()
32    result_katch_mcardle_equation = cut.katch_mcardle_equation()
33    cut.height = 172.69
34
35  def test_case_12():
36    cut = calorie_intake_calc(95.34,183.86,25,'F',-0.11,'N')
37    result_tdee_calculation = cut.tdee_calculation()
38    cut.age = 73
39    result_mifflin_stjeor_equation = cut.mifflin_stjeor_equation()
40
41  def test_case_13():
42    cut = calorie_intake_calc(105.33,173.45,76,'N',0.75,'L')
43    result_determine_calorie_intake = cut.determine_calorie_intake()
44    result_determine_calorie_intake = cut.determine_calorie_intake()
45    result_mifflin_stjeor_equation = cut.mifflin_stjeor_equation()
46    result_determine_calorie_intake = cut.determine_calorie_intake()
47
48  def test_case_17():
49    cut = calorie_intake_calc(36.28,204.86,14,'N',-0.42,'N')
50    result_determine_calorie_intake = cut.determine_calorie_intake()
51    cut.bodyfat = 0.33
52    cut.gender = 'F'
53
54  def test_case_20():
55    cut = calorie_intake_calc(94.88,144.69,10,'M',-0.4,'L')
56    cut.weight = 192.72
57    cut.gender = 'M'
58    cut.bodyfat = 0.06
```

Listing A.2: Subset of the best generated test suite from the SACGA execution.

A.13 Complete Flowchart of the Genetic Algorithm Execution

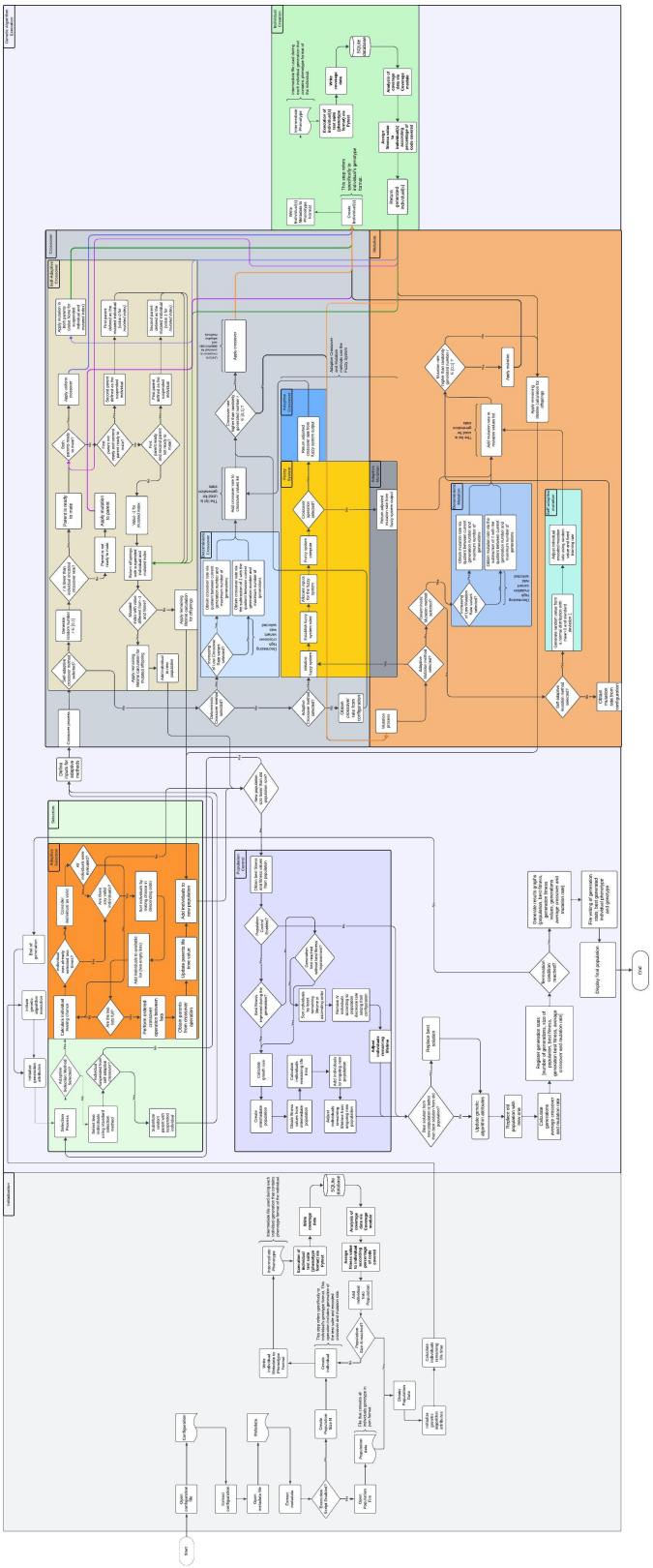


Figure A.4: Genetic algorithm execution flowchart.