Domício Araújo Pereira Neto

EVOLUTIONARY SOFTWARE TESTS SUBSET OPTIMIZATION

Dissertation in the context of the Master's in Informatics Engineering, Specialization in Intelligent Systems, advised by Prof. João Nuno Correia and co-advised by Prof. Fernando Penousal Machado and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2023

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Evolutionary Software Tests Subset Optimization

Domício Araújo Pereira Neto

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Prof. João Nuno Correia and co-advised by Prof. Fernando Penousal Machado and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2023

1 2 9 0

UNIVERSIDADE Ð COIMBRA

Abstract

Software testing constitutes a crucial aspect of any system's life cycle. However, the process demands substantial time and resources, leading to the creation of a large amount of tests for requirement validation. Over time, test suites can expand to an extent that renders them impractical for utilization within restricted time frames. This scenario requires the employment of optimization techniques to strategically select the most pertinent tests for early execution, ensuring effective testing in constrained contexts. Altice, a participant in this project, is directly engaged in the development and maintenance of Optical Line Terminations (OLTs), essential components within their Passive Optical Networks (PONs). These networks provide internet, telephone, and television services to an constantly-growing number of customers. The testing procedure for OLTs encompasses thousands of tests, presenting a substantial demand for both time and human resources. Moreover, black-box testing is used, with tests composed in a high-level language, introducing challenges in evaluating and selecting tests. Thus, a new approach was required to optimize the test subset selection process of OLTs. In this context, this work introduces an evolutionary approach for optimizing test subsets in the distinctive OLT scenario. This approach is founded upon Genetic Algorithms (GAs) and has been validated through extensive experimentation. It proficiently selects subsets of tests using limited computational resources within reasonable time frames, and its adaptability to progressing test evaluation criteria and execution constraints further enhances its viability. The outcomes obtained underscore the potential applicability of this approach in real-world scenarios.

Keywords

Software Testing, Test Subset Optimization, Optical Line Termination, Genetic Algorithm

This page is intentionally left blank.

Resumo

Teste de software constitui um aspecto crucial no ciclo de vida de qualquer sistema. No entanto, o processo exige tempo e recursos substanciais, levando à criação de uma grande quantidade de testes para validação dos requisitos. Com o tempo, conjuntos de testes podem expandir-se a ponto de torná-los impraticáveis para utilização dentro de prazos restritos. Este cenário requer a aplicação de técnicas de otimização para selecionar estrategicamente os testes mais pertinentes para execução precoce, assegurando testagem eficaz em condições limitadas. A Altice, participante neste projeto, está diretamente envolvida no desenvolvimento e manutenção de Terminações de Linha Óptica (OLTs), componentes essenciais em suas Redes Ópticas Passivas (PONs). Essas redes fornecem serviços de internet, telefone e televisão para um número constantemente crescente de clientes. O procedimento de teste para OLTs engloba milhares de testes, apresentando uma demanda substancial tanto de tempo quanto de recursos humanos. Além disso, a testagem segue um paradigma black-box, com testes compostos em uma linguagem de alto nível, o que introduz desafios na avaliação e seleção dos testes. Assim, uma nova técnica foi solicitada para otimizar o processo de seleção de subconjuntos de teste de OLTs. Nesse contexto, este trabalho apresenta uma abordagem evolucionária para otimizar subconjuntos de teste no cenário distinto de OLTs. Essa abordagem é baseada em Algoritmos Genéticos (GAs) e foi validada por meio de experimentação extensiva. Ela seleciona de maneira eficiente subconjuntos de testes usando recursos computacionais limitados dentro de prazos razoáveis, e sua adaptabilidade a critérios de avaliação de testes e limites de tempo em constante mudança reforça a sua viabilidade. Os resultados obtidos destacam a aplicabilidade potencial desta abordagem em cenários do mundo real.

Palavras-Chave

Teste de Software, Otimização de Subconjuntos de Testes, Terminação de Linha Óptica, Algoritmo Genético

Acknowledgments

I extend my sincere gratitude to my advisor, Professor João Correira, and co-advisor, Professor Fernando Penousal, for their unwavering patience and support throughout this year. I'd also like to express my appreciation to my defense committee, whose valuable feedback was instrumental in guiding this work in the right direction. Furthermore, I want to convey my thanks to the OTA project and all participants from Altice Labs, particularly João Carreira, who promptly provided all the necessary assistance. I'm also thankful to my colleague, Inês Teixeira, whose constant presence and encouragement were a source of motivation during the course of this project.

Lastly, I want to express my heartfelt gratitude and special acknowledgment to my family, particularly my parents, without whose unwavering support, I wouldn't have reached this point. I extend my appreciation to my wife, Catarina, who stood by me during the challenging moments and never allowed me to falter, and to my children, that serve as a constant source of motivation in my professional journey. I'm grateful to my extended family, friends, and my church community for their support. Above all, I offer my gratitude to God, who has been my guiding force throughout these years. I hope that my future as a professional might be worthy of the support that I have received from all of those around me.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

AST	Abstract Syntax Tree. 11
BCO	Bee Colony Optimization. 13
EA	Evolutionary Algorithm. 11
GA	Genetic Algorithm. xiv, 1, 3, 11–17, 21, 23–25, 27, 28, 30, 33
GrA	Greedy Algorithm. 9, 10, 12
HAC	Hierarchical Agglomerative Clustering. 12
LLM	Large Language Model. 30, 31
OLT	Optical Line Termination. 1–7, 9, 16–18, 21, 22, 29–31, 33
ONU	Optical Network Unit. 1, 5, 6, 21, 30
PON	Passive Optical Network. 1, 4–6, 33
PSO	Particle Swarm Optimization. 13
TCP	Test Case Prioritization. 2, 9, 11–14
TSO	Test Suite Optimization. 2, 9, 11–14

This page is intentionally left blank.

List of Figures

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Automated software test selection is a crucial aspect of modern software development, aiming to optimize the testing process by running a carefully selected subset of tests that provides comprehensive coverage and efficient defect detection [1]. As software systems become more complex, executing the entire test suite becomes impractical and time-consuming. Software testing may account for between 15% to 25% of the project lifecycle cost as test suites can contain an extensive number of test cases [2], sometimes reaching the tens of thousands, highlighting the need for efficient strategies to quickly and reliably validate software changes [3].

This research focused on developing automatic software test selection specifically for the Optical Line Termination (OLT) system. OLTs are of significant importance in fiber optic communication networks, particularly in the context of Passive Optical Network (PON) [4]. The main function of an OLT is to act as a central hub that connects multiple subscribers' premises to the service provider's core network. The OLT is critical in converting electrical signals from the service provider into optical signals, enabling their transmission over fiber optic cables to reach the Optical Network Units (ONUs) located at the subscribers' end. Due to their pivotal role in PON architectures, OLTs have gained widespread adoption in modern fiber-optic communication infrastructures.

Altice Labs, the research arm of the multinational telecommunications company Altice, is one of the stakeholders of this project. Altice offers a wide range of services through PONs, including television, internet, and telephone services. As part of their operations, Altice Labs is responsible for developing and maintaining OLT software and hardware, although this work focuses only in software testing. This leads to the creation of a significant number of tests to validate new or modified functionalities. However, the sheer volume of tests makes it increasingly difficult to execute all of them within the available time frames. Hence, there is a need for an automatic test selection process capable of choosing the best set of tests to be executed within a given time window.

In this particular scenario, OLTs are subjected to black-box testing, meaning there is no direct access to their underlying source code. Instead, the testing process relies on test scripts that interact with the OLTs through an API. This approach allows testers to modify specific parameters and configurations while also evaluating the OLT's communication with peripheral devices, such as ONUs. Conducting black-box testing is an essential step to validate the OLT's requirements before its deployment and use by service providers.

In this work, an automated approach was developed based on Genetic Algorithms (GAs) to perform test subset optimization. The selection process was guided by specific criteria, such as subset execution time and diversity of tested requirements. Similar methodologies are frequently devised in the domain of Test Case Prioritization (TCP) and Test Suite Optimization (TSO) research [5, 6]. Although both subjects center on test selection, they serve distinct purposes. TCP revolves around determining the order of test cases for execution, guided by specific criteria. Conversely, TSO involves refining the selection of test cases by identifying a subset that optimizes testing efficiency. Both approaches may take into account factors such as code coverage, redundancy reduction, and execution time. However, TSO seeks to permanently curtail the test suite, while TCP focuses on prioritization, without necessarily removing tests. The developed algorithm was constructed based on the test selection principles of TCP and TSO, with the aim of maximizing the number of tests that are executed within a given time frame.

1.1 Objectives

The main objective of this work is to **create an algorithm for optimizing test subsets for OLTs**. This optimization involves respecting the test set execution time limit and test evaluation criteria defined by testers. The algorithm should, nonetheless, be adaptable to changing evaluation methods.

The implementation of such algorithm must enable the extraction of test features that can be used in the selection process, which is based solely on these features and historical execution times. Moreover, the execution of the algorithm itself should not be costly, both in terms of time and computational resources.

Validation of the algorithm with real OLTs is required in order to better evaluate the effectiveness of the approach. This involves comparing the estimated execution time of provided optimized subsets with the real values obtained during validation, which also implies the execution of subsequent selection iterations when the time frames are not fully used during subset execution.

1.2 Planning

The first semester of this project was dedicated to comprehending the intricate OLT testing process. This task involved numerous interactions with Altice personnel, spanning the entirety of the first semester and extending into the second. The Gantt chart for the first semester is depicted in Figure 1.1.

The tasks represented in the figure commenced by installing an OLT emulator, aimed at familiarization with the system's functioning. Subsequently, all tests provided by Altice Labs underwent execution within the emulator to provide insights into the testing process. This phase extended over several weeks. Subsequent steps encompassed a comprehensive review of the state of the art, followed by the initial attempt to construct a test generator, that would serve as an intermediate step to the implementation of the test subset optimizer. The sole prototype that materialized was the random test generator, capable of crafting rudimentary tests featuring a single step. The creation of a more complete test generator was unfeasible due to the high complexity involved in test composition and their relation to tested requirements. Eventually, the first semester report incorporated outcomes derived from the prototype of the random generator and test execution results obtained with the OLT emulator. Next, Figure 1.2 shows the Gantt chart for the second semester.

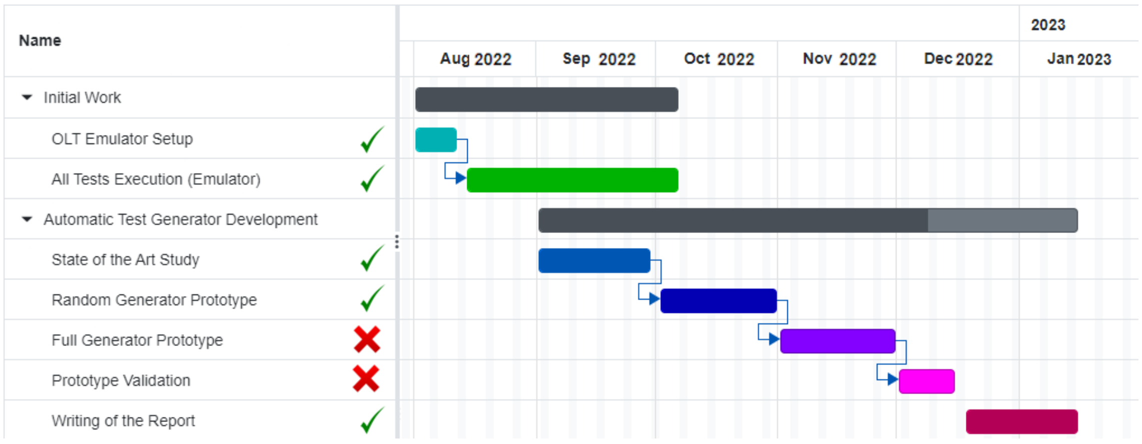


Figure 1.1: First semester Gantt chart.

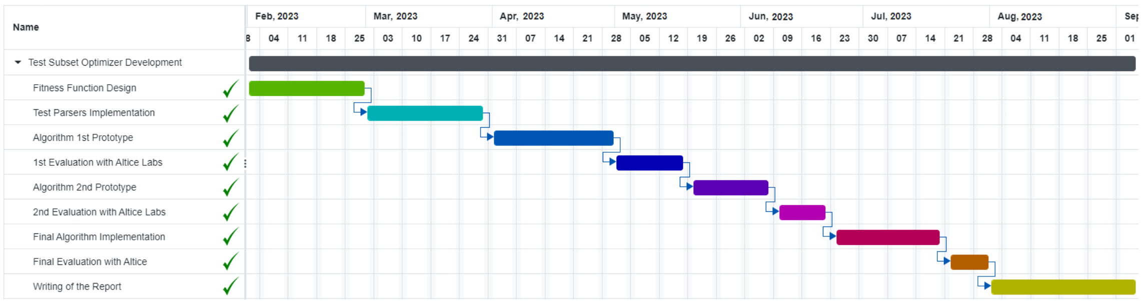


Figure 1.2: Second semester Gantt chart.

To facilitate the implementation of the test subset optimizer, a close engagement with Altice was maintained. Commencing in February, a visit to Altice's headquarters marked the start of efforts to comprehend how tests could be assessed for their perceived value and utility. This comprehension laid the base for devising a fitness function for the GA. Subsequent stages encompassed the creation of text parsers to extract the required information needed for test evaluation. This process unfolded gradually due to the intricate syntax inherent in the high-level language of tests.

Ultimately, multiple development and evaluation iterations were carried out in collaboration with Altice. These interactive cycles led to the creation of the algorithm's final version. A crucial point in this process was establishing a real-world OLT test scenario within Altice's facilities, thereby validating the algorithm in real-world conditions. Subsequently, this report was written, containing the details of this work's development.

1.3 Contributions

This work contributes to the research of testing optimization with the following achievements:

- Presents a novel perspective on the test subset optimization by treating the OLT as the system under test and executing test selection exclusively grounded in the tests themselves, considering the system as a black-box.
- Introduces an evolutionary approach to address the aforementioned problem, validating it through practical implementation on a real-world OLT, thereby reinforcing

its real-world applicability and effectiveness.

- Provided the content for the writing of an paper which will share the obtained knowledge with the academic community.

1.4 Document Structure

In chapter 2, the background concepts regarding OLTs and PONs are introduced. Next, in chapter 3, the state of the art related to test selection techniques is presented. Furthermore, in chapter 4, the details of the approach developed in this work are given. Finally, in chapter 5, the experimentation setup, respective results and analysis are provided, followed by the final remarks in chapter 6.

Chapter 2

Background Concepts

This chapter introduces the fundamental concepts essential for the subsequent sections of this work. To provide a comprehensive understanding of the testing system used in this research, we begin by explaining the Optical Line Termination (OLT). This includes a detailed explanation of what an OLT is, how it functions, and its significance to Passive Optical Network (PON)s.

Following that, an overview is presented of the testing procedures in Altice Lab's OLTs. This explanation will shed light on how tests are designed and used, together with the associated challenges to evaluate and select them.

2.1 Passive Optical Networks and Optical Line Terminations

A PON is a telecommunications network architecture that utilizes optical fiber cables to deliver high-speed data, voice, and video services to end-users. Unlike conventional copper-based networks, PONs employ a passive infrastructure, eliminating the need for active electronic components like repeaters or switches in the distribution network. Instead, PONs leverage the inherent properties of light to transmit data over long distances without experiencing substantial signal degradation. This characteristic renders PON a more energy-efficient and cost-saving technology when compared to traditional network architectures [7].

At the heart of a PON is the OLT, acting as the central hub that connects the service provider's network to the customers' premises. The OLT utilizes passive optical splitters to divide a single optical fiber into multiple branches. These splitters can distribute the signal to Optical Network Units (ONUs) located at the customers' locations. Each ONU serves as the endpoint where individual subscribers connect their devices to access the network's services. PONs are known for their capability to provide high bandwidth, making them ideal for applications that require robust and reliable connectivity, such as high-definition video streaming, online gaming, and cloud-based services. Figure 2.1 shows an abstraction of the PON architecture.

There are several different models of OLTs depending on the manufacturer. In Altice's case, there are currently three models in production: the OLT2T0, OLT2T2 and the OLT2T4 (Figure 2.2). They differ in size, number of ports and processing capacity.

The OLT2T0 is the smallest of the OLT2Tx family, with 8 PON slots and no boards, specially designed to cover low density areas in urban, condominium and rural environments.

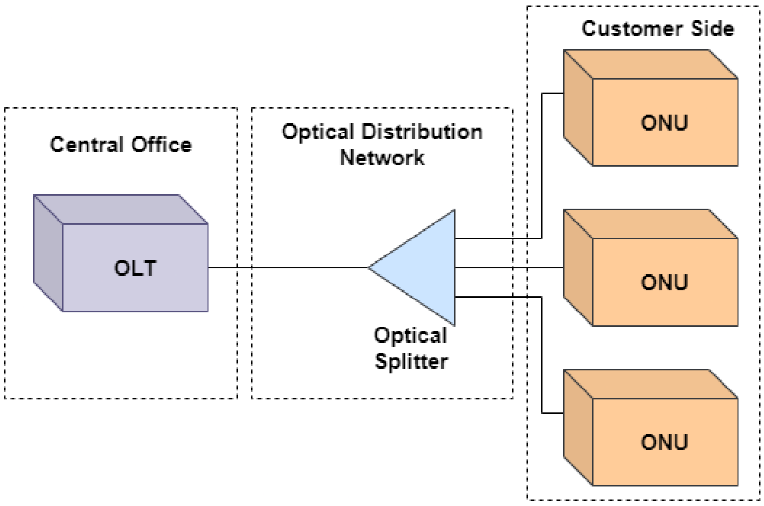


Figure 2.1: PON typical architecture representation.

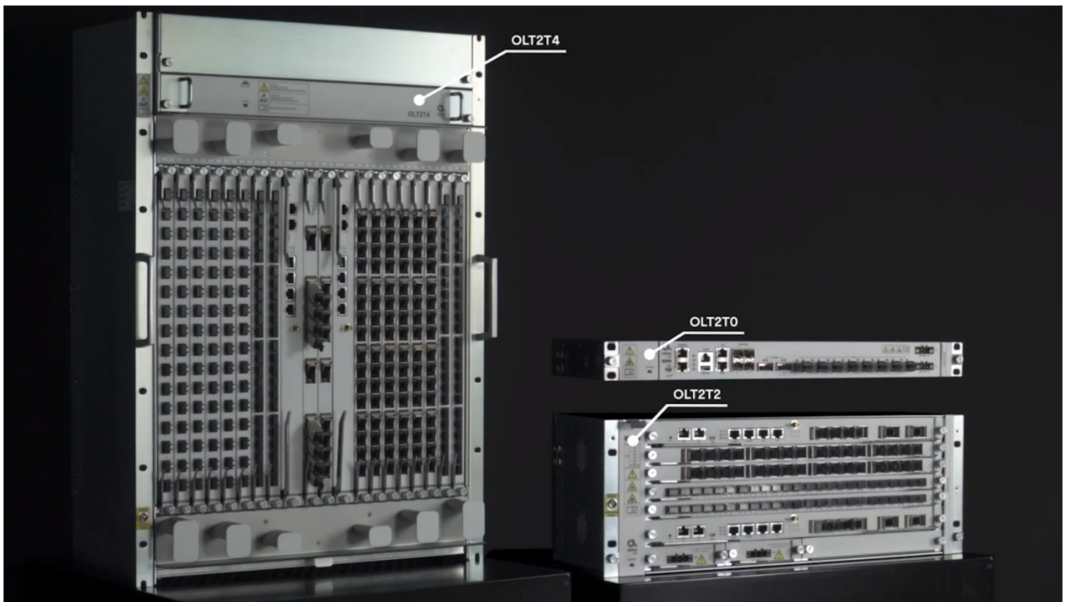


Figure 2.2: Altice's OLT2Tx family. [8].

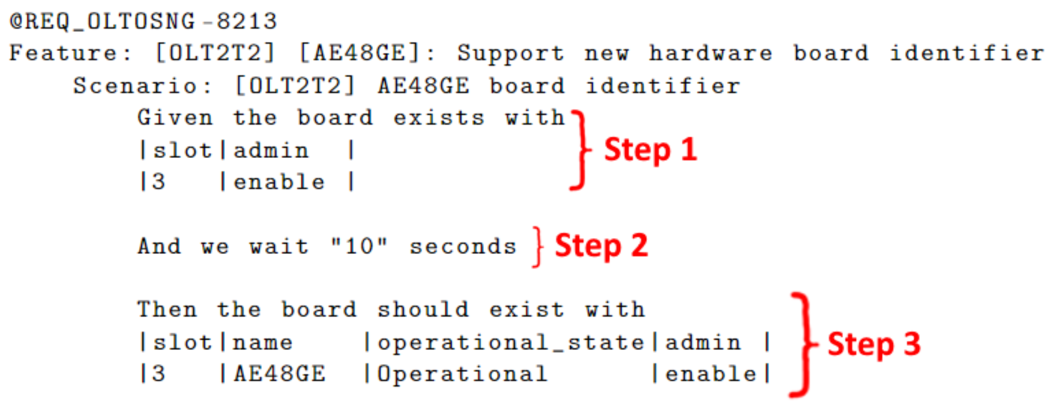
The tests used in this work were designed for this model, as Altice Labs provided a single OLT2T0 connected with 2 ONUs for experiments.

2.2 Test Design and Structure

The testing concerned in this work is related to the maintenance phase of an OLT model, i.e., when the model is already in full production and deployment, thus being occasionally updated for improving reliability, security, and adding new functionalities. Before a new update can be released to the deployed OLTs, it must be tested to validate the associated requirements. This is performed in a controlled environment with dedicated test equipment.

To test an OLT, Gherkin scripts are executed on a separate computer, allowing for remote interaction with the OLT's operating system. Gherkin is a language associated with Cucumber and is designed to support behavior-driven development (BDD) [9]. It enables the specification of software behavior in a logical manner, making it easier for customers and non-programmers to understand it. In this context, Gherkin acts as a bridge, mapping the underlying lower-level programs written in Ruby to a higher-level and more human-comprehensible language. This approach enhances the accessibility and readability of the testing process for a broader audience.

Every Cucumber *feature file* consists of a series of tests related to a specific requirement. Each test is described within a *Scenario* environment, aimed at creating and validating particular configurations and functionalities of the OLT. The following example in Figure 2.3 demonstrates a standard structure for an OLT test.



```
@REQ_OLTOSNG-8213
Feature: [OLT2T2] [AE48GE]: Support new hardware board identifier
    Scenario: [OLT2T2] AE48GE board identifier
        Given the board exists with
        |slot|admin  |
        |3   |enable |

        And we wait "10" seconds

        Then the board should exist with
        |slot|name    |operational_state|admin  |
        |3   |AE48GE  |Operational      |enable|
```

Step 1
Step 2
Step 3

Figure 2.3: OLT test example, written in the Gherkin language.

This simple test aims to confirm the responsiveness of the board connected to the OLT. It involves three steps with different purposes: (step 1) to configure the board, (step 2) to allow a short time for the configurations to take effect, and (step 3) to verify the outcome of the performed action. The test parameters are presented in a tabular format, where the first row specifies the parameter names, and the subsequent rows define their respective values. If the tested conditions are successfully validated, the test will conclude without any errors. However, in case an error occurs during the execution, a message will indicate the nature of the error and the faulty step where it happened, with the remaining steps not being executed.

There are thousands of tests with varying complexities and objectives, with many more being constantly created. However, the main objective of this work is not to understand the specific purpose of each individual test but instead to optimize the selection of tests in a generic and agnostic manner.

The challenges in evaluating and selecting these tests arise from their composition using a high-level language, which complicates the calculation of test similarity measures. Additionally, the difficulty of assessing the covered portions of the software is aggravated by the system's black-box nature during this phase of testing. As a result, applying commonly used test selection metrics like code coverage becomes unfeasible. However, a thorough description of the methodology employed for evaluating tests is provided later in Chapter 4.

This page is intentionally left blank.

Chapter 3

State of the Art Review

A literature review focusing on techniques for the automatic optimization of test subsets is given in this section. It is crucial to highlight that no published studies specifically focused on the unique scenario of OLT software test selection have been identified. This arises from the majority of studies primarily centered on white-box software testing, involving the direct testing of source code. Thus, the literature review requires a broader scope, enabling the adaptation of white-box test selection techniques to this specific context.

As mentioned before in Chapter 1, two research subjects closely related to this problem are Test Case Prioritization (TCP) and Test Suite Optimization (TSO). Both types of approaches revolve around the selection of tests from a larger pool, which constitutes the primary shared aspect with this work. The distinction, however, lies in their objectives: while TSO aims to minimize the test suite permanently by retaining only the most vital and least redundant tests, TCP focuses on ranking and reordering tests, without necessarily reducing the overall test set.

In the context of this work's approach, the tests are sourced from a backlog that requires expeditious clearance. All tests must be accommodated within limited time frames. The primary objective involves the selection of optimal test suites for execution within a designated time frame, without necessitating a reordering. It is acknowledged that certain tests may not be accommodated and instead are deferred for execution in the subsequent available time slot. In short, TCP and TSO techniques could, in theory, be adapted for this purpose.

In a review conducted by Kiran et al. [6], five primary types of approaches were identified: Greedy, Metaheuristic, Clustering, and General (comprising methods that do not align with any specific category). Next, an examination of each category, along with corresponding test selection examples, is provided.

Firstly, a **Greedy Algorithm (GrA)** constitutes a straightforward and intuitive strategy for tackling optimization problems [10]. This method revolves around making locally optimal selections at each algorithmic step, irrespective of potential ramifications or implications of that choice, with the aspiration that these selections will culminate in a globally optimal solution.

In the work of Wang et al. [11], the authors propose an GrA for reducing test suites while maintaining coverage and minimizing execution time. The approach is based on measuring and maximizing the tests' pairwise distance in terms of covered code statements to increase diversity in the test suite. The outcomes of the practical investigation demonstrate that the suggested method enables developers to pick a limited set of test cases, all the while maintaining the ability to attain proficient fault localization.

Furthermore, in the research conducted by Lin et al. [12], an investigation was undertaken to compare three distinct variants of the GrA. These variants encompassed one focused on maximizing coverage, a cost-aware adaptation of the latter, and another cost-aware variation incorporating an irreplaceability metric to reduce redundancy. The cost can be considered to be the test subset execution time. The findings of the study unveiled that the cost-aware techniques outperformed the standard GrA in various aspects, including cost reduction and the effectiveness of fault detection. However, it's worth noting that these cost-aware techniques may exhibit limitations in terms of scalability when the size and complexity of the test suite increases.

Broadly speaking, GrAs stand out as a popular option for addressing test subset optimization problems due to their simpleness and efficacy in generating favorable solutions. Nonetheless, an inherent drawback of this approach is its susceptibility to becoming stuck in local points within the search space, thereby yielding sub-optimal solutions [13]. This limitation, coupled with potential scalability constraints, renders GrAs less advantageous when confronted with complex test suite selection and minimization challenges. Therefore, it is important to examine other types of approaches.

In this regard, **Metaheuristic Algorithms** are optimization strategies utilized to tackle intricate issues that prove challenging or unfeasible to resolve using conventional mathematical or deterministic approaches. These algorithms draw inspiration from natural or social occurrences, leveraging both exploration and exploitation of the solution space to discover solutions of high quality. They are referred as "meta" because these algorithms guide the process of finding optimal or near-optimal solutions, bypassing direct problem resolution. Figure 3.1 shows the categorization of metaheuristic algorithms.

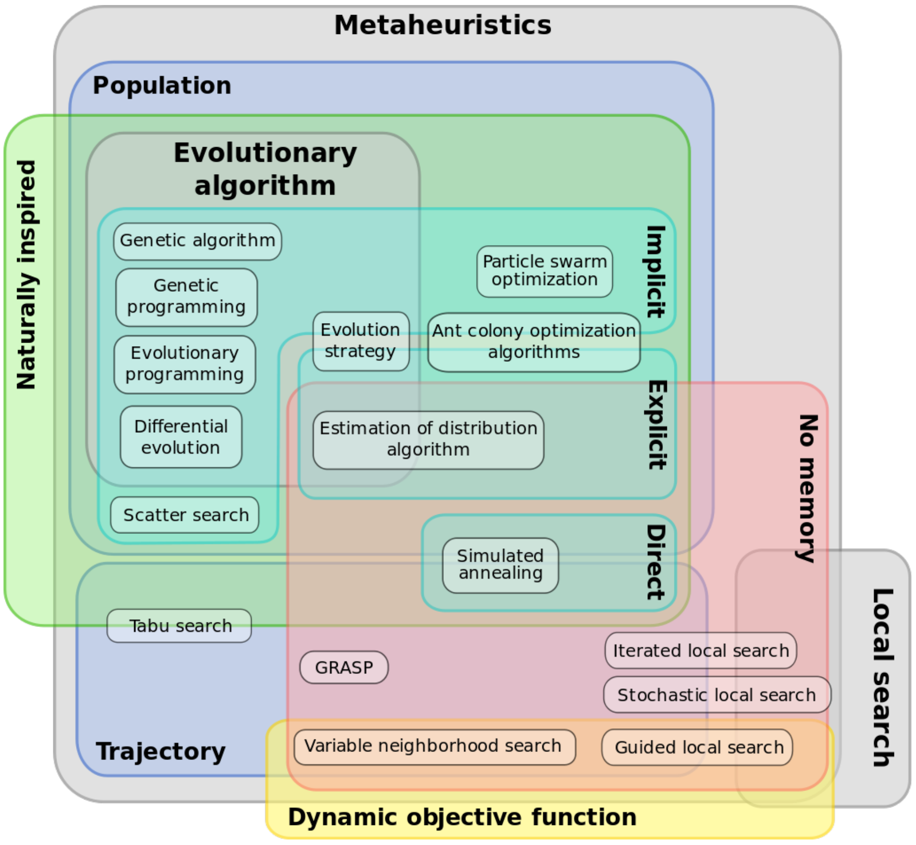


Figure 3.1: Metaheuristic algorithms classification [14].

As metaheuristic is very broad and diverse, this work focus on **Evolutionary Algorithms (EAs)**, which are also a very popular choice for TCP and TSO approaches, constituting a population-based subclass of methaheuristic [15]. EA takes its inspiration from Darwinian evolution theory for finding solutions for complex problems.

The basic functioning of EA revolves around the mechanisms of biological evolution, namely: reproduction, mutation, recombination, and selection. It is a cyclic process that starts by creating a random set of valid solutions. Then, the best individuals are selected for reproduction. The evaluation of an individual is made using a fitness function, which is manually designed according to the related optimization problem. In the reproduction step, new individuals are generated by recombination operations and mutation. Recombination involves combining the attributes of two individuals, named parents, to produce one or more individuals, named offspring. Mutation generates a new individual from a single parent by mutating a set of attributes from the parent. Finally, a new population is obtained from this reproduction process and the iterative cycle of evolution can continue until a certain criteria is met. Figure 3.2 presents a visual representation of the EA cycle.

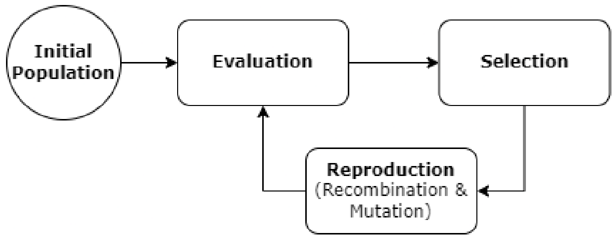


Figure 3.2: EA evolution cycle diagram.

In the context of subset optimization problems, Genetic Algorithm (GA) is the most popular option among EA techniques. The key distinction between GA and other evolutionary algorithms lies in the mechanisms used for genetic operations. While GA primarily centers on selection, crossover, and mutation, other evolutionary algorithms might incorporate distinct operators such as parent-centric recombination, gradient-based selection, or specialized mutation strategies [15].

An example of GA-based test selection approach is shown in the research conducted by Pan et al. [16]. In this study, the authors employ a multi-objective GA called Non-Dominated Sorting Genetic Algorithm II (NSGA-II) to reduce the size of test suites for 16 Java benchmark projects [17]. The evaluation of the tests is based on factors such as similarity, fault detection rate, and test execution time. In order to calculate similarity between tests, the tests were converted into the Abstract Syntax Tree (AST) format, enabling the calculation of characteristics such as the size of common subtrees between tests. The approach was successful in reducing test suits with minimization budgets ranging from 25% to 75%.

In the work of Lachmann et al. [18], further background and historical information are incorporated to reduce test suites using NSGA-II. This includes factors like the fault detection history of each test, quantity and significance of tested requirements, time elapsed since the last execution of each test, and more. Although this type of information can offer valuable insights for evaluating tests, our work is limited to only having access to test files and past execution times, without any additional details that can provide deeper test evaluation.

Ma et al. presents an GA-based method for reducing test suites [19], where fitness is calculated by considering test coverage and cost (execution time). The primary goal is to choose a subset that maintains coverage above a given minimum threshold while simultaneously decreasing both the overall execution time and the size of the test subset. In comparison to a cost-unaware GA and a GrA, the results demonstrated that the proposed approach outperformed the GrA and the cost-unaware GA, achieving superior outcomes in terms of both subset size and cost reduction. Additionally, the proposed approach exhibited faster performance than the cost-unaware GA, albeit slightly slower than the GrA, when it came to reducing the test suite duration. Despite the slightly longer time taken for suite reduction, the positive outcomes in size and cost reduction achieved by the proposed approach make it a favorable choice.

Numerous additional studies have delved into TCP and TSO utilizing GA techniques [20, 21, 22, 23, 24, 25], which establishes it as the prevailing option in this domain [6]. This prevalence can be attributed, in part, to the computational efficiency of GA, enabling resolution of highly complex problems with limited resources, while still yielding satisfactory solutions.

Although GA-based approaches hold a prominent position in testing optimization, **Clustering** stands out as another popular alternative technique that is worth mentioning. Clustering enhances the efficiency and effectiveness of testing protocols by grouping together similar test cases. This technique involves the arrangement of test cases into clusters, facilitating the identification and elimination of redundancy, thus enabling more efficient utilization of testing resources.

In the domains of TCP and TSO, clustering-based methods diverge primarily based on the features utilized for similarity computation and the types of clustering algorithms applied [26]. One of the most common types of clustering approaches in this context is Hierarchical Clustering, which involves the gradual merging or division of clusters, leading to the creation of a hierarchical arrangement often represented as a tree-like structure, known as dendogram. This method proves particularly advantageous when the number of clusters is not predetermined and when there's a desire to explore various potential cluster configurations [27].

In the work of Coviello et al. [28], the authors present a test suite reduction software based on Hierarchical Agglomerative Clustering (HAC) named CUTER (ClUstering-based TEst suite Reduction). This tool calculates the dissimilarity between pairs of tests based on statement coverage, then uses the HAC algorithm to group the tests into clusters. Finally, the program keeps the most representative test in each cluster, removing the remaining tests and producing a reduced test suite. This tool is deployed as an eclipse plugin and was successfully tested with 19 versions of 4 Java programs.

Another common clustering technique used in this context is K-means [29], which stands out as an unsupervised machine learning technique employed to cluster data into distinct groups or clusters. Its objective is to divide data points into K clusters, associating each data point with the cluster that possesses the nearest mean (centroid). This algorithm systematically enhances the allocation of data to clusters and adjusts centroids through iterative steps, continuing this process until a convergence point is reached.

Chetouane et al. introduces a TSO method utilizing the K-means algorithm [30], wherein similarity is computed using coverage as well. Empirical findings from experiments on 13 Java programs demonstrate the method's capacity to decrease test suites while upholding coverage minimums.

In essence, clustering algorithms present another viable option for implementing TCP and TSO methodologies. A significant feature is their deterministic nature, which can be an asset or liability, depending on whether users prioritize result consistency or diversity.

Still, certain researchers have ventured into combining multiple optimization strategies, leading to the development of **Hybrid algorithms** for TCP and TSO. Generally, this entails integrating both metaheuristic and traditional techniques. These hybrid algorithms leverage the inherent strengths of diverse optimization approaches, such as GAs, particle swarm optimization, clustering, among others. By combining these elements, these algorithms are capable of traversing the complex search space that encompasses potential test suites.

For example, in the work of Saraswat and Singhal [31], a GA is combined with Particle Swarm Optimization (PSO). PSO is a nature-inspired metaheuristic method that mimics the behavior of particles traversing a search space, with each particle representing a potential solution [32]. Guided by its own best-known position and the best solution found by the entire swarm, particles iteratively adjust their positions in pursuit of optimal solutions. The authors suggest using the resulting population from an preliminary GA run as an initial population for the PSO. The evaluation of tests is based on the Average Percentage Fault Detection (APFD) metric. This approach holds the potential to enhance the effectiveness of the PSO algorithm, given that commencing with a partially optimized initial population is more advantageous than initiating with a randomly generated population, as substantiated by previous research [33].

Another interesting work is presented by Singhal et al. [34], where an already-developed GA and Bee Colony Optimization (BCO) approach named MHBG_TCS is studied for test suite reduction. BCO represents yet another nature-inspired metaheuristic technique derived from the foraging actions of honeybees [35]. In this technique, artificial bees take the role of both employed and onlooker bees to navigate through the solution space of an optimization problem, effectively balancing the tasks of exploration and exploitation. BCO's key objective is to dynamically manage this balance, enabling the algorithm to discover optimal or highly promising solutions within the search space. In the hybrid adaptation of MHBG_TCS, the modification involves bees that discover solutions leading to reduced test suite execution times, which undergo a one-point crossover process, generating novel test subsets. The algorithm's evaluation encompassed 17 diverse programs implemented in languages such as C, C++, and Java. This comprehensive testing yielded favorable outcomes, including reductions in test suite size and execution time, alongside the preservation of fault detection capabilities.

Another possible combination, now involving clustering and GA, is presented by Pradhan et al. [36], named Cluster-Based Genetic Algorithm with Elitist Selection (CBGA-ES), derived from the NSGA-II algorithm. CBGA-ES's fundamental concept revolves around clustering a predefined set of potential solutions. From this population, elite solutions are chosen to generate offspring. This process involves gathering solutions with comparable fitness levels into clusters, arranging these clusters using the designed cluster dominance strategy, and subsequently selecting solutions from the highest-ranked clusters. These selected solutions become the basis for producing the next generation of offspring through reproduction.

In summary, hybrid algorithms offer several benefits, including accelerated convergence speed by harnessing the strengths of multiple algorithms, heightened exploration of the solution space resulting in more thorough test suite optimization, and increased robustness in tackling a broader array of problem types. Nevertheless, these advantages might be accompanied by elevated computational overhead due to algorithm fusion, potential additional complexity in parameter tuning, and the potential for inheriting constraints from individual algorithms.

While a considerable portion of testing suite reduction methodologies align with the categories mentioned earlier, it's worth noting that certain approaches deviate from these classifications. A number of **other** non-aligned approaches are introduced below.

In the work of Pradhan et al. [36], the authors propose a probability model-based approach to enhance test suite efficiency by minimizing redundancy, taking into account new and old software versions. The method utilizes Bayesian networks to represent the relationship between test cases, program faults, and executable paths. By transforming joint probability models into conditional probability models for both original and modified programs, a theorem is established to reduce the test suite size. Experimental results on 50 versions of a program demonstrate its ability to significantly reduce the test suite while maintaining equivalent fault detection capability

The study presented by Palomo-Lozano et al. introduces an approach that utilizes Integer Linear Programming (ILP) to minimize test suites while upholding mutation coverage [37], with specific emphasis on testing WS-BPEL programs. WS-BPEL is an XML-driven language for web service composition. The proposed technique exhibits efficacy and cost-consciousness in test suite minimization, even for intricate compositions. Given the resource-intensive execution demands involved in web service composition, it showed significant promising traits by curtailing testing cost and suite dimensions while maintaining coverage integrity.

Additional methods are accessible via open source software tools. Notably, DetReduce is made for Android GUI tests, as outlined by Choi et al. [38]. Another option is NEMO, which uses Integer Nonlinear Programming (INP) as its foundation [39].

It's noteworthy that a significant majority of these non-aligned alternatives are designed with highly specific use cases in mind. The distinctive nature of their implementations can potentially render the process of adaptation to other domains more complex. Given the presence of well-established methodologies highlighted in preceding sections, the adoption of these alternatives was considered not imperative.

3.1 Synthesis

GAs offer a distinct advantage over the studied techniques when it comes to implementing a TSO/TCP-like approach. Unlike greedy algorithms that make locally optimal choices at each step, GAs explore a broader solution space by simulating the principles of natural evolution. This enables them to efficiently navigate complex, high-dimensional search spaces and discover globally optimal or near-optimal solutions. Moreover, GAs are well-suited for handling non-linear and noisy objective functions, which are common challenges in test subset optimization. In contrast, clustering algorithms often rely on predefined assumptions about data distribution and may struggle with capturing underlying relationships in diverse test suites. GAs, with their diversity-preserving mechanisms and ability to adapt to changing conditions, such as varying test suite complexity, provide a more robust approach for test subset optimization, making them a preferable choice in scenarios where the goal is to find effective (e.g., in terms of fault detection or requirement coverage), yet efficient (e.g., in terms of subset size and execution time), test subsets.

While the creation of hybrid algorithms could be feasible, the straightforwardness of implementing standalone GA renders it a preferable choice over more intricate hybrid methodologies, unless it fails to yield satisfactory results. Thus, an evolutionary test subset optimizer based on GA was seen as the first choice among all possible algorithms.

Chapter 4

Proposed Approach

This chapter provides a comprehensive account of the approach's development. Given the selection of GA as the preferable technique, each stage of the algorithm's implementation is thoroughly outlined.

4.1 Overview

The mathematical formulation of the problem is presented below.

- Let T be the set of all available tests in a given backlog, where $|T|$ represents the total number of tests.
- Let U_i represent the utility of test $i \in T$. The utility can be any relevant measure that indicates the importance or effectiveness of the test.
- Let E_i represent the execution time of test $i \in T$.
- Let $S \subseteq T$ be the subset of tests that we want to select, and $|S|$ represents the number of tests in the subset S.

Then we have the following optimization problem, as follows:

$$\begin{aligned} \text{Maximize} \quad & z_1 = |S| \\ \text{Maximize} \quad & z_2 = \sum_{i \in S} U_i \\ \text{subject to} \quad & \sum_{i \in S} E_i \leq L \\ & S \subseteq T \end{aligned} \tag{4.1}$$

The goal is to create an approach to tackle this optimization problem, maximizing the number of tests in a subset S, alongside with the combined utility $\sum U_i$, while ensuring that the total execution time of the selected tests does not exceed a given time limit L. The utility metrics used in this work for measuring U_i are presented and explained further below. The general idea of this formulation is to be generic and easily adaptable to

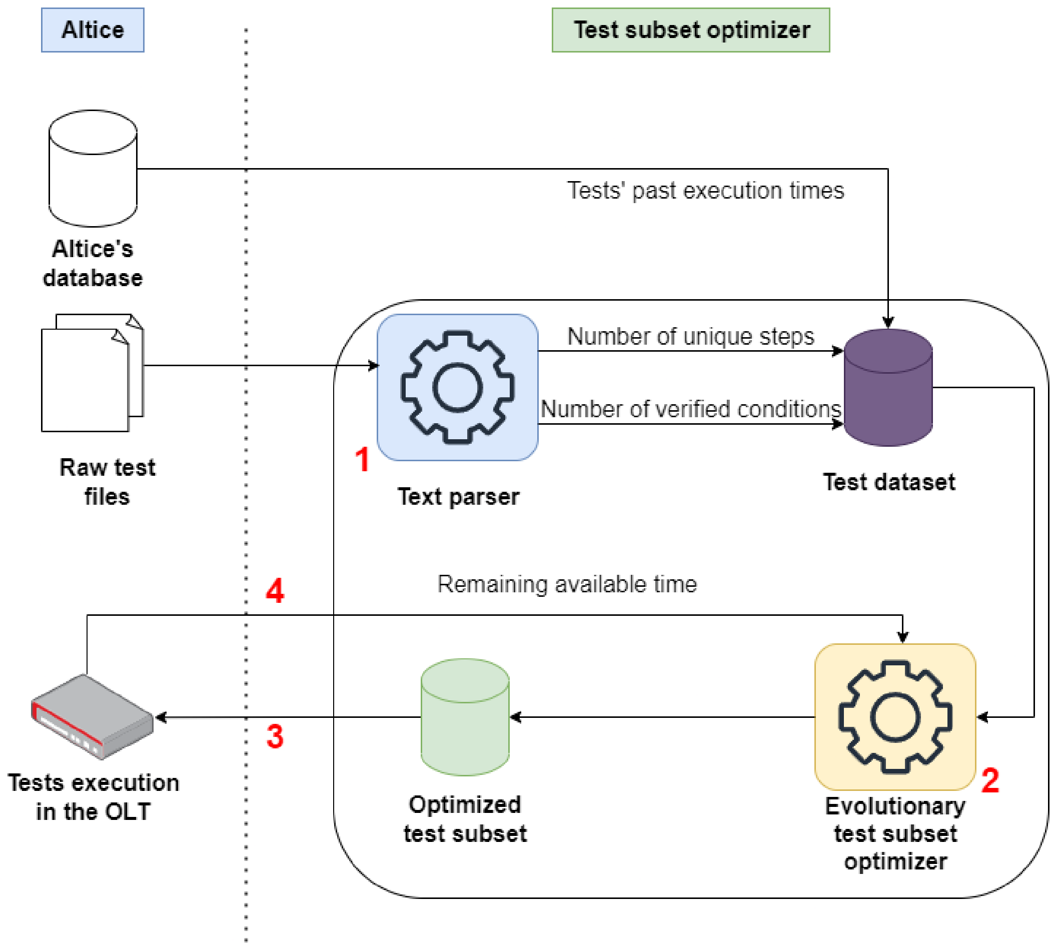


Figure 4.1: Framework diagram depicting the main steps of the pipeline, starting with (1) test text parsing, (2) subset optimization with the GA, (3) subset execution in Altice's OLT, and (4) possible subsequent optimization with remaining available time.

accommodate any future changes in utility evaluation methods. The proposed framework is shown in Figure 4.1.

The initial phase of the framework process involves (1) preprocessing the raw test files created by testers. This preprocessing stage involves extracting pertinent information used in test selection through text parsing. Additionally, external data on historical test execution times from Altice is incorporated into the generated dataset. Subsequently (2), this dataset is employed by the GA to optimize a subset of tests, taking into account user-defined execution time constraints. The resultant subset of tests is executed on Altice's OLT (3). In cases where unused execution time is observed, a subsequent iteration of the GA is triggered (4), using the remaining unexecuted tests.

To implement a GA-based algorithm for this problem, the initial step involves defining a genetic representation for the individuals. The genetic representation allows the process of reproduction, resulting in the creation of new individuals with varying genes. Additionally, a fitness function is essential to assess the individuals, enabling the selection of the most optimal candidates for reproduction and overall evolution of the population. The subsequent sections delve into these two aspects in detail.

4.2 Genetic Representation

An essential factor in GA development is determining a fitting representation. Common choices include binary or integer strings [40]. For representing a test set, a binary string was selected as the preferred approach, illustrated in Figure 4.2.

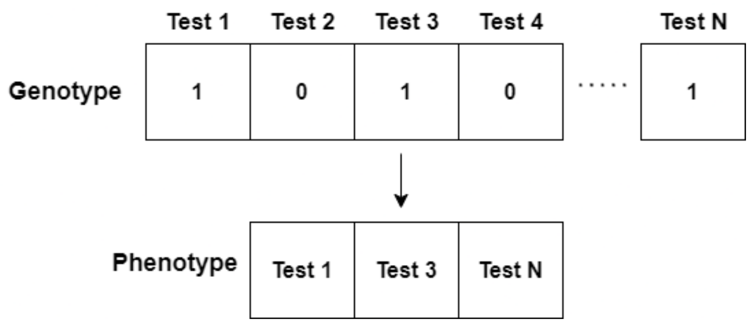


Figure 4.2: Binary genetic representation of a test set and the resulting phenotype, showing the selected subset.

The genotype functions as the genetic depiction of an individual, while the phenotype corresponds to the original representation that can be derived from the genotype [41]. As shown in Figure 4.2, the genotype consists of binary genes, where each gene signifies a distinct test within the provided set of tests. A gene's value denotes whether the corresponding test is encompassed within the set: 1 indicates inclusion, and 0 signifies exclusion. Consequently, the genotype can be transformed into the phenotype, which encompasses the roster of tests constituting the test subset.

4.3 Test Subset Evaluation

In GAs, a fitness function is used to evaluate individual solutions within a population. It assigns a numerical score to each solution, reflecting its alignment with the desired criteria or objectives of the problem. Through the assessment of individual fitness, GAs leverage this information to steer the evolutionary progression, promoting the choice and propagation of solutions boasting superior fitness scores. This iterative approach propels the population toward enhanced and increasingly optimal solutions across subsequent generations.

To assess a test set effectively, it's crucial to grasp the aims of the test selection procedure. These goals harmonize closely with the requirements of the OLT testers engaged in this project, resulting in the formulation of the subsequent guidelines:

1. The total execution time of the test set must stay within a limit defined by the tester.
2. The quantity of tests in the test set should be maximized while still meeting the preceding condition.
3. The test utility of the subset should also be maximized.

The first guideline highlights the primary limitation for testers, which is time. In our particular scenario, the tests are executed as a batch during non-business hours to guarantee the OLTs remain accessible for more interactive experiments. This temporal constraint constitutes the key determinant that establishes the boundary for executing a test set.

The second guideline underscores the importance of promptly executing all tests in the test backlog. Nonetheless, owing to time limitations, it might be frequently impractical to carry out all the tests in the backlog. Hence, this test selection procedure becomes essential to pinpoint the most valuable tests.

The third guideline introduces the notion of test utility, which may be a subjective aspect. It's vital to recognize that each test is precisely crafted to validate a specific requirement, and a test's value is intimately connected to the significance of its corresponding requirement. Given that the test selection method employed in this work treats the OLT as a black-box, lacking supplementary insights beyond the tests themselves, alternate criteria for evaluating test utility had to be formulated.

Following consultations with OLT test experts, a variety of evaluation metrics were established. It was determined that the value of a test set grows as it encompasses a larger array of distinct functionalities under validation, akin to code and requirement coverage seen in white-box testing examples from the previous chapter. Frequently, specific tests are formulated to validate the same functionality, albeit with differing actions or parameter permutations, resulting in a measure of redundancy among them. Essentially, augmenting the variety of tested functionalities within a set enhances its value.

Concerning the provided Cucumber/Gherkin files, the **number of unique steps** within a test set constitutes one of the two custom-defined metrics employed for utility assessment. In this context, a step pertains to a specific segment of Ruby code that executes diverse actions, such as queries and configurations on the OLT. With the creation of novel functionalities for the OLT, testers usually craft new Ruby programs, thereby generating fresh Gherkin steps to examine these functionalities, although some steps may reoccur for basic and recurrent configurations. Thus, it is considered that a step in a given test is unique if no other test in the test suite (i.e. considering all tests) contains the same step. It is apparent that a higher count of unique steps in a given subset exhibits a substantial, albeit not flawless, correlation with a diverse array of functionalities scrutinized within the test set.

The second metric employed for utility assessment is the **number of verified conditions** within a test. As numerous OLT functionalities are parameterized and can yield multiple outcomes, ensuring the proper operation of various configurations is vital. In the test code, these verifications are delineated by steps commencing with the keyword "Then". These steps are commonly accompanied by one or more lines defining different parameter combinations, which, in turn, results in the repeated execution of that same step for every combination. Put simply, a "Then step" in a Gherkin test file is regarded as a verified condition for each line of parameter combination it encompasses, or it is counted as a single verified condition if no parameters are involved.

It's crucial to highlight that these customized metrics derive from expert insights and possess a high level of technical abstraction. They remain susceptible to modifications or adjustments in the future to align with evolving testing needs.

The mentioned rules and criteria are employed to formulate a fitness function, generating a distinct value that quantifies the importance of a test set based on its attributes. In essence, the delineated criteria resulted in the creation of five components within the fitness function:

$$f_1 = 1 - \left| \frac{\text{test set exec. time}}{\text{time limit}} - 1 \right| \tag{4.2}$$

$$f_2 = \frac{\text{time limit} - \text{test set exec. time}}{\text{time limit}} \tag{4.3}$$

$$f_3 = \frac{\text{n. of verified conditions in the test set}}{\text{total n. of possible verified conditions}} \tag{4.4}$$

$$f_4 = \frac{\text{n. of unique steps in the test set}}{\text{total n. of possible unique steps}} \tag{4.5}$$

$$f_5 = \frac{\text{n. of tests in the test set}}{\text{total n. of tests}} \tag{4.6}$$

Table 4.1 explains in detail the properties of each component. It contains the purpose (or optimization objective) related to each of the five fitness components, followed by their calculated maximum and minimum attainable values. The maximum values pertain to the best possible optimization outcome for the given component, while the minimum value is related to the worst outcome.

Table 4.1: Explanation of the purpose of each fitness component, together with the maximum and minimum attainable value for each component.

Component	Purpose	Maximum Value	Minimum Value
f_1	Minimization of the absolute difference between the test set execution time and the defined time limit.	1	$2 - \frac{\text{max. possible test set exec. time}}{\text{time limit}}$
f_2	Minimization of the test set exceeded execution time.	1	$1 - \frac{\text{max. possible test set exec. time}}{\text{time limit}}$
f_3	Maximization of the number of verified conditions.	1	0
f_4	Maximization of the number of unique steps in the test set.	1	0
f_5	Maximization of the number of tests in the test set.	1	0

The first two components, f_1 and f_2, are interrelated and concern the execution time. f_1 strives to maintain the test set's execution time in proximity to the specified limit, thus minimizing any unused excess time. Conversely, f_2 operates as a counterpart, imposing penalties for instances where the execution time surpasses the limit.

Subsequently, the evaluation of test set utility takes place via f_3 and f_4, which address the quantity of verified conditions and unique steps (related to functionality/requirement diversity), respectively. Lastly, f_5 is devised to uphold the incorporation of a greater number of tests into the test set.

Finally, the fitness function for evaluating a test set is given by:

$$\text{fit} = \sum_{i=1}^{5} w_i * f_i \tag{4.7}$$

where f_i represents the ith fitness component, and w_i signifies its corresponding weight. These weights play a pivotal role in governing the impact of each component within the fitness function and were subject to manual adjustment during the experimentation phase.

Every fitness component was intentionally designed with its optimum value established at 1 (excluding the weights). Consequently, the act of maximizing the fitness function results in the partial optimization of distinct test subset attributes linked to each component. It's crucial to recognize that achieving the maximum value for all components simultaneously is unfeasible due to their concurrent nature. By calibrating the weights attributed to these components, specific features of the test subset can be enhanced, albeit at the expense of others.

The fitness function introduced in this section assumes a pivotal role in the developed evolutionary approach, acting as a guiding compass for the evolution process. Through the assignment of suitable weights, the fitness function enables the navigation of the search space toward satisfactory solutions.

Chapter 5

Experimentation

This chapter presents the experimental analysis conducted to investigate and validate the approach proposed in the previous chapter. The experiments were designed to gain insights into the efficacy and efficiency of the GA-based test subset optimizer.

5.1 Setup

While implementing this approach, a substantial collection of test files were supplied by the collaborating company. Yet, it was crucial to meticulously choose tests that could be efficiently conducted in a real OLT setting within the same company. The company was able to provide one OLT2T0 with two ONUs, a scenario compatible with a total of 296 tests. Figure 5.1 shows the pipeline of the experimental work.

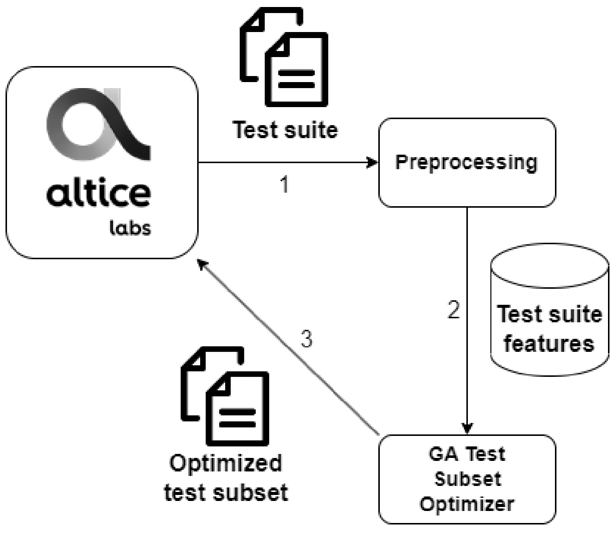


Figure 5.1: Test subset optimization pipeline, depicting the workflow carried out during experimentation, including (1) the receiving of a test suite from Altice Labs, followed by (2) preprocessing and (3) test subset optimization with the developed algorithm, which is sent back to Altice for validation.

Upon receiving the test suite from Altice Labs, an initial preprocessing phase is conducted to compute the attributes for each test, including the count of unique steps, the number of verified conditions, and the estimated test execution time. This involves parsing the tests and generating a roster of all distinct steps to assess the uniqueness of each test.

Furthermore, a counting of verification steps present in each test is also done by text parsing. Notably, the sole piece of information acquired from sources other than the tests themselves is the estimated execution time. These values are ascertained by averaging past execution times across these tests, which are kept in a database, from which Altice provided a snapshot.

Once the algorithm completes its run, the refined test subsets are transmitted back to Altice Labs. These subsets are subsequently executed within the real OLT test scenario that was mentioned before, to validate their execution time. Should the resulting execution time remain within the allocated time frame, an additional iteration of the algorithm is initiated. This subsequent iteration utilizes any remaining time and unused tests to further enhance the test selection process.

Now, the extracted features from the 296-test batch are analyzed. Figure 5.2 illustrates how execution times are distributed across the tests within the batch. Figure 5.3 presents the distribution of unique steps and verified conditions. Meanwhile, Figure 5.4 provides insight into the interplay between these three attributes.

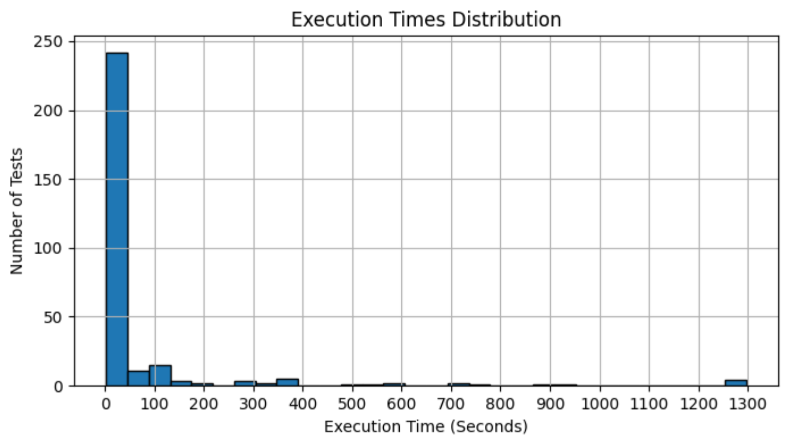


Figure 5.2: Execution times distribution between the 296 tests used in this work.

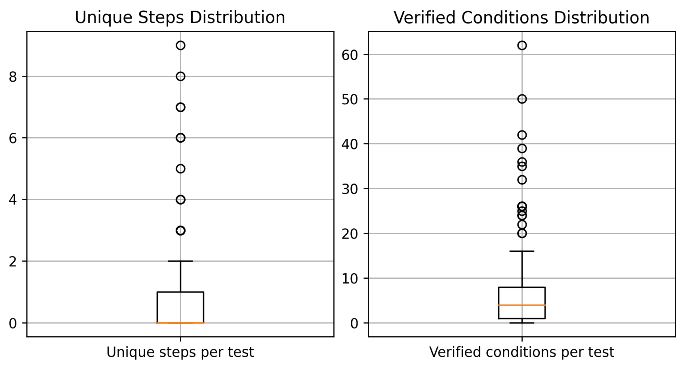


Figure 5.3: In the left plot, the boxplot representation of the distribution of unique steps in the 296-test suite, followed by a similar depiction in the right figure, regarding the number of verified conditions per test.

Regarding execution time, most tests have estimated duration of less than 50 seconds, with 95% of them concluding within ten minutes when combined. Concerning unique steps and verified conditions, the majority of tests encompass fewer than 10 verified conditions and between 0 to 2 unique steps. Upon examining the connection between these three aspects

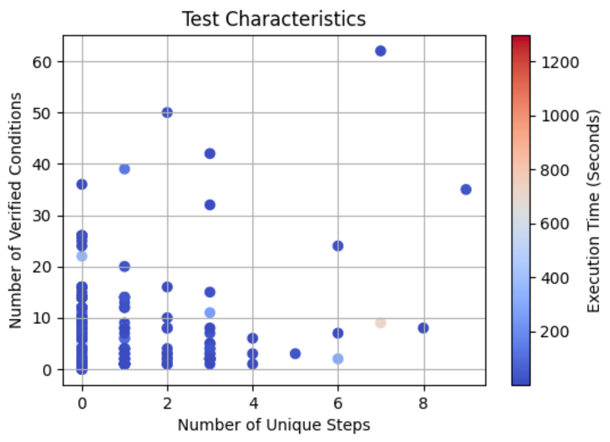


Figure 5.4: Visual analysis of the distribution of the three test features: number of unique steps in the x axis, number of verified conditions in the y axis, and estimated execution time represented by the blue-red color scale.

in Figure 5.4, it becomes apparent that the higher number of unique steps and verified conditions does not lead to higher execution times. Visually, there is no direct correlation between these attributes and execution time.

A **time limit** of 7200 seconds (2 hours) was established for conducting experiments using this test suite. This duration is less than half of the entire suite's execution time, which amounts to approximately 19770 seconds (5.5 hours). The reasoning behind this choice is tied to the time constraints set by Altice Labs. Test batch executions are carried out outside regular business hours, creating two available time slots: a 2-hour interval during lunchtime and a 12-hour slot overnight. Given the insufficiency of tests to occupy the entire 12-hour span, the 2-hour window was selected. Despite its brevity, this time frame still presents a considerable challenge for evaluating the effectiveness of the evolutionary test selector.

After a manual tuning procedure, the hyperparameter values provided in Table 5.1 were utilized during the evolutionary process. The crossover and mutation probabilities are represented on a range of $[0 - 1]$, and an elitism factor of 1 guarantees that the top-performing individual (the best) from each generation is preserved and carried over to the subsequent generation.

Table 5.1: GA hyperparameters used for each of the 30 runs. Selected by manual tuning.

Hyperparameter	Value
Population size	100
Number of generations	1000
Crossover prob.	0.9
Mutation prob.	0.05
Tournament size	4
Elitism factor	1

The implemented GA algorithm is compared with a **Random Search** algorithm. This Random Search algorithm employs identical fitness function and weights as the GA. The divergence lies in the process: while the GA evolves a population through crossover and mutation, the random search method iteratively samples new solutions from the search space and retains the best solution in terms of fitness. To guarantee a fair assessment, the random search was required to generate an identical number of individuals as the GA, specifically one hundred thousand individuals. This equivalence was necessary since the GA evolves a population comprising 100 solutions over the course of 1000 generations.

5.2 Results and Analysis

This section presents the outcomes derived from running the algorithm 30 times. Within each run, a total of 1000 generations, each consisting of 100 individuals, undergo evolutionary development. The best individual solution from the final generation of each of the 30 runs is subsequently harnessed to compute the final results, as showcased in Table 5.2. The "relative" values exhibited in the table are proportional to the highest achievable value for each attribute. It's important to note that in the case of execution time, the relative value is linked to the time frame limit, implying that the execution time could surpass 100% of the designated time frame. Moreover, a statistical Mann-Whitney U test was carried out to evaluate the difference between the outcomes of these two approaches.

Table 5.2: Resulting attribute values from the final solution of each of the 30 runs of the GA and Random Search. Values in bold have significant statistical difference when compared to the other approach.

Approach	GA				Random Search			
Attribute	Absolute value		Relative value		Absolute value		Relative value	
	Mean	Std.	Mean	Std	Mean	Std	Mean	Std.
Number of unique steps	**375.63**	7.07	**93.90**%	1.76%	337.86	12.76	84.44%	3.19%
Number of verified conditions	**1554.80**	31.27	**80.76**%	1.62%	1332.46	73.78	69.22%	3.83%
Number of tests	**214.66**	4.72	**72.52**%	1.59%	198.43	12.42	67.03%	4.19%
Estimated execution time (seconds)	7157.44	70.95	99.40%	0.98%	7191.29	541.73	99.87%	7.52%

The data provided in the table indicates that the chosen 72% of the test suite gathered approximately 94% of the possible unique steps and 81% of the verified conditions, all while utilizing nearly 99% of the available time allocation. The algorithm consistently outperformed the Random Search baseline across most of the evaluated aspects, with exception of the subsets execution time, which were both very close to 100%, although the standard variation of the random search for this attribute is considerably higher than the GA's, at 541 seconds, compared to the 70 seconds of the latter, indicating that the GA is more successful in achieving subsets execution time close (and bellow) to the time limit.

The GA achieved a notable improvement over the baseline's performance metrics: a nearly 9 percentage point advantage in the number of unique steps, approximately 12 percentage points higher in terms of verified conditions, and approximately 5 percentage points more tests present in the generated subsets. Moreover, it is important to analyze the performance of both approaches in respect to the time that it takes to run these algorithms. Figure 5.5 shows the comparison between the execution times of both GA and random search.

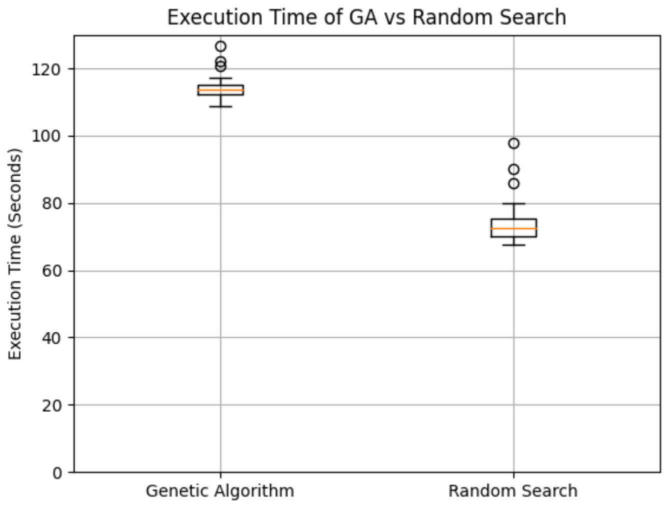


Figure 5.5: Comparison of the distribution of algorithm execution times between the GA and the random search.

The GA takes in average 114 seconds for each run, whereas the random search takes 74 seconds, 40 seconds less than the former approach. This is related to the much more complex structure of the GA, which performs selection, crossover and mutation operations within the population, demanding considerably more time than the random search, that randomly generates a new individual at each iteration, without any other operations. To better analyze the GA results, Figures 5.6 and 5.7 showcase the obtained fitness and attributes evolution, respectively, from the 30 runs of the GA.

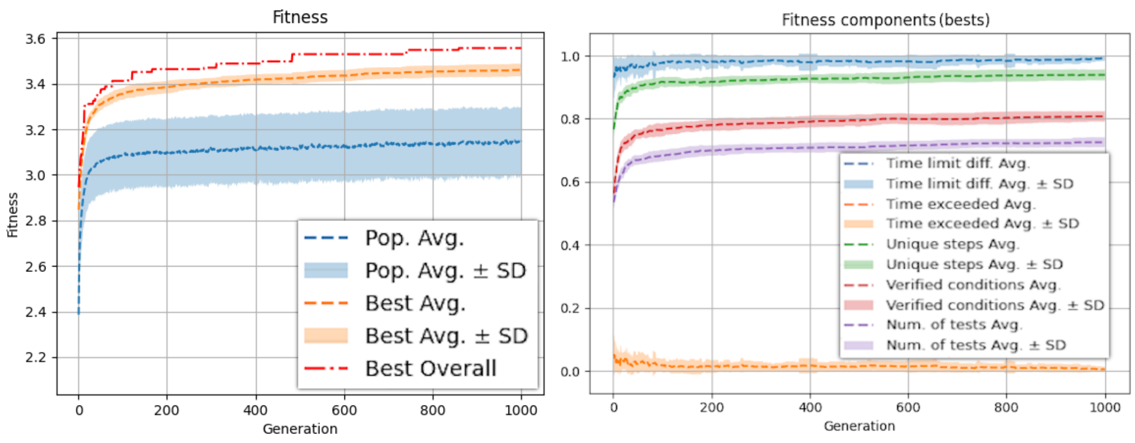


Figure 5.6: Fitness evolution analysis of the 30 runs across the 1000 generations. In the left, two groups are shown, the global 100-individual population in blue, the bests of each generation in orange, and in red the best over all runs for each generation. In the right, the fitness represented by the orange group of the left plot is decomposed into the five fitness components.

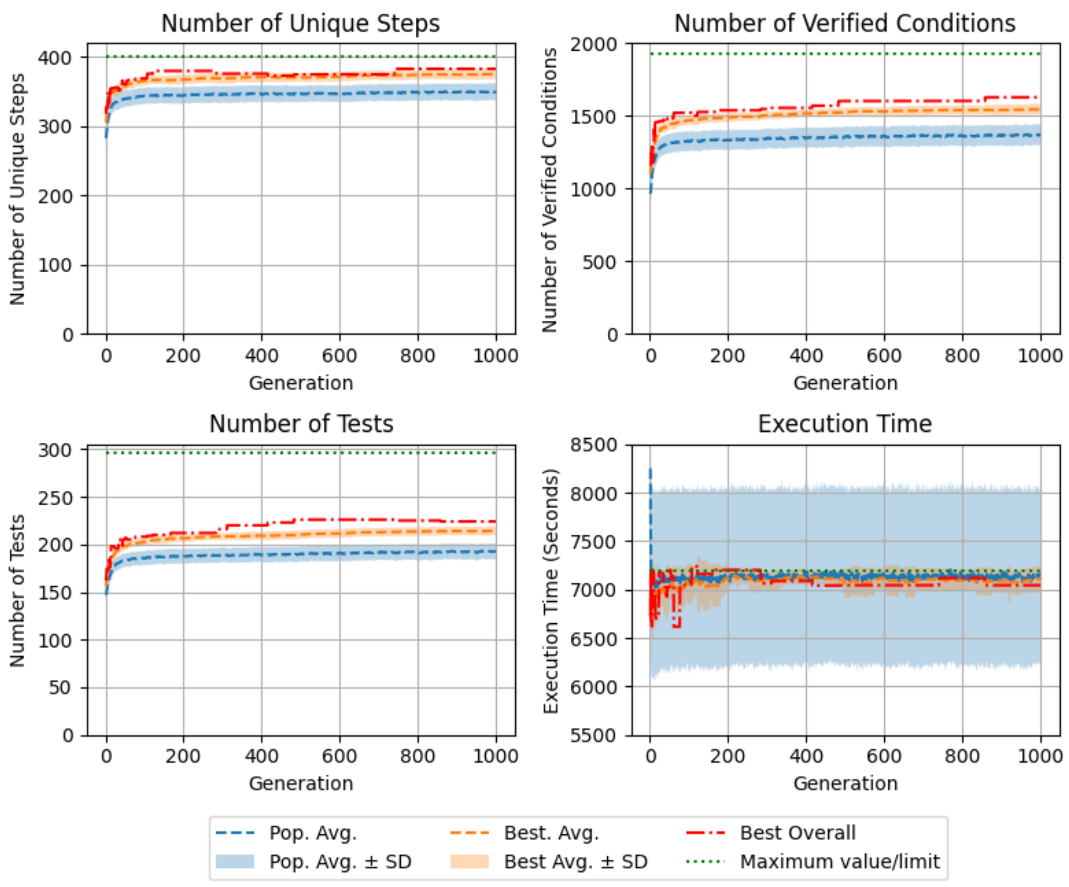


Figure 5.7: Attribute evolution analysis of the 30 runs across the 1000 generations. In clock-wise direction, starting from the upper-left corner is the number of unique steps, followed by the number of verified conditions, with the execution time next, and finally the number of tests. The green line represents the maximum attainable value, except for the execution time, where it represents the time limit.

These visuals illustrate the evolutionary progress for three distinct groups of the population: the entire population (comprising 100 individuals per generation), the best individuals (1 from each generation of each run), and the overall best (the top individual from each generation across all runs). An exception is made in the fitness components plot, where solely the best individuals are taken into account. This means that the summation of these fitness components is equivalent to the fitness of the bests in the first plot.

These figures indicate a discernible pattern in the evolution process. A rapid increase in fitness is evident during the initial 100 generations, succeeded by a gradual yet consistent growth in subsequent generations. All fitness components and attributes exhibit enhancement over the course of generations, except for the execution time. The execution time consistently aligns with the time limit line, displaying varying degrees of deviations. However, it's noteworthy that both the population average and the average of the best individuals remain below the stipulated limit.

In Table 5.3, we examine the best final solution derived from the 30 runs of both the GA and random search methods.

Table 5.3: Comparison between the values obtained with the best solution from both GA and Random Search. Additionally, the percentage differences between this results and the average results from Table 5.2 are shown. The values in bold indicate the best result for each attribute.

Approach	GA			Random Search		
Attribute	Absolute value	Relative value	Difference from average (percentage points)	Absolute value	Relative value	Difference from average (percentage points)
Number of unique steps	**385**	**96.25%**	+2.35pp	341	85.25%	+0.81pp
Number of verified conditions	**1559**	**80.98%**	+0.22pp	1520	79.16%	+9.94pp
Number of tests	218	73.64%	+1.12pp	**231**	**78.04%**	+11.01pp
Estimated execution time (seconds)	**7191.86**	**99.88%**	+0.48pp	7458.70	103.59%	+3.72pp

From this comparison, it can be concluded that in one of its runs, the random search achieved a solution relatively close to the GA's, even outperforming the GA in terms of the number of tests included in the subset. The only attribute exhibiting a difference of more than two percentage points in favor of the GA was the number of unique steps, in which it outperformed the random search by 11 percentage points. The values attained in this specific solution of the random search significantly exceeded the average outcomes of the same algorithm. Conversely, the values for the GA remained close to the average but still exhibited improvement, showcasing the consistent performance of the GA. Thus, the use of GA is preferable, as good results from random search come by chance and are not guaranteed.

The 30 solutions obtained were submitted to Altice Labs for validation of their execution times. All of the test subsets were executed within durations shorter than the predefined time limit. Figure 5.8 shows the unused time frames and tests across the 30 runs of the GA.

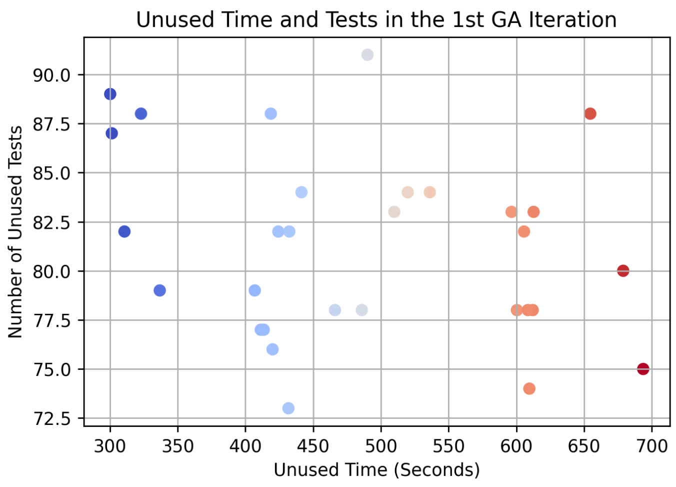


Figure 5.8: Unused time frames and tests across the final solutions from the 30 runs of the single-iteration GA, considering the 7200-seconds time limit and 296-test suite. Each dot represents one of the 30 runs of the algorithm, and the cool-warm gradient is related to the unused execution times.

In average, the unused time and tests with the first GA iteration are approximately 488 seconds and 81 tests, respectively. This outcome paved the path for conducting a second evolutionary iteration for each of the 30 runs. The subsequent iterations involved 1000 generations and retained the other original hyperparameters from the first iteration, except for the time frames. The new time frames are defined by the remaining unused execution time for each subset, and the available tests for selection are confined to those that were not selected in the first iteration. Table 5.4 shows the comparison between the results obtained with one iteration and two iterations of the algorithm.

Table 5.4: Resulting attribute values from the final solution of each of the 30 runs of the single iteration GA and the double-iteration counterpart, taking into account the validation time. Values in bold have significant statistical difference when compared to the other approach.

Approach	Single iteration GA				Double iteration GA			
Attribute	Absolute value		Relative value		Absolute value		Relative value	
	Mean	Std.	Mean	Std	Mean	Std-	Mean	Std.
Number of unique steps	375.63	7.07	93.90%	1.76%	**392.03**	4.05	**98.0%**	1.01%
Number of verified conditions	1554.80	31.27	80.76%	1.62%	**1776.06**	15.50	**92.26%**	0.80%
Number of tests	214.66	4.72	72.52%	1.59%	**267.36**	4.61	**90.32%**	1.55%
Validation execution time (seconds)	6711.50	117.55	93.21%	1.63%	**7172.91**	32.75	**99.62%**	0.45%

The second iteration, utilizing the remaining time and tests, yields a substantial enhancement in the algorithm's outcomes. It notably elevates the proportion of tests in the subset from 72% to 90% of the total test suite. Moreover, the number of unique steps almost attains its maximum value, reaching 98%. This observation underscores the effective reduction of redundancy among tests. However, a critical analysis is required to assess the trade-off between the benefits gained from the second iteration and the associated execution costs in comparison to the single iteration algorithm. Figure 5.9 shows the comparison between the execution times of the first iteration of the algorithm and the second iteration.

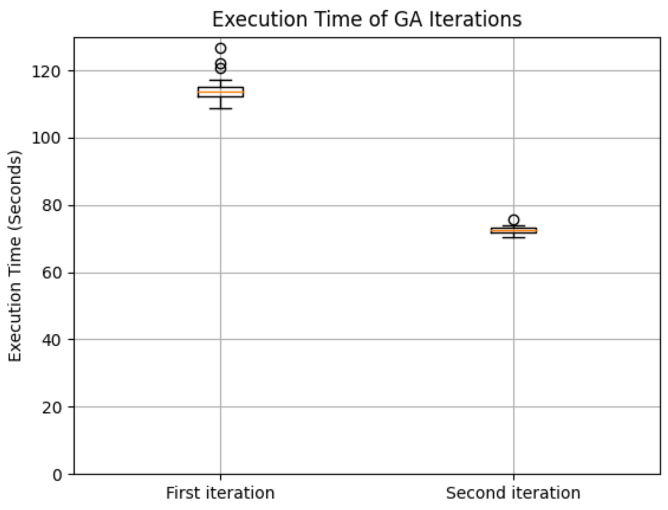


Figure 5.9: Time that takes to perform the test subset optimization, in seconds, among the GA with a single iteration and the GA with two iterations.

As expected, the initial iteration of the algorithm demands significantly more time compared to the second iteration, given that the latter involves a reduced number of tests. On average, the first iteration requires around 114 seconds, whereas the second iteration takes approximately 72 seconds. As previously observed, the average validation time for the initial iteration of the algorithm is 6711 seconds for the 296-test batch. This value is approximately 500 seconds below the 7200 seconds limit. Consequently, the additional 72 seconds required for a second iteration represents on average 14.4% of the surplus time. This proportion can be deemed as an acceptable cost, considering the benefits from the second iteration's enhancements.

It's worth highlighting that the recorded algorithm execution times were obtained using a computer equipped with an Intel Core i5 processor and 16 GB of RAM. Additionally, the executions took place within a Jupyter environment, which could potentially introduce slowdowns due to its reliance on a web browser for visualization, consuming a significant amount of RAM. Thus, it is important to acknowledge that this algorithm holds the potential to achieve faster performance with more robust computational resources and further optimizations in the code.

In summary, the obtained outcomes were regarded as satisfactory, highlighting the algorithm's potential for real-world application. Furthermore, by adjusting the weights, it becomes feasible to prioritize the enhancement of one attribute at the cost of others. Nonetheless, the results exhibited here were well received by OLT testers.

5.3 Next Steps

This section outlines potential paths for future development. The initial section highlights specific points for enhancement within the suggested evolutionary test subset optimizer, while the subsequent section provides a discussion on possible strategies to fulfil the implementation of the yet-to-be-achieved automated test generator.

There are specific challenges that need attention in future developments. Among them is the evaluation methodology employed, which was based on the insights and viewpoints of a limited number of OLT testers. There were no access to technical documentation about the OLTs and tests, leading to the design of a fitness function with a high level of technical abstraction. The addition of more information from the development process is essential to formulate novel test set evaluation methods that encompass the intricate interplay between tested functionalities and various dimensions of OLT development, such as security, integrity, and availability [42].

Moreover, the fitness evaluation approach in this work simplified the optimization problem into a single-objective GA. However, the literature highlights the advantages of employing multi-objective GAs to yield multiple solutions within the context of the Pareto optimal vector [43, 44]. Future iterations of this problem could benefit from such algorithms, which include the previously mentioned and well-established NSGA-II [17], the Multiobjective Evolutionary Algorithm Based on Decomposition (MOEAD/D) [45], and the Strength Pareto Evolutionary Algorithm 2 (SPEA2) [46].

Another aspect requiring enhancement relates to the scope of this approach, which concentrated solely on tests suitable for a specific OLT model within a particular scenario (an OLT2T0 with two ONUs and no traffic generator). To bolster the validation of this selection algorithm, it would be compelling to integrate a more real-life representative array of tests. Furthermore, it is essential to investigate the algorithm's performance when dealing with larger test suites. This is particularly relevant since the test suites utilized by Altice encompass thousands of tests, whereas the subset examined in this work comprises a mere 296 tests.

An additional path for improvement pertains to the creation of the automatic test generator. It is held that such an algorithm holds promise in enhancing the testing procedure for the OLTs. It has the potential to expedite the test creation process by providing tests that efficiently validate new requirements.

The creation of tests revolves around new requirements. Within the Altice Labs framework, OLT software developers offer textual descriptions of the requirements to the testing team, who subsequently devise the new tests. These descriptions of requirements tend to be unstructured and abstract, leaning on the expertise of both developers and testers to understand their meaning. An illustrative excerpt from requirement 14098 is presented below:

> *It must be possible to configure the MAC aging time in units of seconds (global system parameter). The allowed range varies between 15 to 21600 seconds.*

Elaborate requirements encompassing multiple parameters typically include tables illustrating possible values for each parameter. Conversely, some requirements may not entail parameters, but rather pertain to functionalities, and exhibit straightforward descriptions like the one exemplified in requirement 7354:

> *The OLT2T0 must support SSH protocol.*

However, certain requirements could span multiple paragraphs of explanation and incorporate numerous tables. In brief, the present condition of requirement descriptions at Altice is characterized by a high-level language, which abstracts a substantial portion of the technical intricacies inherent to the OLT system.

Recent progress in code generation from natural language has been achieved using Large Language Model (LLM) systems [47], which are AI systems capable of processing and generating text by leveraging large datasets and advanced neural networks like GPT-3 [48]. These LLMs can comprehend natural language, infer context, and produce coherent code in various programming languages. This technology has found application in software testing, where LMs are used to automatically test user interfaces and to generate code and software tests [49, 50, 51]. Overall, the feasibility of creating an automated test generator for specific scenarios using LLMs is promising, but at the same time it implies that the requirements for which the tests are create must be described in a more cohesive and complete manner. This approach, combined with the test subset optimizer, can further improve the testing process of OLTs and potentially other systems as well.

This page is intentionally left blank.

Chapter 6

Conclusion

Software testing stands as a pivotal phase in software development, serving as a crucial means to ensure the reliability and optimal functioning of any system. However, it often demands considerable time, financial resources, and dedicated effort from the development team. It's not uncommon for test suites to encompass thousands of individual tests, leading to challenges in executing them within limited time constraints. Hence, the selection of the most pertinent tests for execution is of great importance in such scenarios.

In the context of this project, Altice, a multinational telecommunications company, employs modern PON technology to provide diverse services to its customers. PONs are centered around OLTs, which are used to control the data flow in these complex networks, converting electric signals from the core network to optical signals for the PONs. The development and upkeep of OLTs constitute demanding responsibilities that involve continuous engagement from software developers and testers. This process also results in an extensive collection of tests, often exceeding feasible execution time frames.

This work introduces a GA-based approach for optimizing test subsets in the context of OLTs. The complexity stems from the Gherkin metalanguage used to articulate test cases, which poses a difficulty due to its elevated, human-readable style, thus rendering programmatic instrumentation of tests challenging. Additionally, only the tests themselves and past execution times are at disposal, with no supplementary information at hand.

In close collaboration with OLT testers, a set of evaluation metrics were formulated to guide the selection of optimal test subsets. The objective was to simultaneously enhance the number of tests executed within a designated time frame and maximize test utility according to these custom-designed criteria. The GA was then tested using a pool of 296 tests, and a fitness function was devised by combining all the established evaluation criteria into a weighted summation of fitness components linked to the attributes there were being optimized. The conclusive findings indicated a favorable outcome, as all the attributes of the test sets fell within an acceptable range. The partner company's testers evaluated the results as satisfactory, signaling their endorsement for future adoption of the algorithm.

In subsequent iterations, potential enhancements might involve refining the information utilized for test evaluation by incorporating more comprehensive and contextual data, including the significance of tested requirements. Furthermore, delving into a multi-objective GA approach could potentially yield superior outcomes and offer enhanced mechanisms for discerning among Pareto optimal solutions. Finally, developing an automatic test generator may further improve the testing process, by providing tests that better validate new system requirements.

This page is intentionally left blank.

References

[1] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering*, 27(2), 2022. ISSN 15737616.

[2] Hardik Shah. How Much Does Software Testing Cost? 9 Proven Ways to Optimize it. `https://www.simform.com/blog/software-testing-cost/`, 2021. Visited in December 2022.

[3] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010. ISSN 09505849.

[4] Cedric F Lam. Passive Optical Networks. In Cedric F Lam, editor, *Academic Press*, pages 1–17. Academic Press, Burlington, 2007. ISBN 978-0-12-373853-0.

[5] Anu Bajaj and Om Prakash Sangwan. A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms. *IEEE Access*, 7:126355–126375, 2019. ISSN 21693536.

[6] Ayesha Kiran, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A Comprehensive Investigation of Modern Test Suite Optimization Trends, Tools and Techniques. *IEEE Access*, 7:89093–89117, 2019. ISSN 21693536.

[7] Huda Saleh Abbas and Mark A. Gregory. The next generation of passive optical networks: A review. *Journal of Network and Computer Applications*, 67:53–74, 2016. ISSN 10958592. doi: 10.1016/j.jnca.2016.02.015. URL `http://dx.doi.org/10.1016/j.jnca.2016.02.015`.

[8] Altice. OLT2Tx. https://www.youtube.com/watch?v=cF27UlqtKWY, 2022. Visited in January 2023.

[9] Yan Wang, Lijuan Jia, Hongjian Cao, Ziqi Jing, and Huan Huang. Applications of Cucumber on Automated Functional Simulation Testing. *Proceedings - 2021 21st International Conference on Software Quality, Reliability and Security Companion, QRS-C 2021*, pages 861–862, 2021.

[10] Anne Benoit, Yves Robert, and Frédéric Vivien. Greedy algorithms. *A Guide to Algorithm Design*, pages 71–98, 2020.

[11] Xingya Wang, Shujuan Jiang, Pengfei Gao, Xiaolin Ju, Rongcun Wang, and Yanmei Zhang. Cost-effective testing based fault localization with distance based test-suite reduction. *Science China Information Sciences*, 60(9):1–15, 2017. ISSN 18691919.

[12] Chu-Ti Lin, Kai-Wei Tang, Jiun-Shiang Wang, and Gregory M. Kapfhammer. Empirically evaluating Greedy-based test suite reduction methods at different levels of test suite complexity. *Science of Computer Programming*, 150:1–25, 2017.

[13] Ali Yamuç, M. Özgür Cingiz, Göksel Biricik, and Oya Kalipsiz. Solving test suite reduction problem using greedy and genetic algorithms. *Proceedings of the 9th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2017*, 2017-January:1–5, 2017.

[14] Johann Dréo. Classifications of metaheuristics, 2007. URL `http://metah.nojhan.net/post/2007/10/12/Classification-of-metaheuristics`. Visited in August 2023.

[15] Pradnya A. Vikhar. Evolutionary algorithms: A critical review and its future prospects. *Proceedings - International Conference on Global Trends in Signal Processing, Information Computing and Communication, ICGTSPICC 2016*, pages 261–265, 2017.

[16] Rongqi Pan, Taher A. Ghaleb, and Lionel Briand. ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1700–1711, 2022. URL `http://arxiv.org/abs/2210.16269`.

[17] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. ISSN 1089778X.

[18] Remo Lachmann, Sandro Schulze, Michael Felderer, Christoph Seidl, Manuel Nieke, and Ina Schaefer. Multi-objective black-box test case selection for system testing. *GECCO 2017 - Proceedings of the 2017 Genetic and Evolutionary Computation Conference*, (December):1311–1318, 2017.

[19] Xue Ying Ma, Bin Kui Sheng, Zhen Feng He, and Cheng Qing Ye. A genetic algorithm for test-suite reduction. In *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, volume 1, pages 133–139, 2005.

[20] Prem Jacob and T Ravi. Optimal regression test case prioritization using genetic algorithm. *Life Science Journal*, 10(3):1021–1033, 2013.

[21] Arvinder Kaur and Shubhra Goyal. A Genetic Algorithm for Fault based Regression Test Case Prioritization. *International Journal of Computer Applications*, 32(8):975–8887, 2011.

[22] Arvinder Kaur and Shubhra Goyal. A genetic algorithm for regression test case prioritization using code coverage. *International journal on computer science and engineering*, 3(5):1839–1847, 2011.

[23] Getachew Mekuria Habtemariam and Sudhir Kumar Mohapatra. A Genetic Algorithm-Based Approach for Test Case Prioritization. In Fisseha Mekuria, Ethiopia Nigussie, and Tesfa Tegegne, editors, *Information and Communication Technology for Development for Africa*, pages 24–37, Cham, 2019. Springer International Publishing. ISBN 978-3-030-26630-1.

[24] S. Nachiyappan, A. Vimaladevi, and C. B. Selvalakshmi. An evolutionary algorithm for regression test suite reduction. *Proceedings of 2010 International Conference on Communication and Computational Intelligence, INCOCCI-2010*, pages 503–508, 2010.

[25] Carmen Coviello, Simone Romano, Giuseppe Scanniello, and Giuliano Antoniol. GASSER: Genetic Algorithm for TeSt Suite Reduction. In *Proceedings of the 14th*

ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375801.

[26] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines. *IEEE Access*, 6(ii):11816–11841, 2018. ISSN 21693536.

[27] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: An overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2 (1):86–97, 2012. ISSN 19424795.

[28] Carmen Coviello, Simone Romano, and Giuseppe Scanniello. Poster: CUTER: ClUstering-based test suite reduction. *Proceedings - International Conference on Software Engineering*, (3):306–307, 2018. ISSN 02705257.

[29] J A Hartigan and M A Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979. ISSN 00359254, 14679876.

[30] Nour Chetouane, Franz Wotawa, Hermann Felbinger, and Mihai Nica. On Using k-means Clustering for Test Suite Reduction. *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020*, pages 380–385, 2020.

[31] Pavi Saraswat and Abhishek Singhal. A hybrid approach for test case prioritization and optimization using meta-heuristics techniques. *India International Conference on Information Processing, IICIP 2016 - Proceedings*, pages 1–6, 2017.

[32] J Kennedy and R Eberhart. Particle Swarm Optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, 4:1942–1948, 1995.

[33] Dongshu Wang, Dapei Tan, and Lei Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22(2):387–408, 2018. ISSN 14337479.

[34] Shweta Singhal, Bharti Suri, and Sanjay Misra. An empirical study of regression test suite reduction using MHBG-TCS tool. *Proceedings of the IEEE International Conference on Computing, Networking and Informatics, ICCNI 2017*, 2017-January (October):1–5, 2017.

[35] Dusan Teodorovic, Panta Lucic, Goran Markovic, and Mauro Dell' Orco. Bee Colony Optimization: Principles and Applications. In *2006 8th Seminar on Neural Network Applications in Electrical Engineering*, pages 151–156, 2006. doi: 10.1109/NEUREL.2006.341200.

[36] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization. *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, (August):367–378, 2017.

[37] Francisco Palomo-Lozano, Inmaculada Medina-Bulo, Antonia Estero-Botaro, and Manuel Núñez. Test suite minimization for mutation testing of WS-BPEL compositions. *GECCO 2018 - Proceedings of the 2018 Genetic and Evolutionary Computation Conference*, pages 1427–1434, 2018.

[38] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. DetReduce: Minimizing Android GUI test suites for regression testing. *Proceedings - International Conference on Software Engineering*, 2018-January, 2018. ISSN 02705257.

[39] Jun Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. *Proceedings - International Conference on Software Engineering*, (1):1039–1049, 2018. ISSN 02705257.

[40] P. Larrañaga, C. M.H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999. ISSN 02692821.

[41] Oliver Kramer. *Genetic Algorithm Essentials*. Springer, 2017.

[42] Mohammad M. Rad, Kerim Fouli, Habib A. Fathallah, Leslie A. Rusch, and Martin Maier. Passive optical network monitoring: Challenges and requirements. *IEEE Communications Magazine*, 49(2):s45–S52, 2011. ISSN 01636804.

[43] Hisao Ishibuchi, Yusuke Nojima, and Tsutomu Doi. Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures. *2006 IEEE Congress on Evolutionary Computation, CEC 2006*, (2):1143–1150, 2006.

[44] Wei Zheng, Xiaoxue Wu, Xibing Yang, Shichao Cao, Wenxin Liu, and Jun Lin. Test suite minimization with mutation testing-based many-objective evolutionary optimization. *Proceedings - 2017 Annual Conference on Software Analysis, Testing and Evolution, SATE 2017*, 2017-January:30–36, 2017.

[45] Qingfu Zhang and Hui Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007. ISSN 1089778X.

[46] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, pages 95–100, 2001.

[47] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. *Conference on Human Factors in Computing Systems - Proceedings*, 2022.

[48] Luciano Floridi and Massimo Chiriatti. GPT-3: Its Nature, Scope, Limits, and Consequences. *Minds and Machines*, 30(4):681–694, 2020. ISSN 15728641. URL `https://doi.org/10.1007/s11023-020-09548-1`.

[49] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1355–1367, 2022.

[50] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code Generation with Generated Tests. pages 1–19, 2022.

[51] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon

Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 2020-December, 2020. ISSN 10495258.